



**M.Sc.Computer Science**

**PAPER - 6**

**INTERNET PROGRAMMING  
AND WEB DESIGN**

**BHARATHIAR UNIVERSITY**

**SCHOOL OF DISTANCE EDUCATION**

**COIMBATORE**



## **PAPER- 6 INTERNET PROGRAMMING AND WEB DESIGN**

**Subject Description:** This Paper presents introduction to internet, HTML, Java script and Dynamic HTML.

**Goals:** To enable the students to write programs for internet and to develop web applications.

**Objectives:** On successful completion of the student should have:

- ★ Understood the internet.
- ★ Learnt HTML. Intermediate HTML and dynamic HTML
- ★ Learnt Java Script.

### **Unit I**

**Introduction to computers and the Internet:** history of the world wide web  
Hardware trends – The say software trend: Object Technology – Java Script:  
Object – based scripting for the web – browser portability.

**Introduction to HTML :** Introduction – markup language – editing HTML – common tags – headers – text styling – linking images – formatting text with <FONT> special characters, horizontal rules and more line breaks – internet and www resources.

**Intermediate HTML :** Introduction – Unordered Lists – nested and ordered lists – basic HTML tables – intermediate HTML tables and formatting – basic HTML forms – more complex HTML forms – internal linking – creating and using images maps <META> Tags, <FRAMESET> tags – internet and www resources.

### **Unit II**

Java Script – Introduction to scripting: Introduction – memory concepts – arithmetic – decision making – java script internet & www resources.

**Java script control structures :** If , if / else selection structure while, for do/ while repetition structure – switch multiple – selection structure – break and continue Statements – Laballed Break and continue Statements – Logical Operators.

**Java Script Functions :** Introduction – Program Modules in Java Script – programmer – Defined Functions – Functions – Duration of Identifiers – Scope Rules – Recursion – Recursion Vs Iteration – Java Script Global Functions.

### **Unit III**

**Java Script Arrays:** Introduction – Arrays – Declaring and Allocating Arrays – References and References Parameters – Passing Arrays to functions – Sorting Arrays – searching Arrays – Multiple Subscripted Arrays.

**Java Script Objects:** Introduction – Thinking about Objects – Math String, Data, boolean and Number Objects.

**Dynamic HTML :** CSS : Introduction – Inline Styles – Creating Style Sheets with the Style Element – Conflicting Styles – Linking External Style Sheets – Positioning Elements – Backgrounds – Element Dimensions – Text flow and the Box model – user Style Sheets – Internet & www resources.

## Unit IV

**Dynamic HTML:** Object model and collections: Introduction – Object Referencing – Collections all and Children – Dynamic Styles – Dynamic Positioning – using the Frames Collection – navigator object.

**Dynamic HTML :** Event model : Introduction – event ON CLICK – Event ON LOAD – error handling with ON ERROR – Tracking the mouse with event ON MOUSE MOVE – Rollovers with ON MOUSE OVER and ONBLUR – more form processing with ON SUBMIT and ON RESET – event Bubbling more DHTML events.

**Dynamic HTML :** Filters and Transitions : Introduction – Flip filters : Flipu and Fliph – transparency with the Chroma filter – Creating Images filters : Invert, Gray and x ray – Adding Shadows to Text – Creating Gradients with Alpha – Making Text Glow – Creating Motion with blur – using the Wave filter – Advanced filters : Drop Shadow and Light – Transitions I : Filter Transition II : Filter Reveal Trans.

**Dynamic HTML :** client Side Scripting with VB Script : Introduction – Operators – Data Types and Control Structures – VB Script Functions – Arrays – String manipulation Classes and Objects – Internet & www resources.

## Unit V

**Active Server Pages (ASP):** Introduction – How ASP Work – Client – Side Scripting Versus – Server Side Scripting – Using Personnel Web Server or Internet Information Server – Server – Side Activex Components – File System Objects Session Tracking and cookies – Accessing a Database form an ASP – Internet & www resources.

CGI and Perl : CGI - Introduction to Perl – Configuring Personal Web Server or Perl/CGI – String Processing and Regular Expressions – Viewing Client/ Server Environment Variables – Form Processing and Business Logic – Server – Side Includes – verifying a username and password – sending E-Mail from a web browser – using ODBC to connect to a Database – cookies and Perl – Internet & www resources.

**XML :** Introduction – Structuring Data – Document Type Definitions – Customized Markup Language – XML Parsers – XHTML – Internet & www resources.

### Reference Books:

1. Deitel, Deitel, Nieto, “Internet and World Wide Web – How to program”, Pearson Education Asia, 2003.
2. Thomas A. Powell, “The Complete Reference HTML and XHTML”, fourth Edition, Tata McGraw Hill Pub. Company Ltd.
3. Achyut s. Godbole, Atul Kahate, “Web Technologies – TCP / IP to Internet Application Architectures”, Tata McGraw – Hill Pub. Company Ltd, 2003.

# **CONTENTS**

<b>Lesson 1 : Introduction to Internet and World Wide Web</b>	<b>1</b>
<b>Lesson 2 : Introduction to HTML</b>	<b>9</b>
<b>Lesson 3 - Intermediate HTML</b>	<b>39</b>
<b>Lesson 4: JavaScript – Basics</b>	<b>67</b>
<b>Lesson 5 : JavaScript – Control Structures</b>	<b>89</b>
<b>Lesson 6 : JavaScript – Functions</b>	<b>105</b>
<b>Lesson 7: JavaScript – Arrays</b>	<b>117</b>
<b>Lesson 8: JavaScript – Objects</b>	<b>135</b>
<b>Lesson 9: Cascading Style Sheet (CSS)</b>	<b>157</b>
<b>Lesson 10: Dynamic HTML: Object Model and Collections</b>	<b>179</b>
<b>Lesson 11: Dynamic HTML - Event Model</b>	<b>197</b>
<b>Lesson 12 : Java Script - Filters and Transitions</b>	<b>211</b>
<b>Lesson 13: VBScript – I</b>	<b>227</b>
<b>Lesson 14 : VBScript – II</b>	<b>257</b>
<b>Lesson 15 : Active Server Pages – I</b>	<b>297</b>
<b>Lesson 16: ASP – using ODBC</b>	<b>347</b>
<b>Lesson 17 : CGI and PERL</b>	<b>367</b>
<b>Lesson 18 : PERL ODBC and Cookies</b>	<b>405</b>
<b>Lesson 19 : XML</b>	<b>425</b>



---

## Lesson 1

# Introduction to Internet and World Wide Web

---

### Content

#### 1.0 Aim and Objective

#### 1.1. Introduction

#### 1.2. History of the Internet

#### 1.3. History of the World Wide Web

#### 1.4. World Wide Web Consortium (W3C)

#### 1.5. Hardware Trends

#### 1.6. Software Trend : Object Technology

#### 1.7. JavaScript : Object based Scripting for the Web

#### 1.8. Browser Portability

#### 1.9. Let us sum Up

#### 1.10. Lesson end Activities

#### 1.11. Check your progress

#### 1.12. Reference

---

#### 1.0 Aim and Objective

#### 1.1 Introduction

---

- **To understand Internet concepts**
- **To learn the software and hardware trends**

---

#### 1.2. History of the Internet

---

The Internet is a collection of networks, including the Arpanet, NSFnet, regional networks such as NYsernet, local networks at a number of University and research institutions, and a number of military networks.

The term "Internet" applies to this entire set of networks. The subset of them that is managed by the Department of Defense is referred to as the "DDN" (Defense Data Network). This includes some research-oriented

networks, such as the Arpanet, as well as more strictly military ones. All of these networks are connected to each other. Users can send messages from any of them to any other, except where there is security or other policy restrictions on access. Officially speaking, the Internet protocol documents are simply standards adopted by the Internet community for its own use. More recently, the Department of Defense issued a MILSPEC definition of TCP/IP. This was intended to be a more formal definition, appropriate for use in purchasing specifications. However most of the TCP/IP community continues to use the Internet standards. The MILSPEC version is intended to be consistent with it.

Whatever it is called, TCP/IP is a family of protocols. TCP/IP is a set of protocols developed to allow cooperating computers to share resources across a network. It was developed by a community of researchers centered on the ARPAnet. Certainly the ARPAnet is the best-known TCP/IP network. However as of June, 87, at least 130 different vendors had products that support TCP/IP, and thousands of networks of all kinds use it.

Thus the most important TCP/IP services are:

### **File Transfer**

The file transfer protocol (FTP) allows a user on any computer to get files from another computer, or to send files to another computer. Provisions are made for handling file transfer between machines with different character set, end of line conventions, etc.

### **Remote login**

The network terminal protocol (TELNET) allows a user to log in on any other computer on the network. You start a remote session by specifying a computer to connect to. From that time until you finish the session, anything you type is sent to the other computer. Generally, the connection to the remote computer behaves much like a dialup connection.

### **Email**

This allows you to send messages to users on other computers. Originally, people tended to use only one or two specific computers. They



would maintain "mail files" on those machines. The computer mail system is simply a way for you to add a message to another user's mail file.

### **Network File Systems**

This allows a system to access files on another computer. A network file system provides the illusion that disks or other devices from one system are directly connected to other systems. There is no need to use a special network utility to access a file on another system. Your computer simply thinks it has some extra disk drives. These extra "virtual" drives refer to the other system's disks.

### **Remote Printing**

This allows you to access printers on other computers as if they were directly attached to yours.

### **Remote Execution**

This allows you to request that a particular program be run on a different computer. This is useful when you can do most of your work on a small computer, but a few tasks require the resources of a larger system.

### **Name Servers**

In large installations, there are a number of different collections of names that have to be managed. This includes users and their passwords, names and network addresses for computers, and accounts. It becomes very tedious to keep this data up to date on all of the computers. Thus the databases are kept on a small number of systems. Other systems access the data over the network.

### **Terminal Servers**

Many installations no longer connect terminals directly to computers. Instead they connect them to terminal servers. A terminal server is simply a small computer that only knows how to run telnet. If your terminal is connected to one of these, you simply type the name of a computer, and you are connected to it. Generally it is possible to have active connections to more than one computer at the same time.

---

## **1.3. History of the World Wide Web**

---

The World Wide Web allows computer users to locate and view multimedia based documents on almost any subject. Even though the Internet was developed decades ago, the introduction of the World Wide Web is a relatively recent event. In 1990 TIM Berners-Lee of CERN developed the World Wide Web and several communication protocols that form the backbone of the Web. The use of the Web exploded with the availability in 1993 of the Mosaic browser, which featured a user-friendly graphical interface. Marc Andreessen, whose team at the National Center for Supercomputing Applications developed Mosaic, went on to found Netscape, the company that many people credit with initiating the explosive Internet economy of the late 1990s.

---

#### **1.4. World Wide Web Consortium (W3C)**

---

In October 1994, Tim Berners-Lee founded an organization - called World Wide Web Consortium (W3C) devoted to developing nonproprietary, interoperable technologies for the World Wide Web. One of the W3C's primary goals is to make the Web universally accessible – regardless of ability, language or culture.

The W3C is also a standardization organization. Web technologies standardized by the W3C are called Recommendations. W3C Recommendations include the Extensible HyperText Markup Language (XHTML), Cascading Style Sheets (CSS) and Extensible Markup Language. A recommendation is not an actual software product, but a document that specifies a technology's role, syntax rules and so forth.

The W3C comprises three primary hosts – the Massachusetts Institute of technology(MIT), the European Research Consortium for Informatics and Mathematics(ERCIM) and Keio University in Japan – and hundreds of members. The W3C home page ([www.w3.org](http://www.w3.org)) provide extensive resources on Internet and Web technologies.

---

#### **1.5. Hardware Trends**

---

The Internet community thrives on the continuing stream of dramatic improvements in hardware, software and communications technologies. Every year, the capacities of computer tend to double, especially the amount of

memory they have in which to execute programs, the amount of secondary storage they have to hold programs and data over the longer term, and the processor speeds.

Recently, the hardware has been moving more and more towards mobile, wireless technology. Small hand-held devices are now more powerful than the super computers of the early 1970s. Portability has become a major focus for the computer industry. Wireless data transfer speeds have become so fast that many Internet users' primary access to the Web is through local wireless networks.

---

### **1.6. Software Trend : Object Technology**

---

Objects are essentially reusable software components that model real-world items. Software developers are discovering that using a modular, object oriented design and implementation approach can make software development groups much more productive than were possible with previous popular programming techniques, such as structured programming. Object-oriented programs are often easier to understand, correct and modify.

With object technology, properly designed software tends to be more reusable in future projects. Libraries of reusable components, such as Microsoft Foundation Classes(MFC), Sun Microsystems's Java Foundation Classes, Microsoft's .NET Framework Class Library (FCL) and those produced by other software development organization can greatly reduce the effort it takes to implement certain kinds of systems.

---

### **1.7. JavaScript : Object Based Scripting for the Web**

---

**JavaScript** is an object-based scripting language with strong support for proper software engineering techniques. JavaScript is a powerful scripting language. Experienced programmers sometimes take pride in creating strange, contorted, convoluted JavaScript expressions.

JavaScript was created by Netscape. Microsoft's version of JavaScript calls Jscript. Both Netscape and Microsoft have been instrumental in the standardization of JavaScript and Jscript by ECMA.

---

## **1.8. Browser Portability**

---

Ensuring a consistent look and feel on client browser is one of the great challenges of developing Web-based applications. Currently, a standard does not exist to which software developers must adhere when creating Web browsers. Although browsers share a common set of features, each browser might render pages differently. Browsers are available in many version and on many different platforms. It is difficult to develop Web pages that render correctly on all version of each browser.

---

## **1.9. Let us Sum Up**

---

The Internet is a collection of networks. The term "Internet" applies to this entire set of networks. The subset of them that is managed by the Department of Defense is referred to as the "DDN" (Defense Data Network). This includes some research-oriented networks, such as the Arpanet, as well as more strictly military ones. All of these networks are connected to each other.

TCP/IP is a set of protocols developed to allow cooperating computers to share resources across a network. It was developed by a community of researchers centered on the ARPAnet. Certainly the ARPAnet is the best-known TCP/IP network.

Thus the most important TCP/IP services are:

File Transfer

Remote login

Email

Network File Systems

Remote Printing

Remote Execution

Name Servers

Terminal Servers

The World Wide Web allows computer users to locate and view multimedia based documents on almost any subject. The use of the Web exploded with the availability in 1993 of the Mosaic browser, which featured a user-friendly graphical interface. Marc Andreessen, whose team at the National Center for Supercomputing Applications developed Mosaic, went on to found Netscape, the company that many people credit with initiating the explosive Internet economy of the late 1990s.

In October 1994, Tim Berners-Lee founded an organization - called World Wide Web Consortium (W3C) devoted to developing nonproprietary, interoperable technologies for the World Wide Web. One of the W3C's primary goals is to make the Web universally accessible – regardless of ability, language or culture.

The W3C is also a standardization organization. Web technologies standardized by the W3C are called Recommendations.

Recently, the hardware has been moving more and more towards mobile, wireless technology. Small hand-held devices are now more powerful than the super computers of the early 1970s. Portability has become a major focus for the computer industry. Wireless data transfer speeds have become so fast that many Internet users' primary access to the Web is through local wireless networks.

Objects are essentially reusable software components that model real-world items. With object technology, properly designed software tends to be more reusable in future projects. Libraries of reusable components, such as Microsoft Foundation Classes(MFC), Sun Microsystems's Java Foundation Classes, Microsoft's .NET Framework Class Library (FCL) and those produced by other software development organization can greatly reduce the effort it takes to implement certain kinds of systems.

**JavaScript** is an object-based scripting language with strong support for proper software engineering techniques. JavaScript was created by Netscape. Microsoft's version of JavaScript calles Jscript. Both Netscape and Microsoft

have been instrumental in the standardization of JavaScript and Jscript by ECMA.

Browsers are available in many version and on many different platforms. It is difficult to develop Web pages that render correctly on all version of each browser.

---

### **1.10. Lesson end Activities**

---

1. What is Internet?
2. What is the W3C?

---

### **1.11. Check your progress**

---

1. Describe Object based scripting
2. What is the software and hardware trends in Internet?

---

### **1.12.Reference**

---

1. Internet & World Wide Web – How to Programe , H.M. Deitel, P.J.Deitel and A.B.Goldberg
2. [www.netforbeginners.com](http://www.netforbeginners.com)
3. [www.w3.org](http://www.w3.org)

## Introduction to HTML

---

### Contents

#### 2.0. Aim and Objective

#### 2.1. Introduction

#### 2.2. Common Tags

#### 2.3. Formatting TAGS

#### 2.4. Special Characters

#### 2.5. Horizontal Rule and Line Break

#### 2.6. Inserting Images

#### 2.7. Let us Sum UP

#### 2.8. Lesson end Activities

#### 2.9. Check Your progress

#### 2.10. Reference

---

#### 2.0. Aim and Objective

---

- To Understand Markup language.
- To learn Basic syntax for writing a web page.

---

#### 2.1. Introduction

---

**Hypertext Markup language (HTML)** is a system for marking up documents with informational tags that indicates how text in the documents should be presented and how the documents are linked.

HTML is defined as Standard Generalised Markup Language(SGML), Document Type Definition(DTD).

Hypertext is text that is not constrained to be linear. Hypertext organizes information as an interconnected web of linked text. Hypermedia applications encompass graphics, video, sound and more. On an HTML hypertext page, the highlighted text that serves as the start of a link is called an anchor.

HTML stands for Hypertext Markup Language. HTML is not compiled but interpreted line by line by the Web Browser. HTML uses tags which direct the Web Browser to display the contents enclosed in the tag in the directed way.

There are only a few general syntax rules to learn in constructing Web documents in HTML. Every HTML document or page is divided into two parts, a head and a body. The head contains information about the document and the body contains the text of the document.

Every markup tag has a tag ID and possibly some attributes. Markup tags are either empty or nonempty. Nonempty tags act upon text enclosed in a pair of starting and ending tags. A starting tag begins with the left angle bracket (<) followed immediately by the tag ID, any attributes, then the right angle bracket (>) to close the tag. Ending tags are exactly the same except that there is a (/) slash immediately between the opening bracket and the tag ID and ending tags cannot contain any attributes. If the tag is an empty tag, then there is no enclosed text and the ending tag is omitted.

Tags are special codes that wrap around various content to affect the content. Tags usually go in pairs.

```
<tagname> some text </tagname>
```

Tags represents the essence of HTML; whenever you want to make your text bold, insert an image or table, add music to your page, you use tags. HTML is essentially a bunch of tags with even more text. Once you learn the syntax of these tags, you can call yourself a HTML expert!

## **Editing HTML**

Editing HTML code we need a text editor. We can use any text editor for example notepad in Windows. We should save the code as .html extension. In windows, it is .htm extension. To run the code or view the output, we have to just click the programe. If any of the browser already loaded in your system it will automatically execute the programe or you can run with the help of open in the browser open menu or you just type the file name with path in the URL window of the browser.



## HTML Page Structure

```
<HTML>

<HEAD>

    <TITLE> Title of the Web Page </TITLE>

</HEAD>

<BODY>

    .

    .

    .

</BODY>

</HTML>
```

TITLE is placed in the HEAD section. Otherwise, there is no distinction between the HEAD and the BODY parts of an HTML page. However, some older browsers do not display colors correctly if the HEAD section contains any HTML tags other than the <TITLE> tag.

---

### 2.2. Common Tags - Creating your first web page

---

Now that you have a vague idea of what tags are, you're ready to learn about the basic tags that make up a basic web page.

The below lists the complete syntax used in creating a very basic web

<b>&lt;html&gt;</b> <b>&lt;/html&gt;</b>	Specifies that this is an HTML document. All html documents begin and end with this tag.
<b>&lt;head&gt;</b> <b>&lt;/head&gt;</b>	Creates a container for meta information, such as the document's title, keywords and description info for search engine crawling, etc to be added to the document.
<b>&lt;title&gt;</b> <b>&lt;/title&gt;</b>	Creates a "title" for the document. Anything you add inside this tag, the browser displays it in the title bar.
<b>&lt;body&gt;</b> <b>&lt;/body&gt;</b>	Creates a container for the document's content (text, images etc). This is where all the "viewable" content will be inserted.

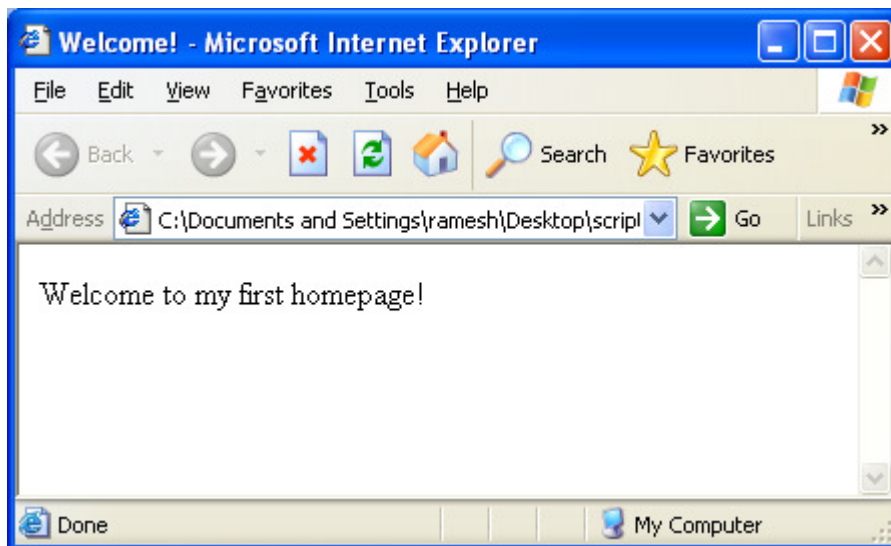
page, with only the text “ Welcome to my first home page” on it.

Example

```
<html>
<head>
<title>Welcome!</title>
</head>
<body>
    Welcome to my first homepage!
</body>
</html>
```

Open up your text editor, type the above, and execute with the help of your browser. You'll see a blank page with the title "Welcome!" on the title bar, and the simple line of text "Welcome to my first homepage" sitting in the main browser area.

Output :



Take another look at the definition of the <body> tag- most of the action in html will take place inside it, since the <body> tag contains all of the document's viewable content.

## **Comment**

Comment is a special text which is not visible in the output. It is used for the programmer's reference.

Comments in HTML are enclosed in `<!-- -->`

Comments can be embedded anywhere in an HTML document except preformatted text.

Example:

```
<!-- html3.htm file -->
```

---

## **2.3. Formatting TAGS**

---

HTML formatting tags can be divided into two loose classes – those that provide structure to the text of a page and those that change the style of the text. The structure class contains heading, paragraphs and lists. The style class contains designing font style.

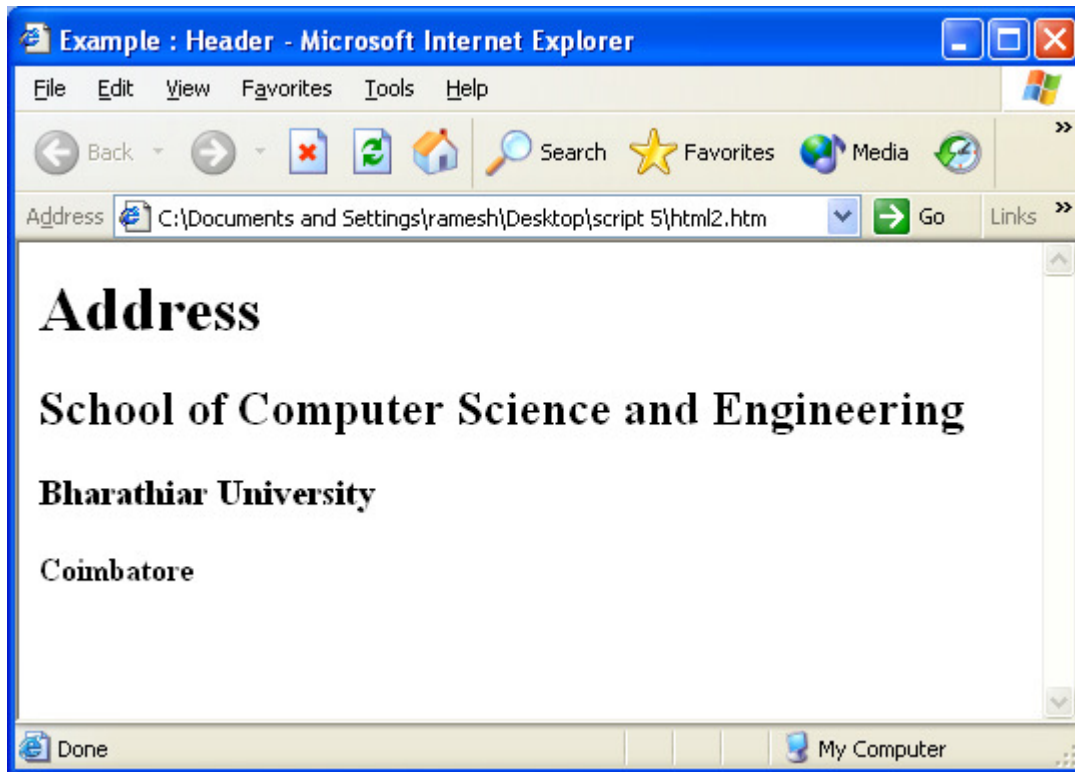
### **Headers**

A header is an extra large, bold text used as, well, headers in a document. HTML support six levels of headings, designated by the tags `<H1>`, `<H2>`, `<H3>`, `<H4>`, `<H5>` and `<H6>`. Heading should be used in their natural hierarchical order, as in an outline. H1 is the highest level of heading, and it is customary to place a level 1 heading as the first element in the body of the document.

Example.

```
<HEAD>
<TITLE> Example : Header </TITLE>
</HEAD>
<BODY>
<H1> Address </H1>
<H2> School of Computer Science and Engineering <H2>
<H3> Bharathiar University <H3>
<H4> Coimbatore <H4>
</BODY>
```

Output :



## Paragraphs

A paragraph can be created in HTML by using the `<p>` tag. The `<p>` tag creates a block of text that is separated by a blank line both above and below the block. A blank line is placed after the paragraph. HTML is self formatting. We need a `<BR>` to force the display to another line (or we can use the `<PRE>` to ask the browser not to format the text). The paragraph tag is empty – that is there is no corresponding end tag.

**Example:** Try removing the `<PRE>` and `</PRE>` tags and viewing the following file again. Experiment by changing the browser window size.

```
<HTML>
```

```
<HEAD>
```

```
<!--This example shows the use of Paragraph Tags.-->
```

```
<TITLE>Examples of Preformatting Tag</TITLE>
```

```
</HEAD>
```

```
<BODY> <PRE>
```

I enjoy developing Web Applications Using ASP.

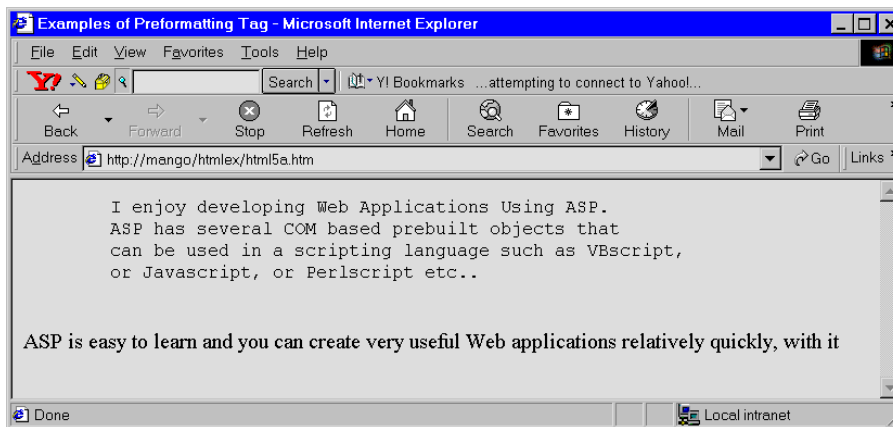
ASP has several COM based prebuilt objects that can be used in a scripting language such as VBscript, or Javascript, or Perlscript etc..

</PRE>

<P> ASP is easy to learn and you can create very useful Web applications relatively quickly, with it

</BODY>

</HTML>



<PRE> Preformatting is useful in displaying tabular data

You can go on to manipulate the alignment of any paragraph by using the align attribute. This attribute accepts three values-left, center, or right. Lets align a paragraph to the right edge of the page:

<p align="right">This is the rightly aligned paragraph. </p>

### **Bold and italic text**

Bold and italic text can be created by using the <b> and <i> tag, respectively:

<b>This text is bold</b>

<i>This text is italic</i>

### **Centering text**

A <center> tag exists that can be used to wrap around virtually around formatting tag to center it. Here are a couple of examples:

**<center><b>This bold text is centered</b></center>**

**<center><h3>This header is centered as well!</h3></center>**

**Example:**

```
<HTML>
```

```
<HEAD>
```

```
<!--This example shows how the PRE tag can be used to enter tabular
data. -->
```

```
<TITLE>Pre-formatted text example</TITLE>
```

```
<H1><CENTER>Regional Sales in Each Quarter</CENTER></H1>
```

```
<HR>
```

```
</HEAD>
```

```
<BODY>
```

```
<H2>1996 Sales by Region<H2>
```

```
<H3>(in thousands $)</H3>
```

```
<BR>
```

```
<PRE>
```

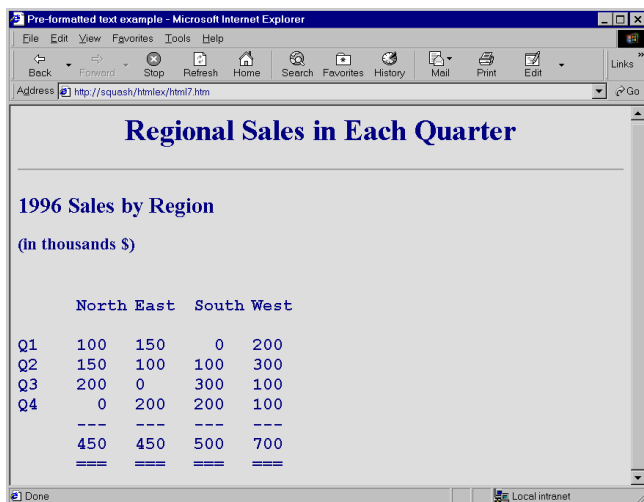
```

      North East  South West
Q1   100   150    0   200
Q2   150   100   100  300
Q3   200    0   300   100
Q4    0   200   200   100
    ---   ---   ---   ---
    450   450   500   700
    ===   ===   ===   ===
```

```
</PRE>
```

```
</BODY>
```

```
</HTML>
```



## Formatting your text

Learning how to format text ranks as one of the most important things to learn in HTML.

## Setting Background, Foreground colors

Background/Foreground Colors are specified in the BODY Tag. Similarly a Background Image is specified in the BODY Tag.

## Changing font color, size, and type

Like any decent word processor, you can also alter the font color, size, and type of the text. The three tags that accomplish this are as follows:

```
<font color="#FF0000">This text is red</font>
```

```
<font size="6">This text is very big!</font>
```

```
<font face="Courier">This text is in courier</font>
```

The valid values for the font color are the hex values of colors - the same values used for background colors. For font size, an integer between 1 and 7 should be used, with 7 representing the largest font. For the font face, use the name of the font type as the value, such as Courier, Arial, etc.

You can easily show different formatting tags into one big code to create the effect desired. For example, if you want text that is bold, 2 in font size, italic, and Arial in font type, do the below:

```
<b><i><font size="5" face="Arial">Complex Text</font></i></b>
```

As you can see, HTML is very flexible, and allows you to throw together various tags to create the desired effect when one by itself cannot do the job.

### Text Formatting Tags

`<BIG>` This text will appear one size larger than the surrounding text `</BIG>`

`<SMALL>` > This text will appear one size larger than the surrounding text `</SMALL>`

`<SUB>` To provide subscripts `</SUB>`

`<SUP>` To provide superscripts `</SUP>`

`<STRIKE>` strike through text `</STRIKE>`

`<FONT size=4 FACE="Verdana, Arial, sans-serif">` some text in a different font `</FONT>`

`<OL TYPE=a>`

`<LI>` First Item

`<LI>` Second Item

`<LI>` Third Item

a. First Item

b. Second Item

c. Third Item

`</OL>`

`<OL TYPE=I>`

`<LI>` First Item

`<LI>` Second Item

`<LI>` Third Item

I. First Item

II. Second Item

III. Third Item

`</OL>`

`<OL TYPE=i>`

`<LI>` First Item

`<LI>` Second Item

`<LI>` Third Item

i. First Item

ii. Second Item

iii. Third Item

`</OL>`

`<OL Start=14>`

`<LI>` First Item

`<LI>` Second Item

`<LI>` Third Item

14. First Item

15. Second Item

16. Third Item

`</OL>`

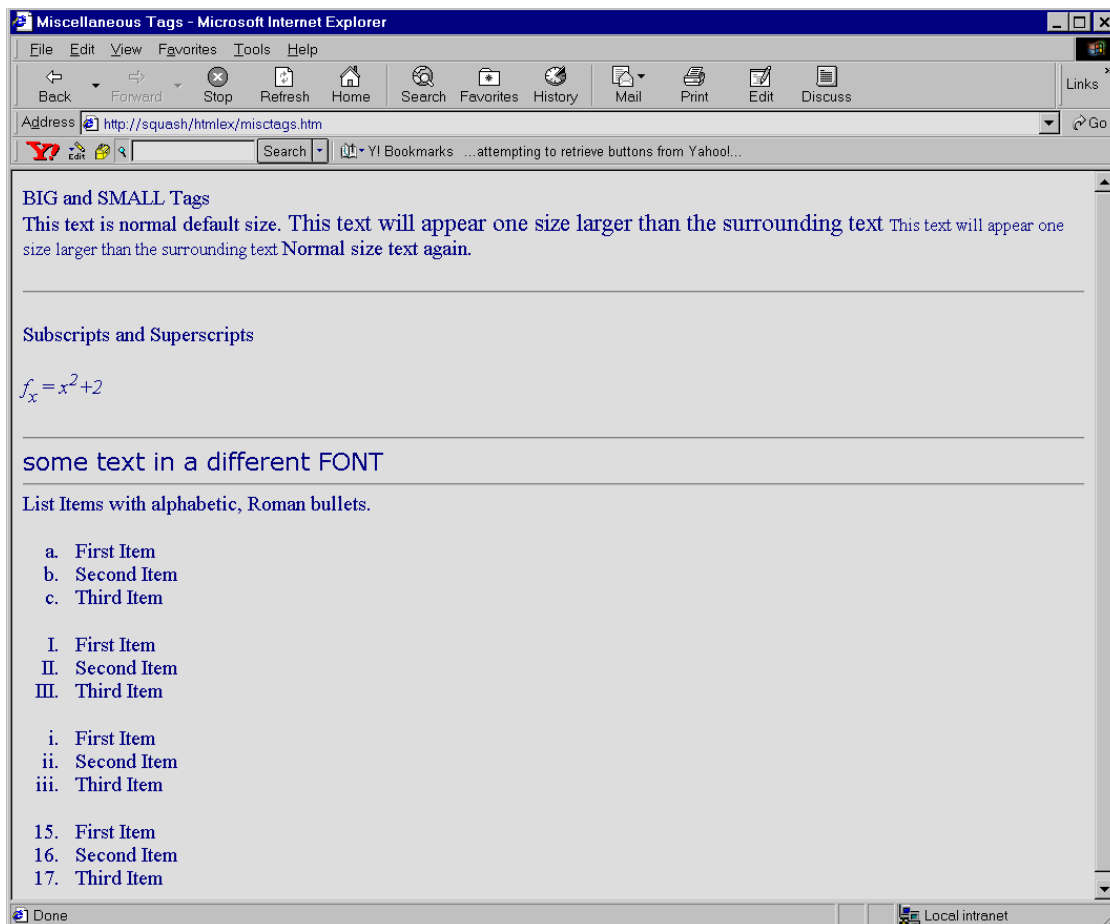


**Example:**

```
<!-- misctags.htm -->
<HTML>
  <HEAD>
    <!--This example shows use of a few useful text formatting tags. -->
    <TITLE> Miscellaneous Tags</TITLE>
  </HEAD>
  <BODY>
    BIG and SMALL Tags <BR>
    This text is normal default size.
    <BIG> This text will appear one size larger than the
      surrounding text </BIG>
    <SMALL> This text will appear one size larger than the
      surrounding text </SMALL>
    Normal size text again. <BR>
    <BR>
    <HR>
    <BR>
    Subscripts and Superscripts <BR>
    <BR>
    <I>f<sub>x</sub></I> = <I>x<sup>2</sup>+2 </I><BR>
    <BR>
    <HR>
    <FONT size=4 FACE="Verdana, Arial, sans-serif">
      some text in a different FONT </FONT>
    <HR>
    List Items with alphabetic, Roman bullets. <BR>
    <OL TYPE=a>
      <LI> First Item
```

```
<LI> Second Item
<LI> Third Item
</OL>
<OL TYPE=I>
  <LI> First Item
  <LI> Second Item
  <LI> Third Item
</OL>
<OL TYPE=i>
  <LI> First Item
  <LI> Second Item
  <LI> Third Item
</OL>
<OL start=15>
  <LI> First Item
  <LI> Second Item
  <LI> Third Item
</OL>

</BODY>
</HTML>
```



## Indenting Text:

Since HTML is autoformatting, indenting requires either using a preformatting specification by the use of `<PRE>` tag, or using a `<BLOCKQUOTE>` tag.

Example :

`<BLOCKQUOTE>`

Learning ASP requires some programming background, knowledge of HTML and a knowledge of scripting language such as VB script. ASP Pages can be developed using simple text editor such as Notepad. However, as your skill level increases, a more efficient development environment such as Visual Interdev is highly recommended.

`</BLOCKQUOTE>`

The above text inside the BLOCKQUOTES will be indented 40 pixels to the right. It is also possible to use the <UL> tag to indent the text without specifying any list item.

**Example:**

```
<!-- misctags2.htm -->
<HTML>
  <HEAD>
    <!--This example shows use of a few useful text formatting tags. -->
    <TITLE> Miscellaneous Tags</TITLE>
  </HEAD>
  <BODY>
    <DIV ALIGN="center">
      <H2> Aligning Text with the DIV and ALIGN Tags </H2>
      <P>    Learning ASP requires some programming background,
knowledge
      of HTML and a knowledge of scripting language such as VB script.
    </P>
    <P> ASP Pages can be developed using simple text editor such as
Notepad.

    However, as your skill level increases, a more efficient development
    environment such as Visual Interdev is highly recommended. </P>
  </DIV>
  <HR>
  <H2> Indenting By the use of BLOCKQUOTE tag </H2>
  <BLOCKQUOTE>
    Learning ASP requires some programming background, knowledge of
HTML
    and a knowledge of scripting language such as VB script. ASP Pages
can
```

be developed using simple text editor such as Notepad. However, as your

skill level increases, a more efficient development environment such as Visual Interdev is highly recommended.

</BLOCKQUOTE>

<HR>

<H2> Indenting By use of UL Tag </H2>

<UL>

Learning ASP requires some programming background, knowledge of HTML

and a knowledge of scripting language such as VB script. ASP Pages can

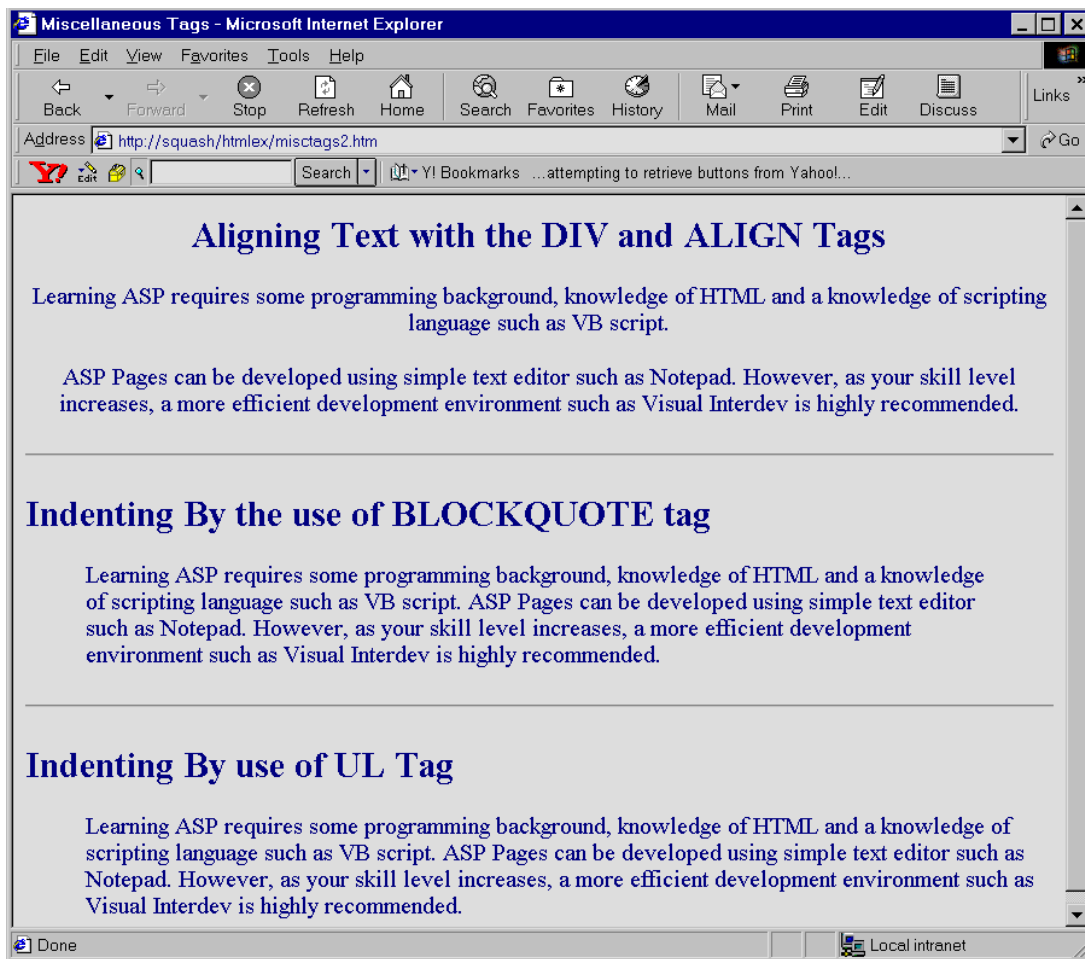
be developed using simple text editor such as Notepad. However, as your

skill level increases, a more efficient development environment such as Visual Interdev is highly recommended.

</UL>

</BODY>

</HTML>



## Adding a background to your document

The first thing most beginner webmasters want to do with their web page is to add a background to it, whether it be a background color or image. By default, a document with no background gray (or white in newer browsers) in the background. That's easy to appears fix. Lets begin by adding a background color. To add a splash of color to your document, add the following code inside the <body> tag itself:

```
<body bgcolor="#XXXXXX">
```

where xxxxxx is the hex code for the color you want. Here's a small chart of the hex code for some common colors:

<b>Black</b>	<b>#000000</b>
<b>White</b>	<b>#FFFFFF</b>
<b>Blue</b>	<b>#0000FF</b>
<b>Yellow</b>	<b>#FFFF00</b>
<b>Red</b>	<b>#FF0000</b>
<b>Green</b>	<b>#008000</b>
<b>Lime</b>	<b>#00FF00</b>
<b>Silver</b>	<b>#C0C0C0</b>

For example, the below gives our document a background of black:

```
<body bgcolor="#000000">
```

**Example: Background Color.**

```
<!-- html11.htm>

<HTML>

<HEAD>

<!--This example shows the use of Background Colors.-->

<TITLE>Examples of Background Color</TITLE>

<H1>Welcome to my Web page.</H1>

<HR>

</HEAD>

<BODY BGCOLOR="#00FF00" TEXT=#FF0000">

<P> I enjoy developing Web Applications Using ASP.<BR>
ASP has several COM based prebuilt objects that <BR>
can be used in a scripting language such as VBscript,<BR>
or Javascript, or Perlscript etc..

</P>

<P> ASP is easy to learn and you can create very useful Web <BR>
```

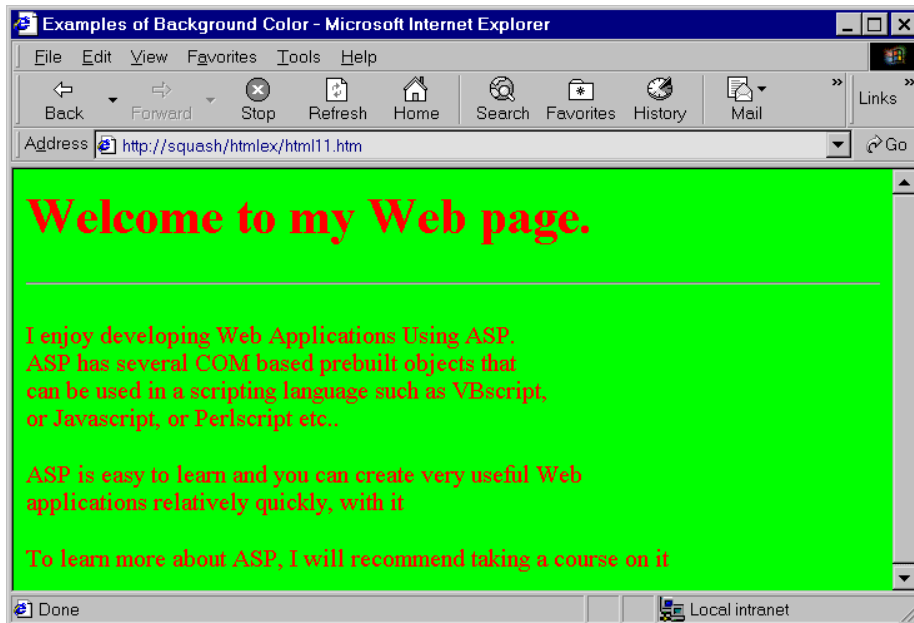
applications relatively quickly, with it <BR>

</P>

To learn more about ASP, I will recommend taking a course on it

</BODY>

</HTML>



#### Example: background Image.

```
<!-- html12.htm>
```

```
<HTML>
```

```
<HEAD>
```

```
<!--This example shows the use of Background Image.-->
```

```
<TITLE>Examples of Background Image</TITLE>
```

```
<H1>Welcome to my Web page.</H1>
```

```
<HR>
```

```
</HEAD>
```

```
<BODY BACKGROUND="c:/htmlex/backgrnd.gif">
```

```
<P> I enjoy developing Web Applications Using ASP.<BR>
```

```
ASP has several COM based prebuilt objects that <BR>
```

```
can be used in a scripting language such as VBscript,<BR>
```



or Javascript, or Perlscript etc..

</P>

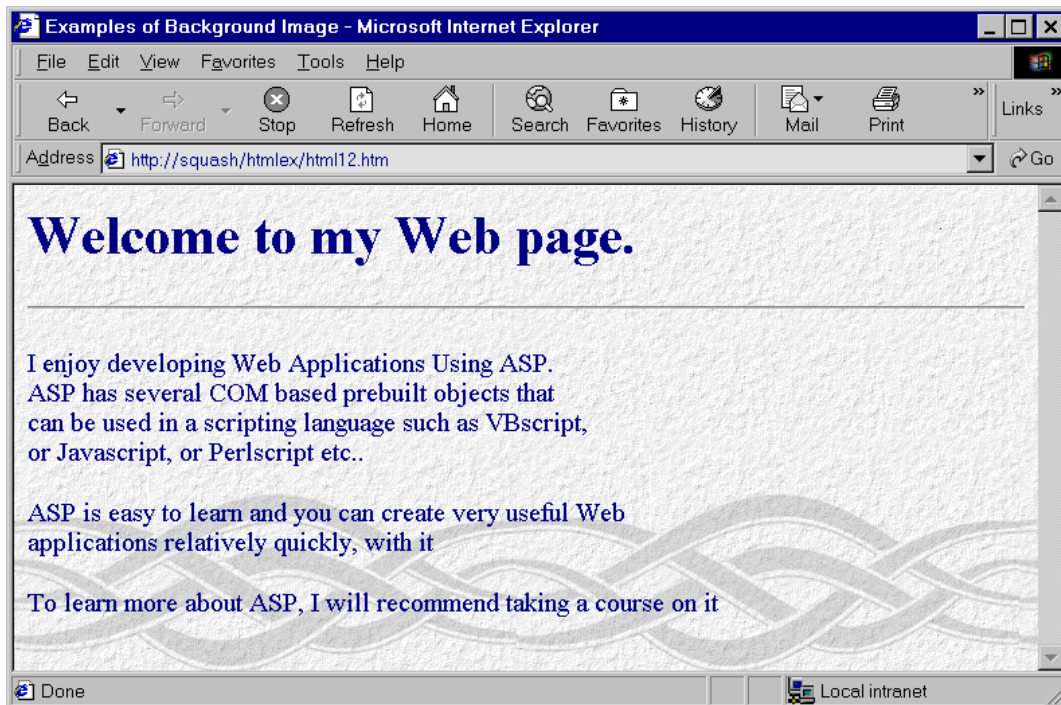
<P> ASP is easy to learn and you can create very useful Web <BR>  
applications relatively quickly, with it <BR>

</P>

To learn more about ASP, I will recommend taking a course on it

</BODY>

</HTML>



**You can also choose Link colors in the BODY Tag.**

VLINK="#rrggbb" => Visited Link Color (default is purple)

ALINK="#rrggbb" => Active Link Color (default is red)

LINK="#rrggbb" => Link (default is Blue)

**Example:**

<!-- html13.htm -->

<HTML>

<HEAD>

<TITLE>Color of Active and Visited Links</TITLE>

```

<H1>This page contains some popular Web links.</H1>

<HR>

</HEAD>

<BODY TEXT="green" VLINK="red"
      LINK="yellow" ALINK="#00FFA0">

<UL>

  <LI><A HREF="http://www.amazon.com">Amazon Shopping</A>

  <LI><A      HREF="http://msdn.microsoft.com/vbasic/">Visual
Basic</A>

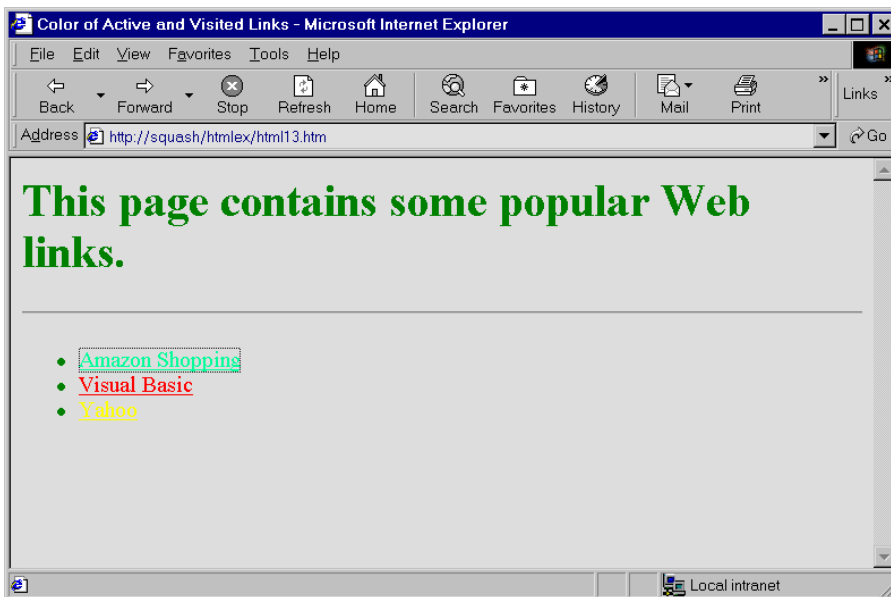
  <LI><A HREF="http://www.yahoo.com">Yahoo</A>

</UL>

</BODY>

</HTML>

```



Now that you know how to give your doc a background color, lets move on to learn how to give it an image as well. For illustration, lets first bring in a nice image we'll be using:

To utilize the above image as the background, use the following syntax:

```
<body background="backgr15.jpg">
```

Many authors like to give their document BOTH a background color, and an image as well. This way, while the image has yet to come through from the server, surfers will see a background color in the meantime:

```
<body bgcolor="#000000" background="backgr15.jpg">
```



backgr15.jpg

## Background Sounds

Use the BGSOUND Tag to embed a sound file. Use LOOP="infinite" to repeatedly play the sound file.

### Example:

```
<!-- html14.htm>
<HTML>
  <HEAD>
    <!--This example shows the use of Background Sound.-->
    <TITLE>Example of Background Sound</TITLE>
    <H1>Welcome to My Musical Page.</H1>
    <HR>
  </HEAD>
  <BODY BGCOLOR="aqua"> <BGSOUND
SRC="d:/htmlex/welcom98.wav" LOOP="infinite">
  <PRE>
```

I enjoy developing Web Applications Using ASP.

ASP has several COM based prebuilt objects that can be used in a scripting language such as VBscript, or Javascript, or Perlscript etc..

ASP is easy to learn and you can create very useful Web applications relatively quickly, with it

To learn more about ASP, I will recommend taking a course on it

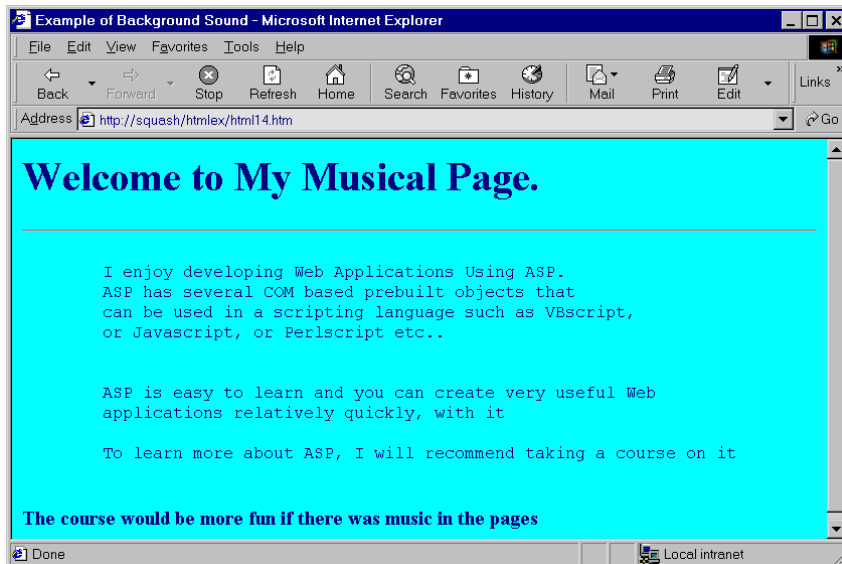
```
</PRE>
```

<STRONG>The course would be more fun if there was music in  
the pages

</STRONG>

</BODY>

</HTML>



---

## 2.4. Special Characters

---

There are two kinds of HTML elements; character entities and markup tags. A characters entity begins with an ampersand (&), followed either by name of a predefined entity or a pound sign, followed by the decimal number of the character, and finally, by a semicolon to terminate by the decimal number of the character, and, finally, by a semicolon to terminate the character entity. For example the tilde(~) can be generated by the sequence &#126;.

Many character entities are predefined for the purpose of placing special characters into the text. Some of them are :

&lt;	Right angle bracket or less-than sign
&gt;	Left angle bracket or greater-than sign
&amp;	Ampersand
&quot;	Double quote mark
&nbsp;	Nonbraking space

---

## 2.5. Horizontal Rules and Line Break

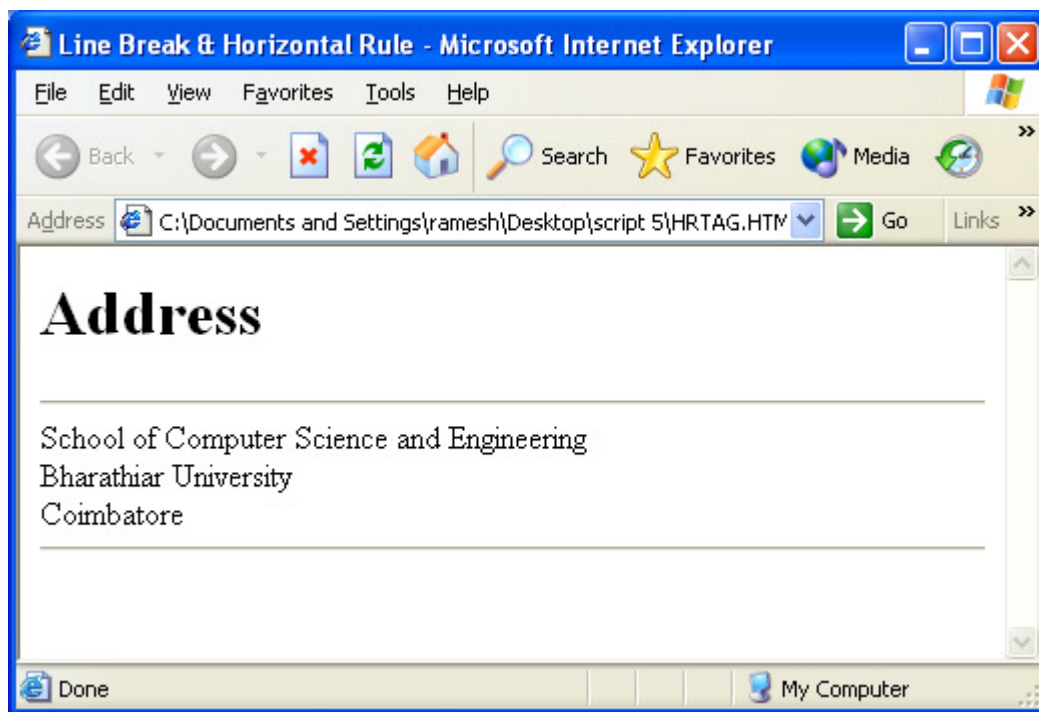
---

The line break tag <BR> is similar to the paragraph tag. It forces the text directly following the tag onto the next line at the left margin without the addition of any white space.

Another way of separating blocks of text is with a line drawn across the width of the page. This is called a horizontal rule, and the tag for it is <HR>. The browser will adjust the length of the line to fit the width of the displayed page.

Eg.

```
<HEAD>
  <TITLE> Line Break & Horizontal Rule </TITLE>
</HEAD>
<BODY>
  <H1> Address </H1>
  <HR>
  School of Computer Science and Engineering <BR>
  Bharathiar University <BR>
  Coimbatore <BR>
  <HR>
</BODY>
```



---

## 2.6. Inserting Images

---

Enough with boring text, lets move on to something more colorful- images!

Images are inserted into a document by using the `<img>` tag. The below inserts an image called "paperboy.gif":



```

```

Two things to notice here. First, all `<img>` tags have a `src` attribute, which is required to specify the file path of the image you're inserting (in this case, its `paperboy.gif`). Second, `<img>` tags do not have closing tags (ie `</img>`). It's one of those rare cases where a closing tag is not required.

### **The width and height attribute of `<img>`**

There's a secret to making your images load faster in a document- use the width and height attribute of the `<img>` tag. These attributes allow us to explicitly specify the dimensions of the image, thus saving the browser from

ownload time. The above paper boy is 98\*100 in dimensions. Lets tell our browser that when defining it, shall we?

```

```

The width/height attribute can actually do more than just speed up an image's download. We can use it to also alter the dimensions of the image. Lets blow up the paperboy by giving it a large width and height:

```

```



Ugly paper boy, from this viewpoint!

---

## 2.7. Let us Sum up

---

**Hypertext Markup language (HTML)** is a system for marking up documents with informational tags that indicates how text in the documents should be presented and how the documents are linked.

Hypertext is text that is not constrained to be linear. Hypertext organizes information as an interconnected web of linked text. Hypermedia applications encompass graphics, video, sound and more.

HTML stands for Hypertext Markup Language. HTML is not compiled but interpreted line by line by the Web Browser. HTML uses tags which direct the Web Browser to display the contents enclosed in the tag in the directed way.

A starting tag begins with the left angle bracket (<) followed immediately by the tag ID, any attributes, then the right angle bracket (>) to close the tag. Ending tags are exactly the same except that there is a (/) slash immediately between the opening bracket and the tag ID and ending tags cannot contain

any attributes. If the tag is an empty tag, then there is no enclosed text and the ending tag is omitted.

## Editing HTML

Editing HTML code we need a text editor. We can use any text editor for example notepad in Windows. We should save the code as .html extension. In windows, it is .htm extension. To run the code or view the output, we have to just click the programe. If any of the browser already loaded in your system it will automatically execute the programe or you can run with the help of open in the browser open menu or you just type the file name with path in the URL window of the browser.

## Common Tags

<b>&lt;html&gt;</b> <b>&lt;/html&gt;</b>	Specifies that this is an HTML document. All html documents begin and end with this tag.
<b>&lt;head&gt;</b> <b>&lt;/head&gt;</b>	Creates a container for meta information, such as the document's title, keywords and description info for search engine crawling, etc to be added to the document.
<b>&lt;title&gt;</b> <b>&lt;/title&gt;</b>	Creates a "title" for the document. Anything you add inside this tag, the browser displays it in the title bar.
<b>&lt;body&gt;</b> <b>&lt;/body&gt;</b>	Creates a container for the document's content (text, images etc). This is where all the "viewable" content will be inserted.

## Comment

Comment is a special text which is not visible in the output. It is used for the programmer's reference.

Comments in HTML are enclosed in `<!-- -->`

## Formatting TAGS

### Headers

A header is an extra large, bold text used as, well, headers in a document. HTML support six levels of headings, designated by the tags `<H1>`, `<H2>`, `<H3>`, `<H4>`, `<H5>` and `<H6>`.



## Bold and italic text

Bold and italic text can be created by using the `<b>` and `<i>` tag, respectively:

## Centering text

A `<center>` tag exists that can be used to wrap around virtually around formatting tag to center it.

## Formatting your text

### Changing font color, size, and type

You can also alter the font color, size, and type of the text. The three tags that accomplish this are as follows:

```
<font color="#FF0000">This text is red</font>
```

```
<font size="6">This text is very big!</font>
```

```
<font face="Courier">This text is in courier</font>
```

## Text Formatting Tags

```
<BIG> This text will appear one size larger than the surrounding text </BIG>
```

```
<SMALL> > This text will appear one size larger than the surrounding text  
</SMALL>
```

```
<SUB> To provide subscripts </SUB>
```

```
<SUP> To provide superscripts </SUP>
```

```
<STRIKE> strike through text </STRIKE>
```

```
<FONT size=4 FACE="Verdana, Arial, sans-serif"> some text in a different font  
</FONT>
```

```
<OL TYPE=a>
```

```
<LI> First Item
```

```
<LI> Second Item
```

```
<LI> Third Item
```

```
</OL>
```

```
<body bgcolor="#XXXXXX">
```

a. First Item
---------------

b. Second Item
----------------

c. Third Item
---------------

where xxxxxx is the hex code for the color you want. Here's a small chart of the hex code for some common colors:

<b>Black</b>	<b>#000000</b>
<b>White</b>	<b>#FFFFFF</b>
<b>Blue</b>	<b>#0000FF</b>
<b>Yellow</b>	<b>#FFFF00</b>
<b>Red</b>	<b>#FF0000</b>
<b>Green</b>	<b>#008000</b>
<b>Lime</b>	<b>#00FF00</b>
<b>Silver</b>	<b>#C0C0C0</b>

Now that you know how to give your doc a background color, lets move on to learn how to give it an image as well. For illustration, lets first bring in a nice image we'll be using:

To utilize the above image as the background, use the following syntax:

```
<body background="backgr15.jpg">
```

```
<body bgcolor="#000000" background="backgr15.jpg">
```



## Special Characters

There are two kinds of HTML elements; character entities and markup tags. A characters entity begins with an ampersand (&), followed either by name of a predefined entity or a pound sign, followed by the decimal number of the character, and finally, by a semicolon to terminate by the decimal number of the character, and, finally, by a semicolon to terminate the character entity. For example the tilde(~) can be generated by the sequence `&#126;`.

Many character entities are predefined for the purpose of placing special characters into the text. Some of them are :

&lt;	Right angle bracket or less-than sign
&gt;	Left angle bracket or greater-than sign
&amp;	Ampersand
&quot;	Double quote mark
&nbsp;	Nonbraking space

## Horizontal Rules and Line Break

The line break tag <BR> is similar to the paragraph tag. Another way of separating blocks of text is with a line drawn across the width of the page. This is called a horizontal rule, and the tag for it is <HR>.

---

### 2.8. Lesson End Activities

---

1. What is Markup language?
2. Who develop the HTML?
3. Write the tag for
  - a. Header
  - b. Italic
  - c. Background color
  - d. Line Break

---

### 2.9. Check Your progress

---

1. Write in detail about HTML.
2. Using HTML design a page for a Marriage Invitation.

---

### 2.10. Reference

---

1. [WWW.W3C](http://WWW.W3C)
2. Thomas A. Powell, "The Complete Reference HTML and XHTML", fourth Edition, Tata McGraw Hill.



---

**Intermediate HTML**

---

**Contents****3.0. Aim and Objective****3.1. Introduction****3.2. List****3.3. Tables****3.4. Forms****3.5. Hyperlinks****3.6. Frames****3.7. <META> tag****3.8. Let us Sum Up****3.9. Lesson End Activities****3.10. Check Your Progress****3.11. Reference**

---

**3.0. Aim and Objective**

---

- To learn advanced HTML
- To write a program using list, table and forms

---

**3.1. Introduction**

---

Your personal home page is the starting point for all your hypertext works. Since it will be created from scratch, use top-down approach. If you are developing your home page, it needs some list of order like bullet, some table format and multiple screen design. This lesson will teach you these concepts.

---

**3.2. Lists**

---

A list is defined as a sequence of paragraphs, each marked with the list item tag <LI>. The entire sequence of list items is enclosed with the starting and

ending tags appropriate to the kind of list. HTML list tags come in **three flavors**:

**Ordered (numbered) lists:** Typically indented with extra line spacing between numbered paragraphs.

```
<OL>
  <LI>First list item</LI>
  <LI>Second item</LI>
  <LI>Third list item</LI>
</OL>
```

**Unordered (bullet) lists:** Like an ordered list but with bullets instead of numbers.

```
<UL>
  <LI>One list item</LI>
  <LI>Another list item</LI>
  <LI>Still another list item</LI>
</UL>
```

**Definition lists:** A list of very short elements, such as file names, possibly rendered in multiple columns.

```
<DL>
  <DT>Term to be defined</DT>
  <DD>Definition of term to be defined...</DD>
  <DT>Another term to be defined</DT>
  <DD>Definition of another term to be defined...</DD>
</DL>
```

It is not necessary to have paired <DT> and <DD> tags. Sometimes definition lists are used just to create different levels of indentation.

### **Example : 3.1**

```
<HTML>
  <HEAD>
    <TITLE>Link and List Tags</TITLE>
```

</HEAD>

<BODY> <H1> This Page contains some popular university web site links </H1>

<HR>

<UL>

<LI><A HREF="http://www.annauniv.ac.in">Anna University</A>

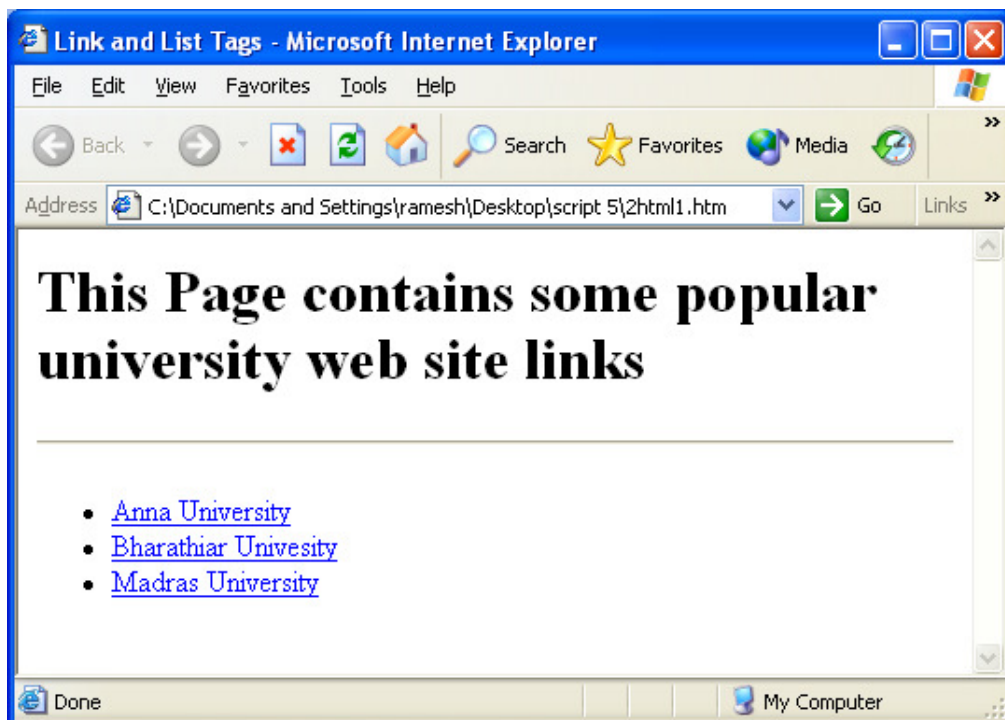
<LI><A HREF="http://www.bharathiaruniv.ac.in ">Bharathiar Univesity</A>

<LI><AHREF="http://www.madrasuniv.ac.in">Madras University</A>

</UL>

</BODY>

</HTML>



### Example : 3.2

<HTML>

<HEAD>

<TITLE>E-mail Links </TITLE>

</HEAD>

```

<BODY>    <H1>This page contains some useful E-mail links.</H1>

<HR>

<UL>

    <LI><A HREF="mailto:vc@bharatthiaruniv.ac.in">Vice Chancellor
</A>

    <LI><A HREF="reg@bharathiaruniv.ac.in">Registrar </A>

</UL>

<PRE>

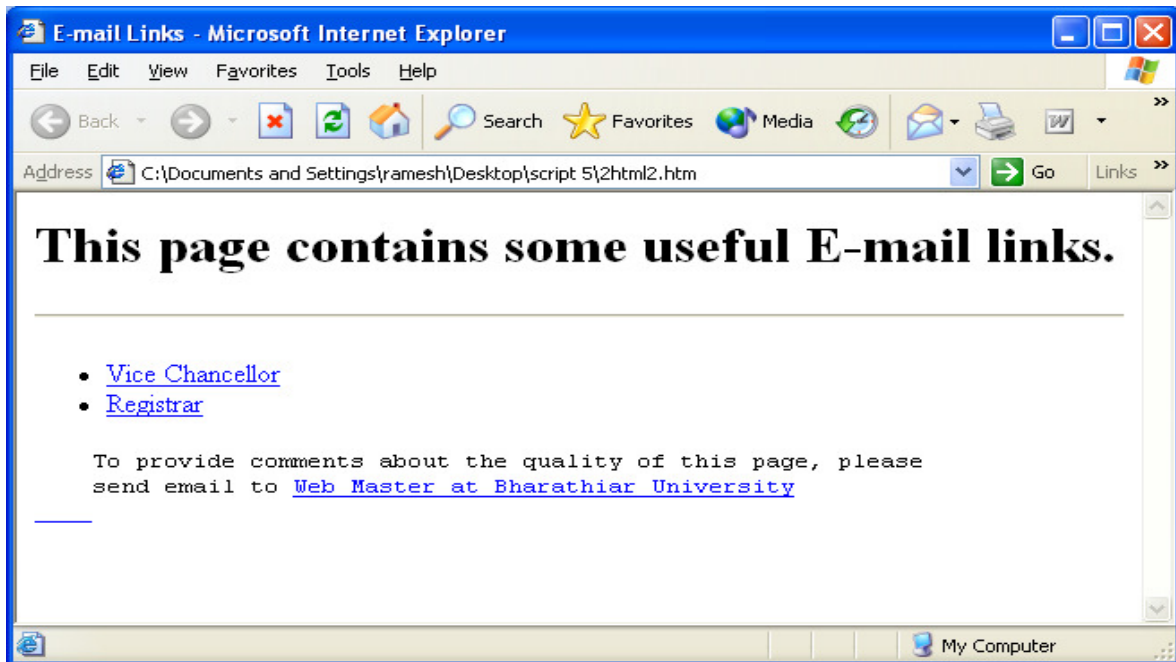
To provide comments about the quality of this page, please
send email to <A HREF="mailto:cse@bharathiaruniv.ac.in">Web
Master at Bharathiar University

</PRE>

</BODY>

</HTML>

```



### 3.3. Tables

Tables are specified using `<TABLE>` `</TABLE>` tags. Caption for table are specified with the `<CAPTION>` `</CAPTION>`. Table caption appear above the table. Tables have header and data cells defined by the empty tags., `<TH>` and

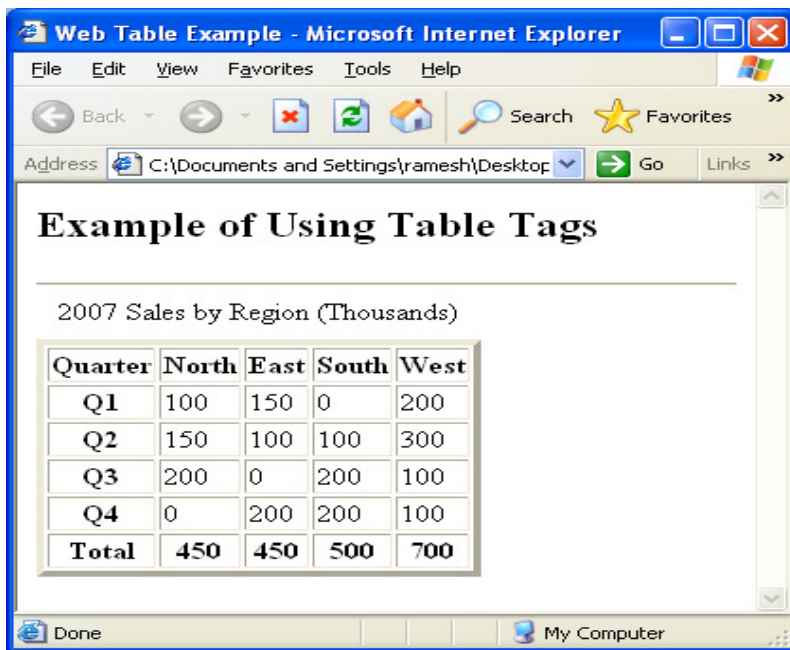


<TD>. Cells may contain text, paragraphs, lists and headers. Each row of a table is ended by table row tag <TR>.

Table Related Tag	Description
<CAPTION>.. ..<CAPTION>	Indicates caption of the table
<TR> .. </TR>	New Row for the table
<TH> .. </TH>	Table Heading (puts data in bold)
<TD> .. </TD>	Table Data

### Example: 3.3

```
<HTML>
<HEAD>
  <TITLE>Web Table Example</TITLE>
  <H2>Example of Using Table Tags</H2>
  <HR>
</HEAD>
<BODY>
  <TABLE BORDER=4>
    <CAPTION>2007 Sales by Region (Thousands)</CAPTION>
    <TR><TH> Quarter <TH> North <TH> East <TH> South <TH> West </TR>
    <TR><TH> Q1      <TD> 100  <TD> 150  <TD> 0    <TD> 200  </TR>
    <TR><TH> Q2      <TD> 150  <TD> 100  <TD> 100  <TD> 300  </TR>
    <TR><TH> Q3      <TD> 200  <TD> 0    <TD> 200  <TD> 100  </TR>
    <TR><TH> Q4      <TD> 0    <TD> 200  <TD> 200  <TD> 100  </TR>
    <TR><TH> Total<TH> 450  <TH> 450  <TH> 500    <TH> 700  </TR>
  </BODY>
</HTML>
```



Use `<TH COLSPAN=number>` or `<TH ROWSPAN=number>` to span multiple columns or rows for a table heading.

### Web Tables can also contain Images

#### Example: 3.5

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>Web Table Example</TITLE>
```

```
<H2>Web Table Example - Images inside a Table</H2>
```

```
<H3>Text and images can span more than one column or row with the  
      use of COLSPAN/ROWSPAN elements.</H3>
```

```
<CENTER>
```

```
<H4>Text and tables can also be centered using the CENTER  
tag!</H4>
```

```
</CENTER>
```

```
<HR>
```

```
</HEAD>
```

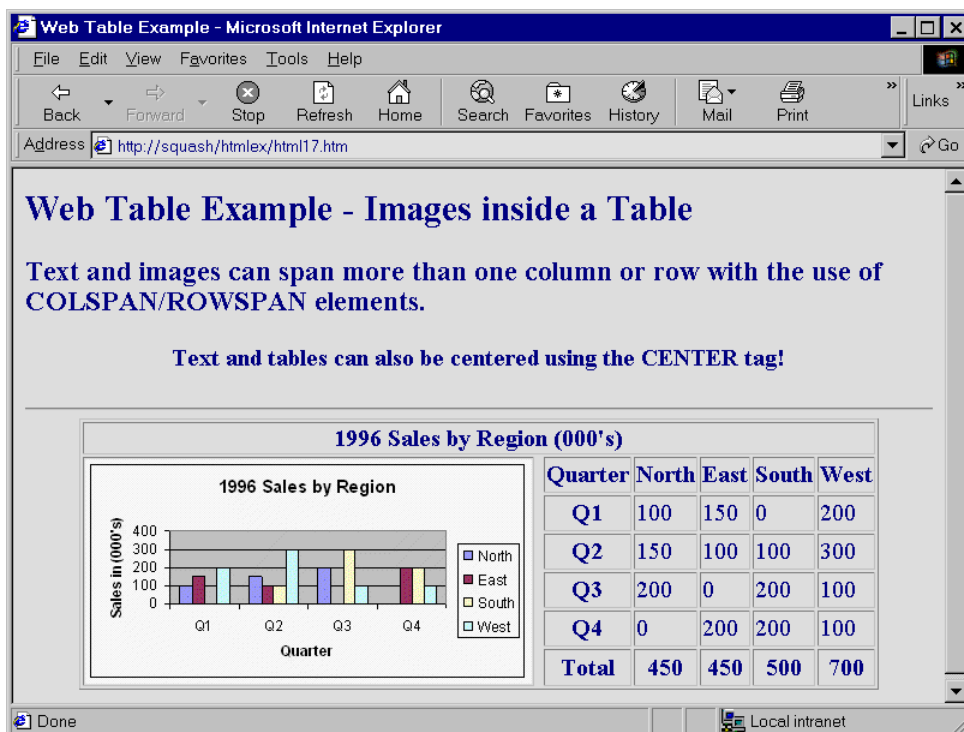
```
<BODY>
```

```
<CENTER>
```

```

<TABLE BORDER>
<TH COLSPAN=7> 1996 Sales by Region (000's)</TH>
<TR><TH ROWSPAN=6><IMG SRC="d:/htmllex/2006.GIF"></TH>
<TH><TH> Quarter <TH> North <TH> East <TH> South <TH> West
<TR><TH><TH> Q1 <TD> 100 <TD> 150 <TD> 0 <TD> 200 </TR>
<TR><TH><TH> Q2 <TD> 150 <TD> 100 <TD> 100 <TD> 300 </TR>
<TR><TH><TH> Q3 <TD> 200 <TD> 0 <TD> 200 <TD> 100 </TR>
<TR><TH><TH> Q4 <TD> 0 <TD> 200 <TD> 200 <TD> 100 </TR>
<TR><TH><TH> Total <TH> 450 <TH> 450 <TH> 500 <TH> 700 </TR>
</CENTER>
</BODY>
</HTML>

```



### 3.4. Forms

A form is a designated area of an HTML page made available for user input. There can be many forms on a page; however, forms cannot be nested. Each form is defined by starting and ending tags with attributes for specifying how the form's input should be processed.

The basic idea of a form is to present input field to the reader for typing in text information, and radio buttons, check boxes, and pop-up menus for selecting items from option lists. Somewhere on the form is a set of action buttons, typically a Reset button and a Submit button.

The Reset Button, when clicked, clears the reader's input form the text fields in the form and sets all input objects back to their default values. The submit button buttons, when clicked, instructs the browser to take the action specified in the form's Action attribute according to the method specified in the METHOD attribute. There are two action methods, METHOD=GET and METHOD=POST.

If the GET method is specified in the form tag, the browser constructs a query URL consisting of the URL of the current page containing the form, followed by a question mark, followed by the values of the form's input fields and objects. The browser sends this to a executable script or program on a server identified by the URL in the ACTION attribute. The script or program can use this information to do any number of things, such as searching and updating databases. The process ends with the server returning a new page to the reader, possibly one dynamically created by the server script. If the POST method is specified, the form's contents are sent to the server script as a data block to standard input.

Layout of Form Elements can be improved by embedding the controls inside a Table.

Example: 3.6

```
<HTML>

<HEAD>

  <TITLE> Controlling Form Layout with a Table </TITLE>

  <H2> Form Elements can be aligned better through a Table </H2>

  <HR>

</HEAD>

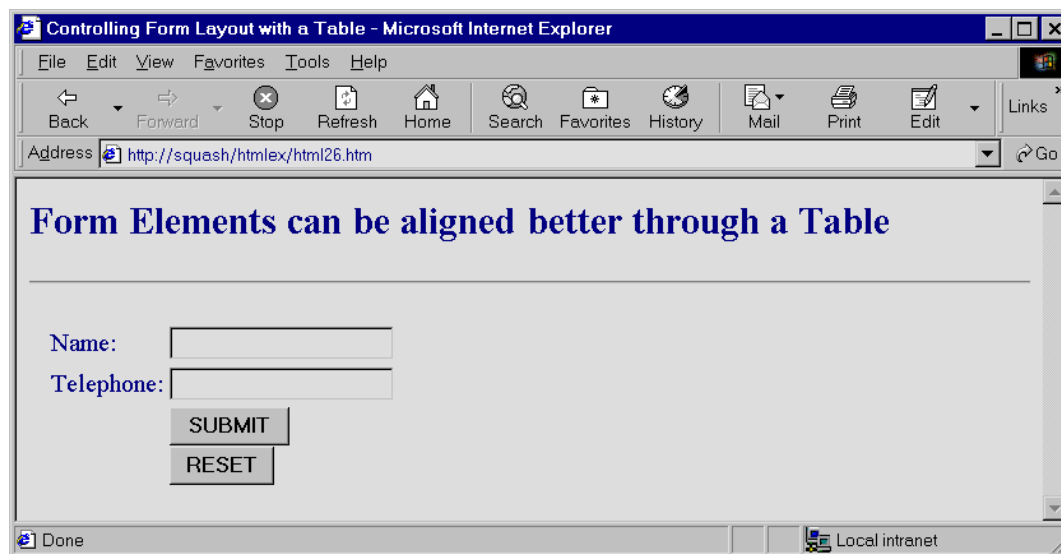
<BODY>

<FORM METHOD=GET ACTION="">
```

```

<TABLE WIDTH=1 BORDER=0>
  <TR>
    <TD>&#160;&#160;<!-- space escape code -->
    <TD>Name:
    <TD><INPUT Length=10 NAME=txtName>
  <TR>
    <TD>
    <TD>Telephone:
    <TD><INPUT NAME=txtPhone>
  <TR>
    <TD>
    <TD>
    <TD><INPUT TYPE=submit value=SUBMIT>
      <INPUT TYPE=reset value=RESET>
    </TD>
  </TR>
</TABLE>
</FORM>
</BODY>
</HTML>

```



---

### 3.5. Hyperlinks

---

Hyperlinks represent the essence of the WWW, linking millions and millions of pages from around the world together. Hyperlinks allow a page to be linked to other Pages. You can also provide email links. Graphic images can also be used to point to a Hyperlink.

#### **Anchors and Links Within a Document**

HTML differs from traditional formatting languages in the ability to include hyperlinks in documents. This is done with <A> anchor tags. Two basic types of anchor tags are needed to define hyperlinks for jumping to different locations within a document:

```
<A NAME="anchorname">Target</A>
```

Defines "Target" as a location in the current document that you can refer to as "#anchorname" in a link.

```
<A HREF="#anchorname" TITLE="description">Reference</A>
```

Defines "Reference" as a location you can click on to jump to the target location "#anchorname". The TITLE attribute should be used if "Reference" is an image. Some browsers use TITLE descriptions to help make web pages accessible to people with vision impairments.

You can easily create links that link to a local document within your site. Just supply the complete path of the target document, with the current document the starting point. Here's are some examples:

```
<a href="section3.htm">Click here for the next section</a> Click here for the next section
```

```
<a href="subdir/Lesson3.htm">Click here for the next Lesson</a> Click here for the next Lesson
```

The first link assumes that section3.htm is in the same directory as the current page, whereas the second assumes Lesson3.htm is stored in the directory "subdir", a sub directory of the current directory.

The ability to make links to documents that reside on computers all over the world makes the web the boundless resource it is today. Links to other

documents are handled by using a more general form of the HREF attribute in the <A> tag.

The most general form is a URL (Uniform Resource Locator). For example, the URL for the "Links to Other Documents" section of the document you are reading is:

`http://www.du.edu/uts/classes/index.html#otherdocs`

This is an absolute reference. Only one location in one document in the entire world matches it. It's also possible (and often desirable) to use relative references instead of absolute references. For example, as you have already seen, an HREF in the same document can specify this target location as:

`#otherdocs`

an HREF in a different document in the same directory can specify this target location as:

`index.html#otherdocs`

an HREF in any document on the www.du.edu server can specify this target location as:

`/uts/classes/index.html#otherdocs`

### **Adding Image links**

Once you understand how to create links in general, creating image links are a snap. Just substitute the text with the <img> tag. For example

`<a href="http://www.B-U.ac.in"></a>`



Notice the blue line surrounding the image- this is how an image link appears by default. We can easily get rid of the border by setting the border attribute to 0:

`<a href="http://www.B-U.ac.in"</a>`

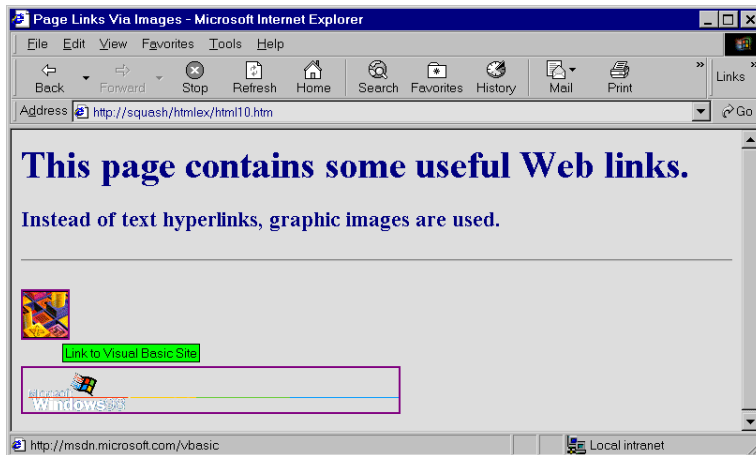


Web pages can use GIF or JPEG images (compression schemes for images). Use forward slashes to provide path to the image file `<IMG SRC="c:/webgifs/mygif1.gif">` even on Windows OS. GIF has maximum of 256 colors (8 bit color). Use GIF for line art e.g., stock charts. JPEG has 24 bit color (16.7 million colors). Use it for human faces, photographic pictures. Use `<ALT>` Tag to provide a text description so that browsers that do not have visual features can still use your information.

Example:

```
<HTML>
<HEAD>
  <TITLE>Page Links Via Images</TITLE>
</HEAD>
<BODY>  <H1>This page contains some useful Web links.</H1>
  <H3>Instead of text hyperlinks, graphic images are used.</H3>
  <HR>
  <P>
    <A HREF="http://msdn.microsoft.com/vbasic">
<IMG SRC="c:/HTMLEx/VB.GIF" ALT="Link to Visual Basic Site"></A>
</P>
  <P>
    <A   HREF="http://www.microsoft.com"><IMG   SRC="winlogo.gif"
ALT="Link to Microsoft"></A>
  </P>
</BODY>
</HTML>
```





## Internal Linking (Linking within the same Document)

We can use the anchor <A> tag to link to another document, or send email.

<A HREF=<http://www.yahoo.com>> link to Yahoo site </A>

<A HREF=moreinfo.htm> For more information, click here </A>

It is also possible to link to different spots within a single large document by using the <A> tag. In this case, it is required that we identify the fragment to link to by giving it a NAME through the <A> tag.

### Example:

```
<!-- misc tags3.htm -->
```

```
<HTML>
```

```
<HEAD>
```

```
<!--This example shows use of linking within the same document. -->
```

```
<TITLE> Linking within the same Document</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<H2> Linking within the same Document </H2>
```

```
<A HREF=#CS305> Course description for CS305 </A>
```

```
<BR>
```

```
<A HREF=#CS455> Course description for CS455 </A>
```

<BR> <BR> <BR>

<A NAME=CS305>

CS 305 <BR>

Introduction to Algorithms. <BR>

This course covers the design and analysis of sequential and  
parallel algorithms. </A>

<BR> <BR> <BR> <BR> <BR> <BR> <BR> <BR> <BR> <BR>

<A NAME=CS455>

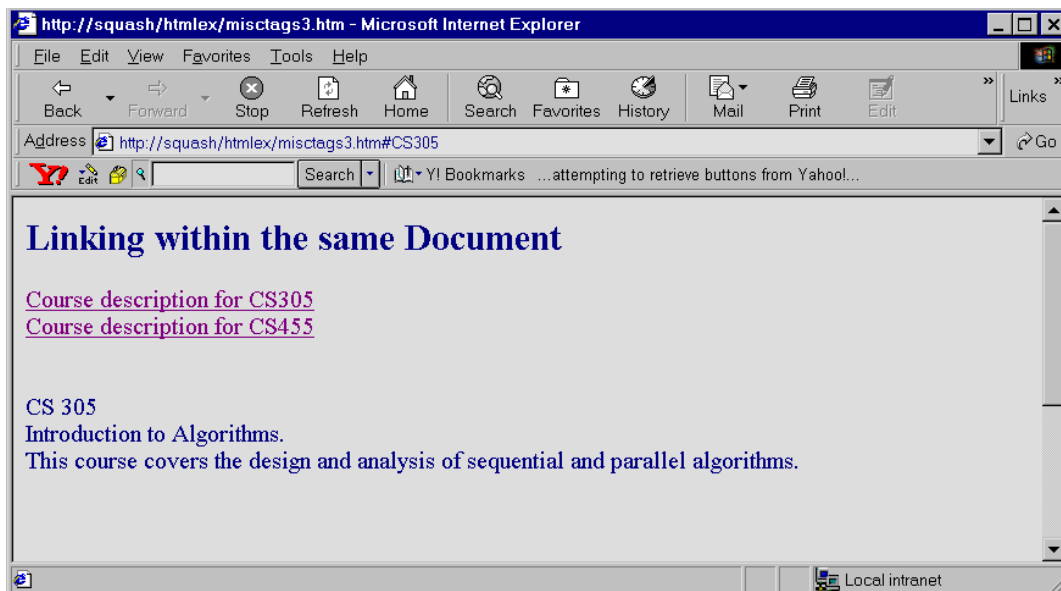
CS 455 <BR>

Introduction to Computer Architecture. <BR>

This course covers the design of a complete 32 bit super pipelined  
RISC processor. </A>

</BODY>

</HTML>



### 3.6. Frames

Frames Break a Web Page Into Sections. One Section in a Frame can be used as Table of Contents.

Frame-Based Pages consist of two main parts

1. Master Page – decides how frames will be displayed
2. Source Pages

Tag	Description
<FRAMESET>	How Master Page is divided into its frame components
<FRAME>	Defines source document for the frame
<NOFRAMES>	Specifies content for browsers that do not use frames
<BASE TARGET>	Specifies a frame that will change based on the selected URL (e.g., URL in table of contents)

Example:

```
<!-- master.htm -->
<HTML>
<HEAD>
<TITLE>Master Frame Page</TITLE>
</HEAD>
<FRAMESET ROWS="20%,80%">
  <FRAME SRC="banner.htm" NAME="banner" MARGINWIDTH="1"
  MARGINHEIGHT="1">
  <FRAMESET COLS="30%,70%">
    <FRAME SRC="sitemap.htm" NAME="sitemap" MARGINWIDTH="1"
    MARGINHEIGHT="1">
    <FRAME SRC="main.htm" NAME="main" MARGINWIDTH="1"
    MARGINHEIGHT="1">
  </FRAMESET>
</FRAMESET>
<NOFRAMES>
<HEAD>
<TITLE>No frames version of the master page</TITLE>
<H2>This is the No Frames Version of the Master Page</H2>
```

```

<HR>
</HEAD>
<BODY>
<H3>Table of Contents</H3>
<P><A HREF="company.htm">Company History</A></P>
<P><A HREF="product.htm">Product Line</A></P>
<P><A HREF="contact.htm">Company Contact Information</A></P>
</BODY>
</NOFRAMES>
</FRAMESET>
</HTML>
<!-- sitemap.htm -->
<HTML>
<HEAD>
<TITLE>Table of Contents Page</TITLE>
<BASE TARGET = "main">
</HEAD>
<H2>Table of Contents Frame</H2>
Clicking hyperlinks in this frame will cause the contents of the main
frame to change.
<P><A HREF="company.htm">Company History</A></P>
<P><A HREF="product.htm">Product Line</A></P>
<P><A HREF="contact.htm">Company Contact Information</A></P>
</BODY>
</HTML>
<!--main.htm -->
<HTML>
<HEAD>
<TITLE>Initial Page for Main Frame</TITLE>

```

```
</HEAD>
```

```
<H2>Main Frame</H2>
```

This is the initial page for the main frame. When hyperlinks are clicked in the table of

contents, the contents of this frame will change.

```
</HTML>
```

```
<!-- company.htm -->
```

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>Company History Page</TITLE>
```

```
</HEAD>
```

```
<H2>Company History</H2>
```

```
</BODY>
```

```
</HTML>
```

```
<!-- product.htm -->
```

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>Product Line Page</title>
```

```
</HEAD>
```

```
<H2>Product Line</H2>
```

```
</BODY>
```

```
</HTML>
```

```
<!-- contact.htm -->
```

```
<HTML>
```

```
<HEAD>
```

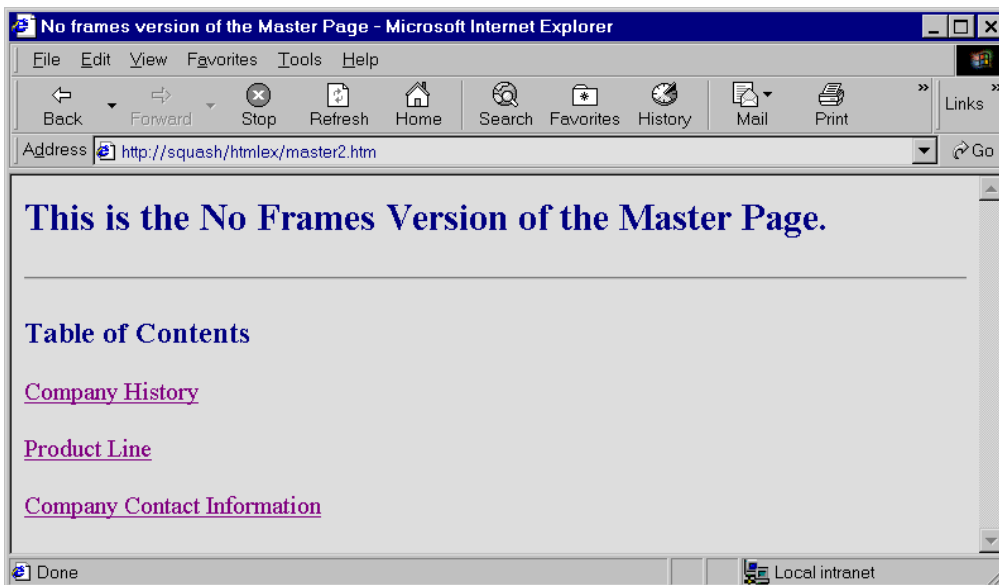
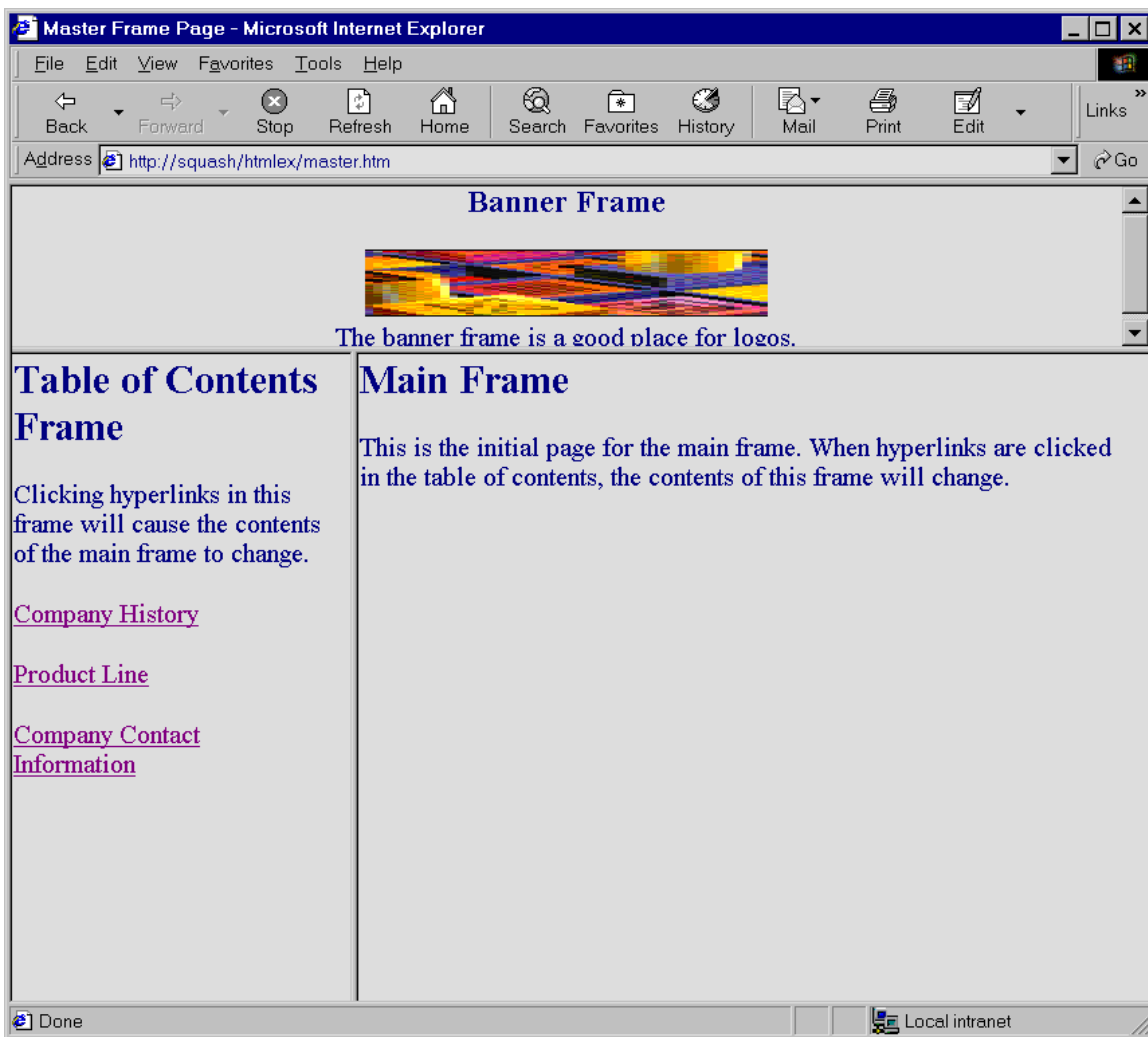
```
<TITLE>Company Contacts Page</TITLE>
```

```
</head>
```

```
<H2>Company Contacts</H2>
```

```
</BODY>
```

```
</HTML>
```



## Image Maps

Image Maps consist of a graphic divided into separate regions. Hot spots can be created in an image map and tied to hyperlinks.

<MAP> specifies hot spots in the image map

<AREA> COORDS attribute of AREA tag identify the X and Y coordinates of each hot spot.

Example:

```
<!-- imagemap.htm -->
```

```
<HTML>
```

```
<HEAD>
```

```
<H2>A hot spot exists for each of the three book catagories listed.</H2>
```

By moving the mouse over each hot spot, the URL associated with each hot spot is displayed in the lower left hand portion of the Web browser.

```
<HR>
```

```
</HEAD>
```

```
<BODY>
```

```
<P><IMG SRC="prima.gif" usemap="#prima"></A>
```

```
<MAP NAME="prima">
```

```
<AREA SHAPE="rect" coords="76,5,216,64" href="lifestyles.htm">
```

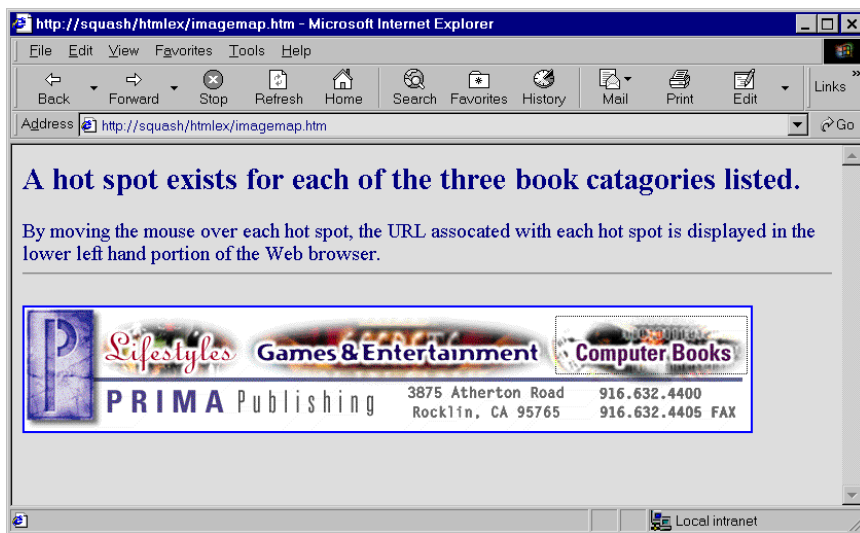
```
<AREA SHAPE="rect" coords="222,7,510,66" href="games_ent.htm">
```

```
<AREA SHAPE="rect" coords="516,8,703,65" href="computer.htm">
```

```
</MAP>
```

```
</BODY>
```

```
</HTML>
```



```
<!-- lifestyles.htm -->
```

```
<HTML>
```

```
<HEAD>
```

```
<TITLE> Lifestyles Page </TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<H1> Page for Lifestyles link </H1>
```

```
</BODY>
```

```
</HTML>
```

```
<!-- games_ent.htm -->
```

```
<HTML>
```

```
<HEAD>
```

```
<TITLE> Games and Entertainment Page </TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<H3> Page for Games and Entertainment </H3>
```

```
</BODY>
```

```
</HTML>
```



```

<!-- computer.htm -->

<HTML>

  <HEAD>

    <TITLE> Computer Books Page </TITLE>

  </HEAD>

  <BODY>

    <H3> Page for Computer Books Listings </H3>

  </BODY>

</HTML>

```

ISMAP corresponds to Server Side Image Maps and USEMAP corresponds to Client side image maps. The AREA attribute in the <MAP> tag can have a SHAPE of either a RECT, POLYGON, CIRCLE, or DEFAULT. RECT region is identified by top left corner of a rectangle and bottom right. The CIRCLE takes three parameters; the center point and the radius. The POLYGON takes coordinates of a collection of lines. DEFAULT is the area specified by no AREA region. An area can also be specified to link to NOHREF.

Example:

```

<!-- imagemap2.htm -->

<HTML>

  <HEAD>

    <TITLE> Image Maps </TITLE>

  </HEAD>

  <BODY>

    <H2> Image Maps - A hot spot exists for different portions on the
    Image</H2>

```

By moving the mouse over each hot spot, the URL associated with each hot spot is

displayed in the lower left hand portion of the Web browser.

```

<P><Align=center>

```

```

        &nbsp;&nbsp;<IMG          SRC="yahooocard.gif"          USEMAP=#MyMap
BORDER=0 WIDTH=200 HEIGHT=200>

```

```

</P>

```

```

<MAP NAME=MyMap>

```

```

    <AREA          SHAPE="rect"          coords="0,0,200,75"
href="http://www.yahoo.com">

```

```

    <AREA          SHAPE="circle"        coords="100,100,25"
href="http://www.amazon.com">

```

```

    <AREA          SHAPE="polygon"        coords="0,200,50,100,100,200"
href="http://www.microsoft.com">

```

```

    <AREA SHAPE="rect" coords="150,150,200,200" NOHREF>

```

```

    <AREA SHAPE="default" href="default.htm">

```

```

</MAP>

```

```

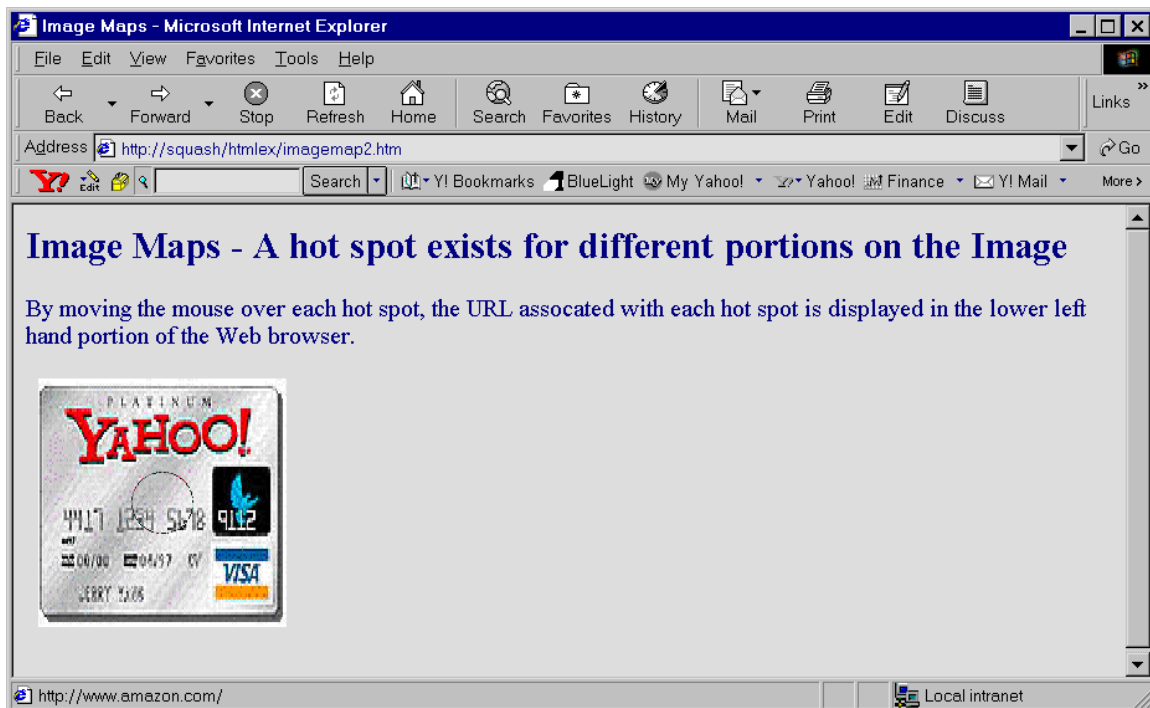
</BODY>

```

```

</HTML>

```



### 3.7. <META> Tag

Provides information about the HTML document such as creation date, author, copyright etc.. It also provides keywords about the document that can be used by Search Engines. META tag can also be used to specify Client Pull Functions . A document can have any number of META tags . META tags are placed in the HEAD section of an HTML document. There are two types of META tags, using either the NAME or HTTP-EQUIV attribute

```
<META HTTP-EQUIV= - - - CONTENT= - - ->
```

```
<META NAME= - - - CONTENT= - - ->
```

The CONTENT attribute provides the values for the HTTP-EQUIV or the NAME field.

- Information provided by the HTTP-EQUIV attribute is added to the http Response header

Client-Pull Examples:

```
<META HTTP-EQUIV=refresh CONTENT=20>
```

This will cause the browser to request the page from the server every 20 seconds. If the server is updating the page periodically, you will see the changes automatically occurring in your browser.

Example:

```
<!-- ClientPull.htm -->
<HTML>
<HEAD>
  <META http-equiv=refresh content=20>
  <TITLE> Client Pull via META tag </TITLE>
</HEAD>
<BODY>
```

Example of Client Pull via META tag. &nbsp;

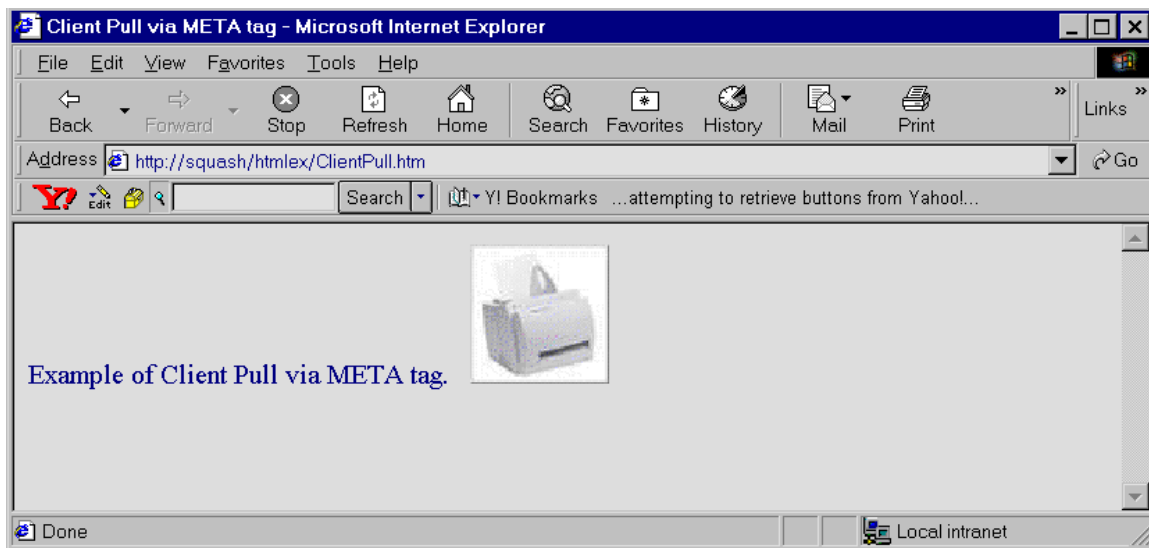
```
<!-- after viewing the page, change the following line to
```

```
  SRC="d:\htmlex\yahocard.gif .. and save it immediately -->
```

```
<IMG SRC="d:\htmlex\hp.jpg" width=100 height=100>
```

```
</BODY>
```

```
</HTML>
```



You can create a slide Show Effect by adding a META refresh type tag and pointing to the next HTML document as demonstrated by the following example:

Example:

```
<!-- Slide1.htm -->

<HTML>

<HEAD>

  <META http-equiv=refresh content="5; URL=slide2.htm">

  <TITLE> Client Pull via META tag </TITLE>

</HEAD>

<BODY>

SLIDE SHOW: SLIDE 1

<IMG SRC="d:\htmlx\hp.jpg" width=100 height=100>

</BODY>

</HTML>

<!-- Slide2.htm -->

<HTML>

<HEAD>

  <META http-equiv=refresh content="5; URL=slide3.htm">

  <TITLE> Client Pull via META tag </TITLE>
```

```

</HEAD>
<BODY>
SLIDE SHOW: SLIDE 2
<IMG SRC="d:\htmlex\yahoocard.gif" width=100 height=100>
</BODY>
</HTML>
<!-- Slide3.htm -->
<HTML>
<HEAD>
  <META http-equiv=refresh content="5; URL=slide1.htm">
  <TITLE> Client Pull via META tag </TITLE>
</HEAD>
<BODY>
SLIDE SHOW: SLIDE 3
<IMG SRC="d:\htmlex\compshop.gif" width=100 height=100>
</BODY>
</HTML>

```

---

### 3.8. Let us Sum Up

---

A list is defined as a sequence of paragraphs, each marked with the list item tag <LI>. The entire sequence of list items is enclosed with the starting and ending tags appropriate to the kind of list. HTML list tags come in **three flavors**:

- Ordered (numbered) lists
- Unordered (bullet) lists
- Definition lists

#### Tables

Tables are specified using <TABLE> </TABLE> tags. Caption for table are specified with the <CAPTION> </CAPTION>. Table caption appear above the table. Tables have header and data cells defined by the empty tags., <TH> and

<TD>. Cells may contain text, paragraphs, lists and headers. Each row of a table is ended by table row tag <TR>.

## Forms

A form is a designated area of an HTML page made available for user input. The basic idea of a form is to present input field to the reader for typing in text information, and radio buttons, check boxes, and pop-up menus for selecting items from option lists. The submit button buttons, when clicked, instructs the browser to take the action specified in the form's Action attribute according to the method specified in the METHOD attribute. There are two action methods, METHOD=GET and METHOD=POST.

Hyperlinks represent the essence of the WWW, linking millions and millions of pages from around the world together. Hyperlinks allow a page to be linked to other Pages. You can also provide email links. Graphic images can also be used to point to a Hyperlink.

HTML differs from traditional formatting languages in the ability to include hyperlinks in documents. This is done with <A> anchor tags.

## Frames

Frames Break a Web Page Into Sections. One Section in a Frame can be used as Table of Contents.

Frame-Based Pages consist of two main parts

Master Page – decides how frames will be displayed

Source Pages

Tag	Description
<FRAMESET>	How Master Page is divided into its frame components
<FRAME>	Defines source document for the frame
<NOFRAMES>	Specifies content for browsers that do not use frames
<BASE TARGET>	Specifies a frame that will change based on the selected URL (e.g., URL in table of contents)

## **<META> Tag**

Provides information about the HTML document such as creation date, author, copyright etc.. It also provides keywords about the document that can be used by Search Engines. META tag can also be used to specify Client Pull Functions . A document can have any number of META tags . META tags are placed in the HEAD section of an HTML document. There are two types of META tags, using either the NAME or HTTP-EQUIV attribute

---

### **3.9. Lesson End Activities**

---

1. What is the purpose of Forms?
2. Explain the usage of <META> tag.
3. Describe different kinds of lists.

---

### **3.10. Check Your Progress**

---

1. Design your home page for your resume using Table for your Academic qualification and image for your photo.
2. Write a HTML programe using Forms to view the result of a student.
3. Write a HTML programe using Frames to list the subject in your syllabus.

---

### **3.11 Reference**

---

1. [WWW.W3C](http://www.w3c.org)
3. Thomas A. Powell, “The Complete Reference HTML and XHTML”, fourth Edition, Tata McGraw Hill.





## JavaScript - Basics

---

### Contents

#### 4.0 Aim and Objective

#### 4.1 Introduction

#### 4.2. Variables

#### 4.3. Escape Sequence

#### 4.4. Expression and Operators

#### 4.5. Special Operators

#### 4.6. Let us Sum Up

#### 4.7. Lesson end Activities

#### 4.8. Check your progress

#### 4.9. Reference

---

#### 4.0. Objective

---

- To be able to write simple JavaScript programs
- To understand arithmetic operation in JavaScript
- To understand decision making statements in JavaScript

---

#### 4.1. Introduction

---

Java Script is an interpreted programming language that is embedded in a web browser. It is not a JAVA Programming Language. To write a Java Script program, Open Window's *Notepad* or your favorite text editor, type in a JavaScript program, and save it as *filename.htm*. Double-clicking *filename.htm* causes your browser to run the program.

Java Scripting can be implemented any one of two types

- Inline scripting, place code INSIDE the HTML code
- Much like an external style sheet in design, file outside of HTML contains JAVASCRIPT code

## Including JavaScript in HTML

Both Inline and External Java Scripting you have to place in the <head> section and it needs both opening and closing <script> tag.

Inline Java Scripting

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

External Java Scripting

```
<script type = "text/javascript" src="javasc1.js">  
</script>
```

## First JavaScript Program

Your first JavaScript Program embedded in HTML "Hello World". First we will create the HTML code that will include the JavaScript and create the JavaScript code.

HTML Code :

```
<head>  
  <title> Hello World </title>  
  <script type= "text/javascript" src="helloworld.js">  
  </script>  
</head>
```

Javascript Code : (helloworld.js)

The file contain only the following line

```
alert("Hello World");
```

alert is a default JavaScript function. It makes an alert box appear on the screen

## Syntax Rules

All default function described in JavaScript are lower case AND case sensitive. All command in JavaScript are case sensitive.

## Use of Comments in JavaScript

Comment used to understand the code for future reference. It won't compile. Java Script provides the comment like C/C++. It is easy to use in JavaScript than in HTML.

Two types of comments are there

- Single line comment `//`
- Multiple line comment `/* code */`

Eg.

### Using Single line Comment

```
// Bharathiar University
// Project 1
// 5/22/08
// Period 1
```

## Multiple line Comment

/\*  
Bharathiar University  
Project 1  
5/23/08  
Period 1  
\*/

## 4.2. Variables

Java Script use **var** to declare ANY type of variable. In Java Script variables are loosely typed. It may or may not have to initialize the variable.

You use variables as symbolic names for values in your application. You give variables names by which you refer to them and which must conform to certain rules.

A JavaScript identifier, or *name*, must start with a letter or underscore (“\_”); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters “A” through “Z” (uppercase) and the characters “a” through “z” (lowercase).

Some examples of legal names are `Number_hits`, `temp99`, and `_name`.

## Declaring Variables

You can declare a variable in two ways:

By simply assigning it a value. For example, `x = 42`

With the keyword `var`. For example, `var x = 42`

## Evaluating Variables

A variable or array element that has not been assigned a value has the value `undefined`. The result of evaluating an unassigned variable depends on how it was declared:

If the unassigned variable was declared without `var`, the evaluation results in a runtime error.

If the unassigned variable was declared with `var`, the evaluation results in the `undefined` value, or `NaN` in numeric contexts.

```
// create a variable and assign a string to it
    var message = "My First JavaScript Variable";
// display the value stored in the variable
    alert( message );
// initialize a number variable
    var a = 0.06;
// initialize a string variable
    var b = "JavaScript in easy steps";
// initialize a boolean variable
    var c = false;
// display the data types of each variable
    alert( typeof a + "\n" + typeof b + "\n" + typeof c );
```

The output of the above looks like



## Variable Scope

When you set a variable identifier by assignment outside of a function, it is called a *global* variable, because it is available everywhere in the current document. When you declare a variable within a function, it is called a *local* variable, because it is available only within the function.

Using `var` to declare a global variable is optional. However, you must use `var` to declare a variable inside a function.

You can access global variables declared in one window or frame from another window or frame by specifying the window or frame name. For example, if a variable called `phoneNumber` is declared in a `FRAMESET` document, you can refer to this variable from a child frame as `parent.phoneNumber`.

## Global variables

Declared in the JavaScript as usual, but not inside a function.

---

## 4.3. Escape Sequences Using Special Characters in Strings -

---

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

```
"one line \n another line"
```

All Escape sequence will start with backslash (`\`)

The following table lists the special characters that you can use in JavaScript strings.

Table 4.1 JavaScript special characters

Special Character	Character Meaning
<code>\b</code>	Backspace
<code>\f</code>	Form feed

<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\'</code>	Apostrophe or single quote
<code>\"</code>	Double quote
<code>\a</code>	audible alert
<code>\v</code>	vertical tab
<code>\\</code>	single backslash character
<code>\?</code>	Question mark

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark.

Eg.:

```
alert( "Hi Friend! \a" );
alert( "Good Luck!!! \n");
alert( " \t You'll need it!!!\t");
```

---

## 4.4. Expressions and Operators

---

### Expressions

An *expression* is any valid set of literals, variables, operators, and expressions that evaluates to a single value; the value can be a number, a string, or a logical value.

Conceptually, there are two types of expressions: those that assign a value to a variable, and those that simply have a value. For example, the expression `x = 7` is an expression that assigns `x` the value seven. This expression itself evaluates to seven. Such expressions use *assignment operators*. On the other hand, the expression `3 + 4` simply evaluates to seven; it does not perform an assignment. The operators used in such expressions are referred to simply as *operators*.

JavaScript has the following types of expressions:

Arithmetic: evaluates to a number, for example 3.14159

String: evaluates to a character string, for example, “Fred” or “234”

Logical: evaluates to true or false

## **Operators**

JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.

Assignment Operators

Comparison Operators

Arithmetic Operators

Bitwise Operators

Logical Operators

String Operators

Special Operators

JavaScript has both *binary* and *unary* operators. A binary operator requires two operands, one before the operator and one after the operator:

*operand1 operator operand2*

For example, 3+4 or x\*y.

A unary operator requires a single operand, either before or after the operator:

*operator operand*

or

*operand operator*

For example, x++ or ++x.

In addition, JavaScript has one ternary operator, the conditional operator. A ternary operator requires three operands.

### **Assignment Operators**

An assignment operator assigns a value to its left operand based on the value of its right operand. The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand.

That is, x = y assigns the value of y to x.

The other assignment operators are shorthand for standard operations, as shown in the following table.

Table 4.2 Assignment operators

Shorthand operator	Meaning
--------------------	---------

<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x &lt;&lt;= y</code>	<code>x = x &lt;&lt; y</code>
<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
<code>x &gt;&gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
<code>x &amp;= y</code>	<code>x = x &amp; y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x  = y</code>	<code>x = x   y</code>

## Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical or string values. Strings are compared based on standard lexicographical ordering, using Unicode values.

### Equal (==)

Returns true if the operands are equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.

```
3 == var1
"3" == var1
3 == '3'
```

### Not equal (!=)



Returns true if the operands are not equal. If the two operands are not of the same type, JavaScript attempts to convert the operands to an appropriate type for the comparison.

```
var1 != 4
```

```
var2 != "3"
```

### **Strict equal (===)**

Returns true if the operands are equal and of the same type.

```
3 === var1
```

### **Strict not equal (!==)**

Returns true if the operands are not equal and/or not of the same type.

```
var1 !== "3"
```

```
3 !== '3'
```

### **Greater than (>)**

Returns true if the left operand is greater than the right operand.

```
var2 > var1
```

### **Greater than or equal (>=)**

Returns true if the left operand is greater than or equal to the right operand.

```
var2 >= var1
```

```
var1 >= 3
```

### **Less than (<)**

Returns true if the left operand is less than the right operand.

```
var1 < var2
```

### **Less than or equal (<=)**

Returns true if the left operand is less than or equal to the right operand.

```
var1 <= var2
```

```
var2 <= 5
```

## Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/). These operators work as they do in most other programming languages, except the / operator returns a floating-point division in JavaScript, not a truncated division as it does in languages such as C or Java. For example:

1/2 //returns 0.5 in JavaScript

1/2 //returns 0 in Java

Explanation of Mathematical Operations

Symbol	Meaning	Example
!	Not	if(answer != 'Q')
++ / --	incrementing/decrementing	covered later
%	modulus – remainder	covered later
+ - / *	normal arithmetic symbols	
<	less than	4 < 8 == True
<=	less than or equal to	6 <= 6 == True
>	greater than	6 > 4 == True
>=	greater than or equal to	6 > = 6 == True
==	used to COMPARE – if they are equal	if ( answer == 'Q')
!=	used to COMPARE – if they are NOT equal	if(answer != 'Q')
&&	And	covered later
	Or	covered later

## Comparison Examples

```
var teststrings1 = ( "JavaScript" == "JavaScript" );
```


```
var teststrings2 = ( "JavaScript" == "javascript" );
```

```

var testnumbers1 = ( 1.785 == 1.785 );
var testnumbers2 = ( 5 != 5 );
var testbooleans1 = ( true == true );
var testbooleans2 = ( false != false );
var testlessthan1 = ( 100 < 200 );
var testlessthan2 = ( 100 < 100 );
var testlessthan_or_equal = ( 100 <= 100 );
var testgreaterthan = ( -1 > 1 );
var a = 8, b = 8.0, testvariables1 = ( a == b );
var c = null, d = null, testvariables2 = ( c == d );
var result = "TEST STRINGS 1: " + teststrings1 + " 2: " + teststrings2 + "\n\n";
result += "TEST NUMBERS 1: " + testnumbers1 + " 2: " + testnumbers2 + "\n\n";
result += "TEST BOOLEANS 1: " + testbooleans1 + " 2: " + testbooleans2 +
"\n\n";
result += "TEST LESS THAN 1: " + testlessthan1 + " 2: " + testlessthan2 + "\n\n";
result += "TEST LESS THAN OR EQUAL: " + testlessthan_or_equal + "\n\n";
result += "TEST GREATER THAN: " + testgreaterthan + "\n\n";
result += "TEST VARIABLES 1: " + testvariables1 + " 2: " + testvariables2 +
"\n\n";
alert(result);

```

### Order of Operations

Highest priority	!	++x	--x
	*	/	% (Modulus)
	+	-	
	<	<=	>
	>=		
	= = (Identical)		! = (Not Identical)
	&& (And)		
Lowest priority	(Or)	x++	x--

- Make sure you use ( )'s to specify order you wish to calculate

```
a = b * c - d % e / f; // not clear
```

```
a = (b * c) - ((d % e) / f); // much clearer
```

```
var addnum = 20 + 30;
```

```
var addstr = "I love " + "JavaScript";
```

```
var sub = 35.75 - 28.25;
```

```
var mul = 8 * 50;
```

```
var mod = 65 % 2;
```

```
var inc = 5 ; inc = ++inc;
```

```
var dec = 5 ; dec = --dec;
```

```
var result = "Addnum is " + addnum + "\n";
```

```
result += "Addstr is " + addstr + "\n";
```

```
result += "Sub is " + sub + "\n";
```

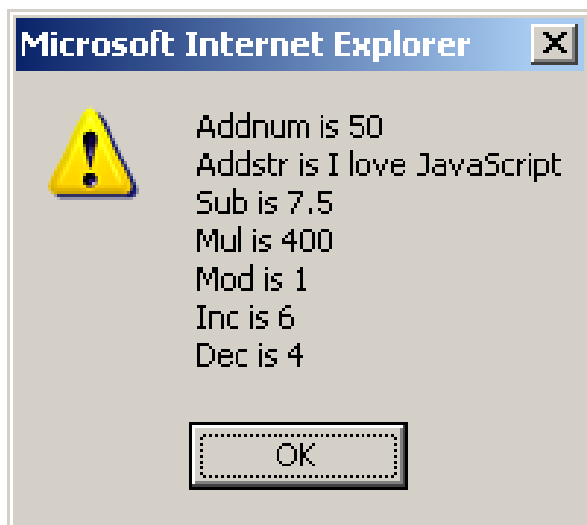
```
result += "Mul is " + mul + "\n";
```

```
result += "Mod is " + mod + "\n";
```

```
result += "Inc is " + inc + "\n";
```

```
result += "Dec is " + dec + "\n";
```

```
alert ( result );
```



## Bitwise Operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

### Bitwise Logical Operators

Conceptually, the bitwise logical operators work as follows:

The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones). Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.

The operator is applied to each pair of bits, and the result is constructed bitwise.

**Table 4.3. Bitwise operators**

Operator	Usage	Description
Bitwise AND	$a \& b$	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	$a   b$	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	$a \wedge b$	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	$\sim a$	Inverts the bits of its operand.
Left shift	$a \ll b$	Shifts $a$ in binary representation $b$ bits to left, shifting in zeros from the right.
Right shift	$a \gg b$	Shifts $a$ in binary representation $b$ bits to right, discarding bits shifted off.

Zero-fill right shift  $a \ggg b$  Shifts  $a$  in binary representation  $b$  bits to the right, discarding bits shifted off, and shifting in zeros from the left.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

$15 \& 9$	yields 9	$(1111 \& 1001 = 1001)$
$15   9$	yields 15	$(1111   1001 = 1111)$
$15 \wedge 9$	yields 6	$(1111 \wedge 1001 = 0110)$

### Bitwise Shift Operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operator.

The shift operators are listed in the following table.

**Table 3.5 Bitwise shift operators**

Operator	Description
<<	(Left shift) This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.
Example $9 \ll 2$ yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.	
>>	(Sign-propagating right shift) This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.

Example `9>>2` yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, `-9>>2` yields -3, because the sign is preserved.

`>>>` (Zero-fill right shift) This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

Example `19>>>2` yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4.

For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

---

## 4.5. Special Operators

---

### **typeof**

The `typeof` operator is used in either of the following ways:

1. `typeof operand`
2. `typeof (operand)`

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional. Suppose you define the following variables:

```
var myFun = new Function("5+2")
var shape="round"
var size=1
var today=new Date()
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun is object
typeof shape is string
typeof size is number
typeof today is object
typeof dontExist is undefined
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

`typeof true` is `boolean`

`typeof null` is `object`

For a number or string, the `typeof` operator returns the following results:

`typeof 62` is `number`

`typeof 'Hello world'` is `string`

For property values, the `typeof` operator returns the type of value the property contains:

`typeof document.lastModified` is `string`

`typeof window.length` is `number`

`typeof Math.LN2` is `number`

For methods and functions, the `typeof` operator returns results as follows:

`typeof blur` is `function`

`typeof eval` is `function`

`typeof parseInt` is `function`

`typeof shape.split` is `function`

For predefined objects, the `typeof` operator returns results as follows:

`typeof Date` is `function`

`typeof Function` is `function`

`typeof Math` is `function`

`typeof Option` is `function`

`typeof String` is `function`

## **void**

The `void` operator is used in either of the following ways:

1. `void (expression)`
2. `void expression`

The `void` operator specifies an expression to be evaluated without returning a value. `expression` is a JavaScript expression to evaluate. The



<A HREF="javascript:void(0)">Click here to do nothing</A>

<A HREF="javascript:void(document.form.submit())"> Click here to  
t</A>

Java Script is an interpreted programming language that is embedded in a web browser. It is not a JAVA Programming Language. T

- Inline scripting, place code INSIDE the HTML code
- Much like an external style sheet in design, file outside of HTML contains JAVASCRIPT code

Two types of comments are there

- Single line comment `//`
- Multiple line comment `/* code */`

Java Script use **var** to declare ANY type of variable. In Java Script variables are loosely typed. It may or may not have to initialize the variable.

Special Character	Character Meaning
-------------------	-------------------

<code>\b</code>	Backspace
<code>\f</code>	Form feed

<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\'</code>	Apostrophe or single quote
<code>\"</code>	Double quote
<code>\a</code>	audible alert
<code>\v</code>	vertical tab
<code>\\</code>	single backslash character
<code>\?</code>	Question mark

## Expressions

An *expression* is any valid set of literals, variables, operators, and expressions that evaluates to a single value; the value can be a number, a string, or a logical value.

## Operators

JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.

Assignment Operators

Comparison Operators

Arithmetic Operators

Bitwise Operators

Logical Operators

String Operators

Special Operators

Explanation of Mathematical Operations

Symbol	Meaning	Example
!	Not	<code>if(answer != 'Q')</code>
<code>++ / --</code>	incrementing/decrementing	covered later
%	modulus – remainder	covered later

+ - / *	normal arithmetic symbols	
<	less than	4 < 8 == True
<=	less than or equal to	6 <= 6 == True
>	greater than	6 > 4 == True
>=	greater than or equal to	6 >= 6 == True
==	used to COMPARE – if they are equal	if ( answer == 'Q' )
!=	used to COMPARE – if they are NOT equal	if( answer != 'Q' )
&&	And	covered later
	Or	covered later

### Order of Operations

<b>Highest priority</b>	!	++x	--x
	*	/	% (Modulus)
	+	-	
	<	<=	> >=
	== (Identical) != (Not Identical)		
	&& (And)		
<b>Lowest priority</b>	(Or)	x++	x--

### Bitwise Operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

## Bitwise Logical Operators

Operator	Usage	Description
Bitwise AND	$a \& b$	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	$a   b$	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	$a \wedge b$	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	$\sim a$	Inverts the bits of its operand.
Left shift	$a \ll b$	Shifts $a$ in binary representation $b$ bits to left, shifting in zeros from the right.
Right shift	$a \gg b$	Shifts $a$ in binary representation $b$ bits to right, discarding bits shifted off.
Zero-fill right shift	$a \ggg b$	Shifts $a$ in binary representation $b$ bits to the right, discarding bits shifted off, and shifting in zeros from the left.

## Bitwise shift operators

Operator	Description
$\ll$	(Left shift) This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.
$\gg$	(Sign-propagating right shift) This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.

>>> (Zero-fill right shift) This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

## **typeof**

The typeof operator is used in either of the following ways:

1. `typeof operand`
2. `typeof (operand)`

The typeof operator returns a string indicating the type of the unevaluated operand. operand is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional. Suppose you define the following variables:

## **void**

The void operator is used in either of the following ways:

1. `void (expression)`
2. `void expression`

The void operator specifies an expression to be evaluated without returning a value. expression is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them. You can use the void operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document. The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, void(0) evaluates to 0, but that has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())"> Click here to submit</A>
```

---

#### **4.7 Lesson End Activities**

---

1. What is the need of variable?
2. What is bit wise operator
3. What is the purpose of typedef operator?

---

#### **4.8 Check your progress**

---

1. Explain Operator hierarchy with an example.
2. Write a program for arithmetic operation.

---

#### **4.9 Reference**

---

1. [WWW.W3C](http://WWW.W3C)
2. Thomas A. Powell, “The Complete Reference HTML and XHTML”, fourth Edition, Tata McGraw Hill.
3. Deitel, Deitel, Nieto, “Internet and World Wide Web – How to Program”, Pearson Education Asia, 2003

## JavaScript – Control Structures

---

### Contents

#### 5.0. Aim and Objectives

#### 5.1. Introduction

#### 5.2. Control Statements

#### 5.3. loop statements

#### 5.4. label statement

#### 5.5. break statement

#### 5.6. Logical Operator

#### 5.7. Let us Sum Up

#### 5.8. Lesson end Activities

#### 5.9. Check your Progress

#### 5.10. Reference

---

### 5.0. Aim and Objectives

---

- To understand the programme flow
- To learn various control structure available in JavaScript

---

### 5.1. Introduction

---

JavaScript supports a compact set of statements that you can use to incorporate a great deal of interactivity in Web pages.

---

### 5.2. Conditional Statements

---

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

#### **if...else Statement**

##### **if** structure

- is a single-selection structure because it selects or ignores a single action.

if (condition 1)                      (if) →

- **if/else** structure

The **if/else** structure is called a double-selection structure, because it selects between two different actions. if/else's' tells the program to choose and execute one or the other body of code, depending on the values or conditions



example:

```
if(grade >= 70)
{
    alert("You passed");
}
else
{
    alert("You failed");
}
```

Use the if statement to perform certain statements if a logical condition is true; use the optional else clause to perform other statements if the condition is false. An if statement looks as follows:

```
if (condition) {
    statements1
}
else {
    statements2
}
```



The condition can be any JavaScript expression that evaluates to true or false. The statements to be executed can be any JavaScript statements, including further nested if statements. If you want to use more than one statement after an if or else statement, you must enclose the statements in curly braces, {}.

Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object. Any object whose value is not undefined or null, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. For example:

```
var b = new Boolean(false);  
  
if (b) // this condition evaluates to true
```

**Example.** In the following example, the function checkData returns true if the number of characters in a Text object is three; otherwise, it displays an alert and returns false.

```
function checkData () {  
    if (document.form1.threeChar.value.length == 3) {  
        return true  
    } else {  
        alert("Enter          exactly          three          characters.          "          +  
            document.form1.threeChar.value + " is not valid.")  
        return false  
    }  
}
```

## **switch Statement**

To use a switch statement if we know a limited range of ONE variable

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

Keywords used:

- case
  - denotes a clause
- default
  - if no clauses match, uses THIS line as a default clause
  - does NOT have a break after since LAST in list
- break
  - all cases use, after LAST line in case statement
  - breaks out of switch statement

A switch statement looks as follows:

```
switch (expression){  
  case label :  
    statement;  
  break;  
  case label :  
    statement;  
  break;  
  ...  
  default : statement;  
}
```

The program first looks for a label matching the value of expression and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of switch.

The optional break statement associated with each case label ensures that the program breaks out of switch once the matched statement is executed and continues execution at the statement following switch. If break is omitted, the program continues execution at the next statement in the switch statement.

**Example.** In the following example, if `expr` evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When `break` is encountered, the program terminates switch and executes the statement following switch. If `break` were omitted, the statement for case "Cherries" would also be executed.

```
switch (expr) {  
  case "Oranges" :  
    document.write("Oranges are Rs.40 per Kg.<BR>");  
    break;  
  case "Apples" :  
    document.write("Apples are Rs.80 per Kg.<BR>");  
    break;  
  case "Bananas" :  
    document.write("Bananas are Rs. 24 per Kg.<BR>");  
    break;  
  case "Cherries" :  
    document.write("Cherries are Rs.40 per Kg.<BR>");  
    break;  
  default :  
    document.write("Sorry, we are out of " + i + "<BR>");  
}  
document.write("Is there anything else you'd like?<BR>");
```

---

### 5.3. Loop Statements

---

A loop is a set of commands that executes repeatedly until a specified condition is met. JavaScript supports the `for`, `do while`, `while`, and `label loop` statements (`label` is not itself a looping statement, but is frequently used with these statements). In addition, you can use the `break` and `continue` statements within loop statements.

## for Statement

A for loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C for loop. A for statement looks as follows:

```
for ([initialExpression]; [condition]; [incrementExpression]) {  
  statements  
}
```

When a for loop executes, the following occurs:

### 1. Initialization

The initializing expression initial-expression, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity. In any loop that we use, a variable will be the sole controller of when the loop is started and ended. The variable can be a number that will need to change from a starting value, to an ending value, usually the start and ending values are big enough to warrant a loop. The variable can also be a char, and when the char finally matches, the loop ends.

### 2. Testing/Condition

In any loop we create, there MUST be a condition where our variable will fail (so the loop will end). Conditions in a loop will look the same conditions we covered in IF statements. The condition expression is evaluated. If the value of condition is true, the loop statements execute. If the value of condition is false, the for loop terminates.

3. The statements execute.

4. The update expression increment Expression executes, and control returns to Step 2.

## Incrementing/Decrementing

++	- add one
--	- subtract one
x++	- value changed AFTER modification/loop in code completed
++x	- value changed BEFORE modification/loop in code completed

**Example.** The following function contains a for statement that counts the number of selected options in a scrolling list (a Select object that allows multiple selections). The for statement declares the variable i and initializes it to zero. It checks that i is less than the number of options in the Select object, performs the succeeding if statement, and increments i by one after each pass through the loop.

```
<SCRIPT>

function howMany(selectObject) {
    var numberSelected=0
    for (var i=0; i < selectObject.options.length; i++) {
        if (selectObject.options[i].selected==true)
            numberSelected++
    }
    return numberSelected
}

</SCRIPT>

<P><B>Choose some music types, then click the button below:</B>
<BR><SELECT NAME="musicTypes" MULTIPLE>
<OPTION SELECTED> R&B
<OPTION> Jazz
<OPTION> Blues
<OPTION> New Age
<OPTION> Classical
<OPTION> Opera
</SELECT>
<P><INPUT TYPE="button" VALUE="How many are selected?"
onClick="alert ('Number of options selected: ' +
howMany(document.selectForm.musicTypes))">
</FORM>
```

## **do...while Statement**

The do...while statement repeats until a specified condition evaluates to false. A do...while statement looks as follows:

```
do {  
    statement  
} while (condition)
```

statement executes once before the condition is checked. If condition returns true, the statement executes again. At the end of every execution, the condition is checked. When the condition returns false, execution stops and control passes to the statement following do...while.

**Example.** In the following example, the do loop iterates at least once and reiterates until i is no longer less than 5.

```
do {  
    i+=1;  
    document.write(i);  
} while (i<5);
```

## **while Statement**

A while statement executes its statements as long as a specified condition evaluates to true. A while statement looks as follows:

```
while (condition) {  
    statements  
}
```

If the condition becomes false, the statements within the loop stop executing and control passes to the statement following the loop.

The condition test occurs before the statements in the loop are executed. If the condition returns true, the statements are executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following while.

**Example 1.** The following while loop iterates as long as n is less than three:

```
n = 0
x = 0
while( n < 3 ) {
  n ++
  x += n
}
```

With each iteration, the loop increments n and adds that value to x. Therefore, x and n take on the following values:

After the first pass: n = 1 and x = 1

After the second pass: n = 2 and x = 3

After the third pass: n = 3 and x = 6

After completing the third pass, the condition  $n < 3$  is no longer true, so the loop terminates.

**Example 2: infinite loop.** Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate. The statements in the following while loop execute forever because the condition never becomes false:

```
while (true) {
  alert("Hello, world")
}
```

---

## 5.4. label Statement

---

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

The syntax of the label statement looks like the following:

```
label :
  statement
```

The value of *label* may be any JavaScript identifier that is not a reserved word. The *statement* that you identify with a label may be any type.

**Example.** In this example, the label markLoop identifies a while loop.

```
markLoop:
while (theMark == true)
doSomething();
}
```

---

## 5.5. break Statement

---

Use the break statement to terminate a loop, switch, or label statement.

When you use break with a while, do-while, for, or switch statement, break terminates the innermost enclosing loop or switch immediately and transfers control to the following statement.

When you use break within an enclosing label statement, it terminates the statement and transfers control to the following statement. If you specify a label when you issue the break, the break statement terminates the specified statement.

The syntax of the break statement looks like the following:

1. break
2. break [*label*]

The first form of the syntax terminates the innermost enclosing loop, switch, or label; the second form of the syntax terminates the specified enclosing label statement.

**Example.** The following example iterates through the elements in an array until it finds the index of an element whose value is theValue:

```
for (i = 0; i < a.length; i++) {
if (a[i] = theValue);
break;
}
```

---

## 5.6. continue Statement

---

The continue statement can be used to restart a while, do-while, for, or label statement.



In a while or for statement, continue terminates the current loop and continues execution of the loop with the next iteration. In contrast to the break statement, continue does not terminate the execution of the loop entirely. In a while loop, it jumps back to the condition. In a for loop, it jumps to the increment-expression.

In a label statement, continue is followed by a label that identifies a label statement. This type of continue restarts a label statement or continues execution of a labelled loop with the next iteration. Continue must be in a looping statement identified by the label used by continue.

The syntax of the continue statement looks like the following:

1. continue
2. continue [*label*]

**Example 1.** The following example shows a while loop with a continue statement that executes when the value of i is three. Thus, n takes on the values one, three, seven, and twelve.

```
i = 0
n = 0
while (i < 5) {
    i++
    if (i == 3)
        continue
    n += i
}
```

**Example 2.** A statement labeled checkiandj contains a statement labeled checkj. If continue is encountered, the program terminates the current iteration of checkj and begins the next iteration. Each time continue is encountered, checkj reiterates until its condition returns false. When false is returned, the remainder of the checkiandj statement is completed, and checkiandj reiterates until its condition returns false. When false is returned, the program continues at the statement following checkiandj. If continue had a label of checkiandj, the program would continue at the top of the checkiandj statement. checkiandj :

```

while (i<4) {
document.write(i + "<BR>");
i+=1;
checkj :
while (j>4) {
document.write(j + "<BR>");
j-=1;
if ((j%2)==0);
continue checkj;
document.write(j + " is odd.<BR>");
}
document.write("i = " + i + "<br>");
document.write("j = " + j + "<br>");
}

```

---

## 5.7. Logical Operators

---

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the `&&` and `||` operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

**Table 5.1. Logical Operator**

Operator	Usage	Description
<code>&amp;&amp;</code>	<code>expr1 &amp;&amp; expr2</code>	(Logical AND) Returns <code>expr1</code> if it can be converted to false; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>&amp;&amp;</code> returns true if both operands are true; otherwise, returns false.

<code>  </code>	<code>expr1    expr2</code>	(Logical OR) Returns <code>expr1</code> if it can be converted to true; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>  </code> returns true if either operand is true; if both are false, returns false.
<code>!</code>	<code>!expr</code>	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

The following code shows examples of the `&&` (logical AND) operator.

```

a1=true && true    // t && t returns true
a2=true && false    // t && f returns false
a3=false && true    // f && t returns false
a4=false && (3 == 4) // f && f returns false
a5="Cat" && "Dog"   // t && t returns Dog
a6=false && "Cat"   // f && t returns false
a7="Cat" && false   // t && f returns false

```

The following code shows examples of the `||` (logical OR) operator.

```

1=true || true // t || t returns true
2=false || true // f || t returns true
3=true || false // t || f returns true
4=false || (3 == 4) // f || f returns false
5="Cat" || "Dog" // t || t returns Cat
6=false || "Cat" // f || t returns Cat
7="Cat" || false // t || f returns Cat

```

The following code shows examples of the `!` (logical NOT) operator.

```

n1=!true // !t returns false
n2=!false // !f returns true
n3!="Cat" // !t returns false

```

## Short-Circuit Evaluation

As logical expressions are evaluated left to right, they are tested for possible “short-circuit” evaluation using the following rules:

`false && anything` is short-circuit evaluated to false.

`true || anything` is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

---

## 5.8. Let us Sum Up

---

JavaScript supports a compact set of statements that you can use to incorporate a great deal of interactivity in Web pages.

### Conditional Statements

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

#### **if...else Statement**

Use the `if` statement to perform certain statements if a logical condition is true; use the optional `else` clause to perform other statements if the condition is false. The condition can be any JavaScript expression that evaluates to true or false. The statements to be executed can be any JavaScript statements, including further nested `if` statements. If you want to use more than one statement after an `if` or `else` statement, you must enclose the statements in curly braces, `{}`.

#### **switch Statement**

A `switch` statement allows a program to evaluate an expression and attempt to match the expression’s value to a case label. If a match is found, the program executes the associated statement.

### Loop Statements

A loop is a set of commands that executes repeatedly until a specified condition is met. JavaScript supports the `for`, `do while`, `while`, and `label loop` statements (`label` is not itself a looping statement, but is frequently used with

these statements). In addition, you can use the break and continue statements within loop statements.

### **for Statement**

A for loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C for loop.

### **do...while Statement**

The do...while statement repeats until a specified condition evaluates to false.

### **while Statement**

A while statement executes its statements as long as a specified condition evaluates to true.

### **label Statement**

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

### **break Statement**

Use the break statement to terminate a loop, switch, or label statement.

When you use break with a while, do-while, for, or switch statement, break terminates the innermost enclosing loop or switch immediately and transfers control to the following statement.

### **continue Statement**

The continue statement can be used to restart a while, do-while, for, or label statement.

### **Logical Operators**

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.

---

### **5.9. Lesson end Activities**

---

1. What is the advantage of switch statement.
2. Explain logical operator in JavaScript.

---

### **5.10. Check your Progress**

---

1. What is the purpose of break and continue statement.
2. Write a javascript program using for loop for displaying number 1 to 100.

---

### **5.11. Reference**

---

1. [WWW.W3C](http://www.w3c.org)
2. Thomas A. Powell, “The Complete Reference HTML and XHTML”, fourth Edition, Tata McGraw Hill.
3. Deitel, Deitel, Nieto, “Internet and World Wide Web – How to Program”, Pearson Education Asia, 2003

## JavaScript - Functions

---

### Contents

#### 6.0. Aim and Objectives

#### 6.1. Introduction

#### 6.2. Programmer Defined function

#### 6.3. Function calling function and Multiple Arguments

#### 6.4. Scope Rule

#### 6.5. Predefined Functions

#### 6.6. Recursion

#### 6.7. Recursion vs Iteration

#### 6.8. Let us Sum Up

#### 6.9. Lesson end Activities

#### 6.10. Check your Progress

#### 6.11. Reference

---

#### 6.0. Aim and Objectives

---

**To understand how to construct program modularly**

**To be able to create new function**

---

#### 6.1. Introduction

---

Best way to develop and maintain a large program is to construct it from small, simple pieces, or modules.

#### **Program Modules in JavaScript**

Modules in JavaScript is called functions. The predefined functions that belong to JavaScript objects are often called methods. The programmer can write functions to define specific tasks that may be used at many points in a script. These functions are referred to as programmed defined functions.

A function is invoked by a function call. The function call specifies the function name and provides information that the called function needs to perform its task.

---

## 6.2. Programmer-Defined functions

---

Functions allow the programmer to modularize a program. All variables declared in function definitions are local variables – this means that they are known only in the function in which they are defined. Most functions have a list of parameters that provide the means for communicating information between functions via function call.

### Defining Functions

A function definition consists of the function keyword, followed by the name of the function. A list of arguments to the function enclosed in parentheses and separated by commas. The JavaScript statements that define the function are enclosed in curly braces, { }. The statements in a function can include calls to other functions defined in the current application.

Generally, you should define all your functions in the HEAD of a page so that when a user loads the page, the functions are loaded first. Otherwise, the user might perform an action while the page is still loading that triggers an event handler and calls an undefined function, leading to an error.

Function Syntax : HTML code

```
<html>
<head>
  <title>First Function</title>
  <script type="text/javascript" src="first-fcn.js">
  </script>
</head>
<body onload = "call_alert()">
  <p> </p>
</body>
</html>
```



First-fcn.js file

```
// a function to display a message in an alert dialog

function call_alert()
{
    alert( "My First JavaScript Function" );
}
```

Output of the above



### 6.3. Functions calling Functions & Multiple arguments.

Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. *Calling* the function actually performs the specified actions with the indicated parameters.

The call to the other function will look exactly the same as if it came from the HTML file. There may be more than one time where a function calls other functions.

Call from HTML

```
<html>
<head>
    <title>Multiple Functions</title>
    <script type="text/javascript" src="multi-fcn.js">
        </script>
</head>
    <body onload = "call_alert(4)">
        <p> </p>
</body>
</html>
```

Call from another function

```
// a function to call a second function
// and display a returned result
function call_alert( num )
{
    var new_number = make_double( num );
    alert( "The Value Is " + new_number );
}

// a function to double a value
// and return the result to the caller
function make_double( num )
{
    var double_num = num + num;
    return double_num;
}
```

---

## 6.4. Scope Rules

---

The scope of an identifier for a variable functions is the portion of the program in which the identifier can be referenced. Global variables or script levels variables are declared in head elements of the documents and they can be accessible in any part of a script and are said to have global scope.

Identifiers declared inside a function have function local scope and can be used only in that function. Function scope begins with the opening left brace ( { ) of the function in which the identifier is declared and ends at the terminating right brace ( } ) of the function. If the local variable in function has the same name as global variable, the global variable is hidden from the body of the function.

Example

```
<html>

<head>

<title> A Scope Example</title>
```

```

<script type = "text/javascript">
var x = 1;
function start()
{
    var x = 5;
    document.writeln("local x in start is " + x + "</p>");

    functionA();
    functionB();

    document.writeln("local x in start is " + x + "</p>");
}
Function functionA()
{
    var x = 25;
    document.writeln("local x in start is " + x + "</p>");
}
Function functionB()
{
    document.writeln("Global x in start is " + x + "</p>");
}
</script>
</head>
<body onload = start() > </body>
</html>

```

---

## 6.5. Predefined Functions

---

JavaScript has several top-level predefined functions:

- `eval`
- `isFinite`
- `isNaN`
- `parseInt` and `parseFloat`
- `Number` and `String`
- `escape` and `unescape`

### **eval Function**

The `eval` function evaluates a string of JavaScript code without reference to a particular object. The syntax of `eval` is:

`eval(expr)`

where *expr* is a string to be evaluated.

If the string represents an expression, `eval` evaluates the expression. If the argument represents one or more JavaScript statements, `eval` performs the statements. Do not call `eval` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

### **isFinite Function**

The `isFinite` function evaluates an argument to determine whether it is a finite number. The syntax of `isFinite` is:

`isFinite(number)`

where *number* is the number to evaluate.

If the argument is NaN, positive infinity or negative infinity, this method returns false, otherwise it returns true.

The following code checks client input to determine whether it is a finite number.

```
if(isFinite(ClientInput) == true)
{
    /* take specific steps */
}
```

## **isNaN Function**

The isNaN function evaluates an argument to determine if it is “NaN” (not a number). The syntax of isNaN is:

```
isNaN(testValue)
```

where testValue is the value you want to evaluate.

The parseFloat and parseInt functions return “NaN” when they evaluate a value that is not a number. isNaN returns true if passed “NaN,” and false otherwise.

The following code evaluates floatValue to determine if it is a number and then calls a procedure accordingly:

```
floatValue=parseFloat(toFloat)
if (isNaN(floatValue)) {
  notFloat()
} else {
  isFloat()
}
```

## **parseInt and parseFloat Functions**

The two “parse” functions, parseInt and parseFloat, return a numeric value when given a string as an argument. The syntax of parseFloat is

```
parseFloat(str)
```

where parseFloat parses its argument, the string str, and attempts to return a floating-point number. If it encounters a character other than a sign (+ or -), a numeral (0-9), a decimal point, or an exponent, then it returns the value up to that point and ignores that character and all succeeding characters. If the first character cannot be converted to a number, it returns “NaN” (not a number). The syntax of parseInt is

```
parseInt(str [, radix])
```

parseInt parses its first argument, the string str, and attempts to return an integer of the specified radix (base), indicated by the second, optional argument, radix. For example, a radix of ten indicates to convert to a decimal number, eight octal, sixteen hexadecimal, and so on. For radices above ten, the

letters of the alphabet indicate numerals greater than nine. For example, for hexadecimal numbers (base 16), A through F are used.

If `parseInt` encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If the first character cannot be converted to a number in the specified radix, it returns “NaN.” The `parseInt` function truncates the string to integer values.

## **Number and String Functions**

The Number and String functions let you convert an object to a number or a string. The syntax of these functions is:

```
Number(objRef)
```

```
String(objRef)
```

where `objRef` is an object reference.

The following example converts the Date object to a readable string.

```
D = new Date (430054663215)
// The following returns
// "Thu Aug 18 04:37:43 GMT-0530 (Indian Standard Time) 2008"
x = String(D)
```

## **escape and unescape Functions**

The escape and unescape functions let you encode and decode strings. The escape function returns the hexadecimal encoding of an argument in the ISO Latin character set. The unescape function returns the ASCII string for the specified hexadecimal encoding value. The syntax of these functions is:

```
escape(string)
```

```
unescape(string)
```

These functions are used primarily with server-side JavaScript to encode and decode name/value pairs in URLs.

```
escape(string)
```

```
unescape(string)
```

These functions are used primarily with server-side JavaScript to encode and decode name/value pairs in URLs.

---

## 6.6. Recursion

---

A recursion function is a function that calls itself, either directly, or indirectly through another function.

```
<html>
<head>
<title> Recursice Factorial Function</title>
<script language = "javascript">
    document.writeln("<h1> Factorial of 1 to 10 </h1>");
    for (var = 0; i <= 10; i++)
        document.writeln(i "</p>" + factorial(i));
    function factorial ( number )
    {
        if (number <= 1)
            return 1;
        else
            return number * factorial(number - 1);
    }
</script>
</head>
<body> </body>
</html>
```

---

## 6.7. Recursion vs Iteration

---

Both iteration and recursion are based on a control statement; Iteration uses a repetition statement; recursion uses a selection statement. Both iteration and recursion involves repetition; Iteration explicitly uses a repetition statement; recursion achieves repetition through repeated function calls. Iteration and recursion each involve a termination test; Iteration terminates when the loop continuation condition fails; recursion terminates when a base case is recognized.

---

## 6.8. Let us Sum Up

---

Best way to develop and maintain a large program is to construct it from small, simple pieces, or modules.

Modules in JavaScript is called functions. The predefined functions that belong to JavaScript objects are often called methods. The programmer can write functions to define specific tasks that may be used at many points in a script. These functions are referred to as programmed defined functions.

A function definition consists of the function keyword, followed by the name of the function. A list of arguments to the function enclosed in parentheses and separated by commas. The JavaScript statements that define the function are enclosed in curly braces, `{ }`. The statements in a function can include calls to other functions defined in the current application.

Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. *Calling* the function actually performs the specified actions with the indicated parameters.

The call to the other function will look exactly the same as if it came from the HTML file. There may be more than one time where a function calls other functions.

The scope of an identifier for a variable functions is the portion of the program in which the identifier can be referenced. Global variables or script levels variables are declared in head elements of the documents and they can be accessible in any part of a script and are said to have global scope.

### **Predefined Functions**

JavaScript has several top-level predefined functions:

- `eval`
- `isFinite`
- `isNaN`
- `parseInt` and `parseFloat`
- `Number` and `String`
- `escape` and `unescape`



above ten, the letters of the alphabet indicate numerals greater than nine. For example, for hexadecimal numbers (base 16), A through F are used.

If parseInt encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If the first character cannot be converted to a number in the specified radix, it returns “NaN.” The parseInt function truncates the string to integer values.

## **Recursion**

A recursion function is a function that calls itself, either directly, or indirectly through another function.

### **Recursion vs Iteration**

Both iteration and recursion are based on a control statement; Iteration uses a repetition statement; recursion uses a selection statement. Both iteration and recursion involves repetition; Iteration explicitly uses a repetition statement; recursion achieves repetition through repeated function calls. Iteration and recursion each involve a termination test; Iteration terminates when the loop continuation condition fails; recursion terminates when a base case is recognized.

---

## **6.9. Lesson end Activities**

---

1. What is local and global variable?
2. Explain any two predefined function.

---

## **6.10. Check your progress**

---

1. How values are transfer into a function body?
2. Write a recursion program.

---

## **6.11. Reference**

---

1. [WWW.W3C](http://WWW.W3C)
2. Thomas A. Powell, “The Complete Reference HTML and XHTML”, fourth Edition, Tata McGraw Hill.
3. Deitel, Deitel, Nieto, “Internet and World Wide Web – How to Program”, Pearson Education Asia, 2003



## JavaScript - Arrays

---

### Contents

#### 7.0. Aim and Objective

#### 7.1. Introduction

#### 7.2. Array Objects

#### 7.3. Using Arrays

#### 7.4. Arrays Methods

#### 7.5. Deleting array elements

#### 7.6. Passing array to function

#### 7.7. Sorting Arrays

#### 7.8. Searching Arrays

#### 7.9. Multi Dimensional Arrays

#### 7.10. Let us Sum Up

#### 7.11. Lesson end Activities

#### 7.12. Check your Progress

#### 7.13. Reference

---

#### 7.0. Aim and Objective

---

- **To understand JavaScript Arrays concepts**
- **To learn Javascript Array processing**

---

#### 7.1. Introduction

---

Arrays are one of the most powerful programming tools available. They provide the programmer with a way of organizing a collection of homogeneous data items (i.e. items that have the same type and the same length) into a single data structure. An **array**, then, is a data structure that is made up of a number of variables all of which have the same data type; for example, all the exam scores for a class of ten MIS students.

The individual data items that make up the array are referred to as the *elements* of the array. Elements in the array are distinguished from one another by the use of an *index* or *subscript*, enclosed in parentheses or brackets, following the array name. The subscript indicates the position of an element within the array. The subscript or index may be a number or a variable, and may then be used to access any item within the valid bounds of an array.

Arrays are an internal data structure, i.e. they are required only for the duration of the program in which they are defined. They are a very convenient mechanism for storing and manipulating a collection of similar data items in a program, and a programmer should be familiar with the operations most commonly performed on them. The most typical operations performed on arrays are:

- loading a set of initial values into the elements of an array;
- processing the elements of an array;
- searching an array, using a linear or binary search, for a particular element; and
- Writing out the contents of an array to a report.

---

## 7.2. Array Object

---

JavaScript does not have an explicit array data type. However, you can use the predefined Array object and its methods to work with arrays in your applications. The Array object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions. Arrays can grow dynamically – just add new elements at the end. Arrays can have holes, elements that have no value. Array elements can be anything numbers, strings, or arrays!

### Creating an Array

An array in JavaScript is an **Array** object. The process of creating new objects is known as *creating an instance*. All objects are created in JavaScript using the **new** keyword. The syntax for declaring an array is

```
1. arrayObjectName = new Array(element0, element1, ..., elementN)
```

2. `arrayObjectName = new Array(arrayLength)`

`arrayObjectName` is either the name of a new object or a property of an existing object. When using `Array` properties and methods, `arrayObjectName` is either the name of an existing `Array` object or a property of an existing object.

`element0, element1, ..., elementN` is a list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's `length` property is set to the number of arguments.

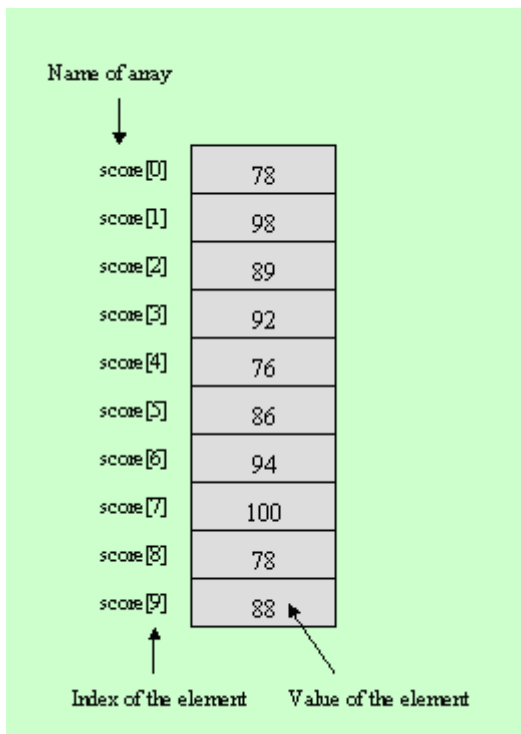
`arrayLength` is the initial length of the array.

```
var arrayName = new Array(n);
```

where the keyword **Array** is used to indicate that the object is an array, and **n** is the number of elements in the array. For example, to declare an array called **score** that has 10 elements, you would write

```
var score = new Array(10);
```

Arrays occupy space in memory. The array declaration instructs the computer to reserve ten blocks of memory. Each block of memory will be able to hold a single value from the **score** array. Each value is assigned to a position within the array, called the element/s index. You can use the index to refer to individual elements in the array.



Above Figure illustrates an array of integer values named **score** used for a 10-student MCA class. The *name* of the array is **score**. Array names follow the same conventions as other identifiers. The array **score** contains 10 *elements*. Any one of these elements may be referred to by giving the name of the array followed by the index of the particular element in square brackets ([]). The numbering of elements within an array starts with an index number of zero (0). For example, the first element in the score array is **score[0]**, the second element is **score[1]**, the third element is **score[2]**, and so on. In general, the *i*th element of array **x** is referred to as **x[i-1]**. The *value* of **score[0]** is 78, the value of **score[1]** is 98, and so on.

As with all variables, the value of each element within the array is undefined until you assign the element a value. The process of assigning values to elements in an array is called *populating the array*. To populate the array, specify the index number, and then assign the value to each element. Elements are not required to have an assigned value when the array is created. The syntax for populating an array is

```
arrayName[index] = value;
```

Any value that can be stored in a variable can be stored in an element in an array. Although array elements are often used to store strings or numbers, they can be used to store any valid data type.

The following sample code illustrates how to populate the array **score** that contains ten elements.

```
score[0] = 78;  
score[1] = 98;  
score[2] = 89;  
score[3] = 92;  
score[4] = 76;  
score[5] = 86;  
score[6] = 94;  
score[7] = 100;  
score[8] = 78;  
score[9] = 88;
```

---

### 7.3. Using Arrays

---

After the array is created and populated, the program can retrieve the value of any element in the array. To retrieve the value of an array element, use the array name and the index number of the element. The value from the array is then available to be used in an expression, displayed in a Web page, or placed into a variable. The script of Figure 20-2 illustrates an example of using arrays. This example declares two empty arrays (**score** and **grade**) and shows you how the arrays can grow dynamically to accommodate new elements. The function **populate\_Array()** allows the user to input the exam scores. Then the program calls the function **convert\_to\_grade()** to convert the scores into grades. Finally the function **output\_Array()** is invoked to display the contents of each array as HTML tables.

Every array in JavaScript knows its own length. The length of the array is determined by the expression:

```
arrayName.length
```

Note the expression `score.length` in the **for** structure condition to determine the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable *i* is less than 10. For a 10-element array, the index values are 0 through 9, so using the less than operator, `<`, guarantees that the loop does not attempt to access an element beyond the end of the array. Referring to an element outside the array bounds is normally a logic error. Figure 20-2 also shows you the first two user inputs and final report.

Notice that the function **output\_Array()** receives two arguments—a string to be output as the caption of the table and the array to output. To pass an array argument to a function, specify the name of the array (a reference to the array) without brackets.

#### Referring to Array Elements

You refer to an array's elements by using the element's ordinal number. For example, suppose you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

You then refer to the first element of the array as `myArray[0]` and the second element of the array as `myArray[1]`.

The index of the elements begins with zero (0), but the length of array (for example, `myArray.length`) reflects the number of elements in the array.

---

## 7.4. Array Methods

---

The Array object has the following methods:

<code>concat</code>	joins two arrays and returns a new array.
<code>join</code>	joins all elements of an array into a string.
<code>pop</code>	removes the last element from an array and returns that element.
<code>push</code>	adds one or more elements to the end of an array and returns that last element added.
<code>reverse</code>	transposes the elements of an array: the first array element becomes the last and the last becomes the first.
<code>shift</code>	removes the first element from an array and returns that element
<code>slice</code>	extracts a section of an array and returns a new array.
<code>splice</code>	adds and/or removes elements from an array.
<code>sort</code>	sorts the elements of an array.
<code>unshift</code>	adds one or more elements to the front of an array and returns the new length of the array.

For example, suppose you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

`myArray.join()` returns “Wind,Rain,Fire”; `myArray.reverse` transposes the array so that `myArray[0]` is “Fire”, `myArray[1]` is “Rain”, and `myArray[2]` is “Wind”. `myArray.sort` sorts the array so that `myArray[0]` is “Fire”, `myArray[1]` is “Rain”, and `myArray[2]` is “Wind”.

### Extra Commas in Array Literals

You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with spaces for the unspecified elements.

The following example creates the fish array:



```
fish = ["Lion", , "Angel"]
```

This array has two elements with values and one empty element (fish[0] is “Lion”, fish[1] is undefined, and fish[2] is “Angel”):

If you include a trailing comma at the end of the list of elements, the comma is ignored. In the following example, the length of the array is three. There is no myList[3]. All other commas in the list indicate a new element.

```
myList = ['home', , 'school', ];
```

In the following example, the length of the array is four, and myList[0] is missing.

```
myList = [ , 'home', , 'school'];
```

In the following example, the length of the array is four, and myList[3] is missing. Only the last comma is ignored. This trailing comma is optional.

```
myList = ['home', , 'school', , ];
```

## **Arrays and Regular Expressions**

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `regexp.exec`, `string.match`, and `string.replace`.

---

### **7.5. Deleting array elements**

---

When you delete an array element, the array length is not affected. For example, if you delete `a[3]`, `a[4]` is still `a[4]` and `a[3]` is undefined. When the delete operator removes an array element, that element is no longer in the array. In the following example, `trees[3]` is removed with delete.

```
trees=new Array("redwood","bay","cedar","oak","maple")
delete trees[3]
if (3 in trees) {
    // this does not get executed
```

If you want an array element to exist but have an undefined value, use the undefined keyword instead of the delete operator. In the following example, `trees[3]` is assigned the value undefined, but the array element still exists:

```
trees=new Array("redwood","bay","cedar","oak","maple")
trees[3]=undefined
if (3 in trees) {
  // this gets executed
}
```

JavaScript arrays are *dynamic* entities, in that they can change size after they are created. An array contains a set of data represented by a single variable name. Thus, an array is a group of memory locations that all have the same name and are normally of the same type (although this is not required). You can think of an array as a collection of variables contained within a single variable. To refer to a particular location or element in the array, we specify the name of the array and the *position number (index)* of the particular element in the array.

---

## 7.6. Passing Arrays to Functions

---

In many programming languages there are two ways to pass arguments to functions: *pass-by-value* (or *call-by-value*) and *pass-by-reference* (or *call-by-reference*). When you pass an argument by **value**, the computer creates a copy of the argument in memory, and only the copy is passed to the receiving function. Although the receiving function can change the contents of the copy, it cannot change the contents of the original argument.

If, on the other hand, you pass an argument by **reference**, the computer passes the argument's actual address in memory. Because the actual address is passed to the receiving function, the receiving function has the ability to directly access the argument's data, and thus the contents of the argument can be changed permanently by the receiving function.

In some programming language, arguments can be passed either by value or by reference. Programmers can control which way the arguments are passed. However, JavaScript does not allow the programmer to choose whether to pass each argument by value or by reference. In JavaScript, numbers and Boolean values are passed to functions by value, and arrays and all other objects are passed to functions by reference.

Although entire arrays are passed by using pass-by-reference, individual numeric and Boolean array element are passed by using pass-by-value exactly as simple numeric and Boolean variables are passed. Such simple single pieces of data are called *scalars*. To pass an array element to a function, use the name of the array followed by index number within brackets as an argument in the function call. Figure 20-3 uses an example to demonstrate the difference between passing an entire array and passing an array element. The array named **a** is declared and initialized in the statement

```
var a = [1, 2, 3, 4, 5];
```

If an array/s element values are known in advance, the elements of the array can be declared and initialized in this way. A comma-separated *initializer* list is enclosed in square brackets ([]). Note that the preceding declaration does not require the keyword **new** to create the array object. You also can use the statement

```
var a = new Array(1, 2, 3, 4, 5);
```

to declare and initialize array **a**. In this case, the initial values of the array elements are specified as arguments in the parentheses following **new Array**.

In the example first passes the array **a** to function **modify\_array()**. Function **modify\_array()** multiplies each element of array **a** by 2. To illustrate that the elements of array **a** were modified, function **output\_array()** is used to display the contents of array **a** after it is modified. Then the program passes the forth element **a[3]** of array **a** to function **modify\_element()**. Since **a[3]** element is actually one integer in the array **a**, a copy of **a[3]** is passed. Function **modify\_element()** multiplies its argument by 2 and stores the result in its parameter **e**. Since the parameter of function **modify\_element()** is a local variable in that function, the local variable is destroyed when the function terminates. The element **a[3]** still holds unmodified value.

In function **output\_array()**, the statement

```
document.write(header + theArray.join(", ") + "<BR>");
```

uses **Array** method **join** to create a string containing all the elements in **theArray**. Method **join** takes as its argument a string containing the separator

that should be used to separate the elements of the array in the string that is returned.

The statement

```
for (var i in theArray)
    theArray[i] *= 2;
```

shows the syntax of a **for/in** structure. Inside the parentheses, we declare the variable *i* that will be used to select each element in the object (**theArray**, in this case) to the right of keyword **in**. In the **for/in** structure, JavaScript automatically determines the number of elements in the object. In the case of an **Array** object (**theArray**, in this case), the value assigned to variable *i* is a subscript in the range from 0 up to (but not including) **theArray.length**.

The arguments of a function are maintained in an array. Within a function, you can address the parameters passed to it as follows:

```
arguments[i]
functionName.arguments[i]
```

where *i* is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

Using the arguments array, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then treat each argument using the arguments array. For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
    result="" // initialize list
    // iterate through arguments
    for (var i=1; i<arguments.length; i++) {
        result += arguments[i] + separator
```

```

    }
    return result
}

```

You can pass any number of arguments to this function, and it creates a list

using each argument as an item in the list.

```

// returns "red, orange, blue, "
myConcat(" ", "red", "orange", "blue")

// returns "elephant; giraffe; lion; cheetah;"
myConcat("; ", "elephant", "giraffe", "lion", "cheetah")

// returns "sage. basil. oregano. pepper. parsley. "
myConcat(". ", "sage", "basil", "oregano", "pepper", "parsley")

```

---

## 7.7. Sorting Arrays

---

*Sorting* data (i.e. arranging data in alphabetical, numerical, or chronological order) is one of the most important computing applications. Virtually every organization must sort some data. A **sort** is an algorithm for ordering an array. Sorting data is an intriguing problem that has attracted intense research efforts in the field of computer science. This section introduces the simplest known sorting algorithm: the **bubble sort** (also called the **sinking sort**). Smaller values gradually ?bubble? their way to the top of the array like air bubbles rising in water and larger values ?sink? to the bottom of the array. This technique compares adjacent items and swaps those that are out of order. If this process is repeated enough times, the list will be ordered. Bubble sort is to make several passes through the array. On each pass, successive pairs of elements are compared. Sorting an array requires a pair of nested loops. The inner loop performs a single pass and the outer loop controls the number of passes.

The script creates the function **bubble\_sort()** to perform the bubble-sort task. In this example, an array of string is passed to the function **bubble\_sort()** through the argument **theArray**. The array is automatically passed by call-by-reference.

First the **bubble\_sort()** function compares theArray[0] to theArray[1], theArray[1] to theArray[2], then theArray[2] to theArray[3], and so on until it completes the pass by comparing the last two elements of **theArray**. On the first pass, the largest value is guaranteed to sink to the bottom element of the array. Therefore, the second pass does not have to consider it and so requires one less comparison. At the end of the second pass, the last two items will be in their proper position. Thus, each successive pass requires one less comparison.

This example uses the event **ONLOAD**, which calls an event handler when the <BODY>; of the HTML document is completely loaded into the browser window. In this example, the function **start()** is called by the browser as the event handler for the <BODY>'s **ONLOAD** event.

---

## 7.8. Searching Arrays

---

A common operation on arrays is to search the elements of an array for a particular data item. **Searching** is the process of finding a particular element of an array. One approach would be to start with the first element of the array and compares each element to the array with the **search key** (the value you want to find) until a match was found. This searching algorithm is called a **linear search** (or **sequential search**). The sequential search works well for unsorted arrays. If the array is sorted, the high-speed binary search algorithm can be used.

The **binary search** algorithm removes from consideration one half of the elements in the array being searched after each comparison. The algorithm locates the middle element of the array and compares it to the search key to determine in which half of the array the search key lies. The other half is then discarded and the retained half is temporarily regarded as the entire array. The process is repeated until the item is found.

The function **binary\_search()** locates the middle element of **theArray** and compare it to the searchKey. If they are equal, the searchKey is found and the array index of that element is returned. If the searchKey does not match the middle element of the array, the low index or high index is adjusted so that a smaller sub-array can be searched. If the searchKey is less than the middle element, the high index is set to middle - 1, and the search is continued on the

elements from low to middle - 1. If the searchKey is greater than the middle element, the low index is set to middle + 1, and the search is continued on the elements from middle + 1 to high.

The expression `searchForm.inputVal.value` specifies the **value** property of the text field **inputVal**. The **value** property specifies the text to display in the text field. To access this property, we specify the name of the form (**searchForm**) that contains the text field followed by a *dot operator* (.) followed by the name of the text field we would like to manipulate. The dot operator is also known as the *field access operator* or the *member access operator*. In the preceding expression, the dot operator is used to access the **inputVal** member of the **searchForm** form. Similarly, the second member access operator is used to access the **value** member (or property) of the **inputVal** text field.

---

## 7.9. Multiple-Dimensional Arrays

---

Each array discussed so far held a single list of items. Such arrays are called one-dimensional arrays or single-subscripted array, that is, only one subscript is needed to locate an element in an array. In some business applications, there is a need for multiple-dimensional arrays, where two or more subscripts are required to locate an element in an array. Multiple-dimensional arrays with two subscripts are called two-dimensional arrays and often used to represent tables of values consisting of information arranged in rows and columns.

To store the contents of tables, a two-dimensional array uses two subscripts, each with its own range. The range of the first subscript is determined by the number of rows in the table, and the range of the second subscript is determined by the number of columns. JavaScript does not support multiple-dimensional arrays directly, but allows the programmer to specify one-dimensional arrays whose elements are also one-dimensional array, thus achieving the same effect.

	Column0	Column1	Column2	Column3
Row 0	<b>a[0][0]</b>	<b>a[0][1]</b>	<b>a[0][2]</b>	<b>a[0][3]</b>
Row 1	<b>a[1][0]</b>	<b>a[1][1]</b>	<b>a[1][2]</b>	<b>a[1][3]</b>
Row 2	<b>a[2][0]</b>	<b>a[2][1]</b>	<b>a[2][2]</b>	<b>a[2][3]</b>

**Fig. : Two Dimensional Array**

Above Figure illustrates a two-dimensional array, **a**, with three rows and four columns (a 3-by-4 array). Every element in array **a** is identified by an element name of the form **a[i][j]**; **a** is the name of the array and **i** and **j** are the indexes that uniquely identify the row and column of each element in **a**. For example, to declare and initialize an array name **mileageArray** for the preceding road mileage example, we may use the following statement:

```
var mileageArray = [ [0, 2054, 802, 738], [2054, 0, 2786, 2706],
                    [802, 2786, 0, 100], [738, 2706, 100, 0] ];
```

The values are grouped by row in square in square brackets. So, [0, 2054, 802, 738] initialize the first row; that is, mileage[0][0], mileage[0][1], mileage[0][2], and mileage[0][3]. And [2054, 0, 2786, 2706] initialize the second row, and so on. Thus, two-dimensional array are maintained as arrays of arrays. It is also possible to create a two-dimensional array in which each row has a different number of columns. For example, the declaration

```
var a = [ [1, 2], [3, 4, 5, 6] ];
```

created and initialized array **a** with row 0 containing two elements and row 1 containing four elements. A two-dimensional array can also be declared as follows:

```
var a;
a = new Array(2);    // declare rows
a[0] = new Array(2); // declare columns in row 0
a[1] = new Array(4); // declare columns in row 1
```



The preceding statements create a two-dimensional array with two rows. Row 0 contains 2 columns and row 1 contains four columns.

Since a two-dimensional array is organized in columns within row order, it is common to use a nested **for** structure to handle the array.

---

## 7.10 Let us Sum Up

---

JavaScript does not have an explicit array data type. However, you can use the predefined Array object and its methods to work with arrays in your applications. The Array object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them.

An array in JavaScript is an **Array** object. The process of creating new objects is known as *creating an instance*. All objects are created in JavaScript using the **new** keyword.

You refer to an array's elements by using the element's ordinal number. The Array object has the following methods:

concat	joins two arrays and returns a new array.
join	joins all elements of an array into a string.
pop	removes the last element from an array and returns that element.
push	adds one or more elements to the end of an array and returns that last element added.
reverse	transposes the elements of an array: the first array element becomes the last and the last becomes the first.
shift	removes the first element from an array and returns that element
slice	extracts a section of an array and returns a new array.
splice	adds and/or removes elements from an array.
sort	sorts the elements of an array.
unshift	adds one or more elements to the front of an array and returns the new length of the array.

You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with spaces for the unspecified elements.

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `regexp.exec`, `string.match`, and `string.replace`.

When you delete an array element, the array length is not affected.

The arguments of a function are maintained in an array. Within a function, you can address the parameters passed to it as follows:

```
arguments[i]
```

```
functionName.arguments[i]
```

where *i* is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

*Sorting* data (i.e. arranging data in alphabetical, numerical, or chronological order) is one of the most important computing applications. Virtually every organization must sort some data. A sort is an algorithm for ordering an array. Sorting data is an intriguing problem that has attracted intense research efforts in the field of computer science. This section introduces the simplest known sorting algorithm: the bubble sort (also called the sinking sort).

A common operation on arrays is to search the elements of an array for a particular data item. **Searching** is the process of finding a particular element of an array. One approach would be to start with the first element of the array and compares each element to the array with the **search key** (the value you want to find) until a match was found. This searching algorithm is called a **linear search** (or **sequential search**). The sequential search works well for unsorted arrays. If the array is sorted, the high-speed binary search algorithm can be used.

The function **binary\_search()** locates the middle element of **theArray** and compare it to the `searchKey`. If they are equal, the `searchKey` is found and the array index of that element is returned. If the `searchKey` does not match the middle element of the array, the low index or high index is adjusted so that a smaller sub-array can be searched. If the `searchKey` is less than the middle

element, the high index is set to middle - 1, and the search is continued on the elements from low to middle - 1. If the searchKey is greater than the middle element, the low index is set to middle + 1, and the search is continued on the elements from middle + 1 to high.

Each array discussed so far held a single list of items. Such arrays are called one-dimensional arrays or single-subscripted array, that is, only one subscript is needed to locate an element in an array. In some business applications, there is a need for multiple-dimensional arrays, where two or more subscripts are required to locate an element in an array. Multiple-dimensional arrays with two subscripts are called two-dimensional arrays and often used to represent tables of values consisting of information arranged in rows and columns.

To store the contents of tables, a two-dimensional array uses two subscripts, each with its own range. The range of the first subscript is determined by the number of rows in the table, and the range of the second subscript is determined by the number of columns. JavaScript does not support multiple-dimensional arrays directly, but allows the programmer to specify one-dimensional arrays whose elements are also one-dimensional array, thus achieving the same effect.

---

### **7.11. Lesson end Activities**

---

1. Describe JavaScript Array.
2. How JavaScript implement two dimensional array.

---

### **7.12 Check your Progress**

---

1. Write a JavaScript array program implement array multiplication.
2. Write a JavaScript program to implement binary search.

---

### **7.13. Reference**

---

1. [WWW.W3C](http://WWW.W3C)
2. Thomas A. Powell, "The Complete Reference HTML and XHTML", fourth Edition, Tata McGraw Hill.
3. Deitel, Deitel, Nieto, "Internet and World Wide Web – How to Program", Pearson Education Asia, 2003



## JavaScript – Objects

---

### Contents

#### 8.0. Aim and Objective

#### 8.1. Introduction

#### 8.2. Objects and Properties

#### 8.3. Using Object Initialization

#### 8.4. Using a Construction function

#### 8.5. Indexing Objects

#### 8.6. Defining properties for an Object type

#### 8.7. Using this for Object reference

#### 8.8. Predefined core objects

##### 8.8.1. Boolean Object

##### 8.8.2. Date Object

##### 8.8.3. Function Object

##### 8.8.4. Math Object

##### 8.8.5. Number Object

##### 8.8.6. String Object

#### 8.9. Object manipulation statements

#### 8.10. Let us Sum Up

#### 8.11. Lesson end Activities

#### 8.12. Check Your progress

#### 8.13. Reference

---

#### 8.0. Aim and Objective

---

- To understand how to use objects, properties, functions, and methods,
- To understand how to create your own objects.

---

## 8.1. Introduction

---

JavaScript is designed on a simple object-based paradigm. An object is a construct with properties that are JavaScript variables or other objects. An object also has functions associated with it that are known as the object's *methods*. In addition to objects that are predefined in the Navigator client and the server, you can define your own objects.

---

## 8.2. Objects and Properties

---

A JavaScript object has properties associated with it. You access the properties of an object with a simple notation:

*objectName.propertyName*

Both the object name and property name are case sensitive. You define a property by assigning it a value. For example, suppose there is an object named `myCar` (for now, just assume the object already exists). You can give it properties named `make`, `model`, and `year` as follows:

```
myCar.make = "Ford"
myCar.model = "Mustang"
myCar.year = 1969;
```

An array is an ordered set of values associated with a single variable name. Properties and arrays in JavaScript are intimately related; in fact, they are different interfaces to the same data structure. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar["make"] = "Ford"
myCar["model"] = "Mustang"
myCar["year"] = 1967
```

This type of array is known as an *associative array*, because each index element is also associated with a string value. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function show_props(obj, obj_name) {
  var result = ""
```

```

for (var i in obj)
result += obj_name + "." + i + " = " + obj[i] + "\n"
return result
}

```

So, the function call `show_props(myCar, "myCar")` would return the following:

```

myCar.make = Ford
myCar.model = Mustang
myCar.year = 1967

```

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`). You should not use an object literal at the beginning of a statement. This will lead to an error.

The following is an example of an object literal. The first element of the car object defines a property, `myCar`; the second element, the `getCar` property, invokes a function (`Cars("honda")`); the third element, the special property, uses an existing variable (`Sales`).

```

var Sales = "Toyota";

function CarTypes(name) {
if(name == "Honda")
return name;
else
return "Sorry, we don't sell " + name + ".";
}

car = {myCar: "Saturn", getCar: CarTypes("Honda"), special: Sales}

document.write(car.myCar); // Saturn
document.write(car.getCar); // Honda
document.write(car.special); // Toyota

```

---

### 8.3. Using Object Initializers

---

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. “Object initializer” is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
objectName = {property1:value1, property2:value2,..., propertyN:valueN}
```

where `objectName` is the name of the new object, each `propertyI` is an identifier (either a name, a number, or a string literal), and each `valueI` is an expression whose value is assigned to the `propertyI`. The `objectName` and assignment is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable.

If an object is created with an object initializer in a top-level script, JavaScript interprets the object each time it evaluates the expression containing the object literal. In addition, an initializer used in a function is created each time the function is called.

The following statement creates an object and assigns it to the variable `x` if and only if the expression `cond` is true.

```
if (cond) x = {hi:"there"}
```

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

---

### 8.4. Using a Constructor Function

---

You can also create an object with these two steps:

1. Define the object type by writing a constructor function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `car`, and



you want it to have properties for make, model, year, and color. To do this, you would write the following function:

```
function car(make, model, year) {  
  this.make = make  
  this.model = model  
  this.year = year  
}
```

Notice the use of this to assign values to the object's properties based on the values passed to the function.

Now you can create an object called mycar as follows:

```
mycar = new car("Eagle", "Talon TSi", 1993)
```

This statement creates mycar and assigns it the specified values for its properties. Then the value of mycar.make is the string "Eagle", mycar.year is the integer 1993, and so on. You can create any number of car objects by calls to new. For example, kenscar = new car("Nissan", "300ZX", 1992)

```
vpgscar = new car("Mazda", "Miata", 1990)
```

An object can have a property that is itself another object. For example, suppose you define an object called person as follows:

```
function person(name, age, sex) {  
  this.name = name  
  this.age = age  
  this.sex = sex  
}
```

and then instantiate two new person objects as follows:

```
rand = new person("Rand McKinnon", 33, "M")  
ken = new person("Ken Jones", 39, "M")
```

Then you can rewrite the definition of car to include an owner property that takes a person object, as follows:

```
function car(make, model, year, owner) {  
  this.make = make
```

```
this.model = model  
this.year = year  
this.owner = owner  
}
```

To instantiate the new objects, you then use the following:

```
car1 = new car("Eagle", "Talon TSi", 1993, rand)  
car2 = new car("Nissan", "300ZX", 1992, ken)
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the arguments for the owners. Then if you want to find out the name of the owner of `car2`, you can access the following property:

```
car2.owner.name
```

Note that you can always add a property to a previously defined object. For example, the statement

```
car1.color = "black"
```

adds a property `color` to `car1`, and assigns it a value of "black." However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the `car` object type.

---

## 8.5. Indexing Object Properties

---

In JavaScript you can initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index.

This applies when you create an object and its properties with a constructor function, as in the above example of the `Car` object type, and when you define individual properties explicitly (for example, `myCar.color = "red"`). So if you define object properties initially with an index, such as `myCar[5] = "25 mpg"`, you can subsequently refer to the property as `myCar[5]`.

The exception to this rule is objects reflected from HTML, such as the `forms` array. You can always refer to objects in these arrays by either their ordinal number (based on where they appear in the document) or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME`

attribute of “myForm”, you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

---

## 8.6. Defining Properties for an Object Type

---

You can add a property to a previously defined object type by using the prototype property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a color property to all objects of type `car`, and then assigns a value to the color property of the object `car1`.

```
Car.prototype.color=null
```

```
car1.color="black"
```

### Defining Methods

A *method* is a function associated with an object. You define a method the same way you define a standard function. Then you use the following syntax to associate the function with an existing object:

```
object.methodname = function_name
```

where `object` is an existing object, `methodname` is the name you are assigning to the method, and `function_name` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined `car` objects; for example,

```
function displayCar() {  
    var result = "A Beautiful " + this.year + " " + this.make  
    + " " + this.model  
    pretty_print(result)  
}
```

where `pretty_print` is function to display a horizontal rule and a string.

Notice the use of `this` to refer to the object to which the method belongs. You can make this function a method of `car` by adding the statement `this.displayCar = displayCar;` to the object definition. So, the full definition of `car` would now look like

```
function car(make, model, year, owner) {  
  this.make = make  
  this.model = model  
  this.year = year  
  this.owner = owner  
  this.displayCar = displayCar  
}
```

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar()  
car2.displayCar()
```

---

### 8.7. Using `this` for Object References

---

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have a function called `validate` that validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {  
  if ((obj.value < lowval) || (obj.value > hival))  
    alert("Invalid Value!")  
}
```

Then, you could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<INPUT TYPE="text" NAME="age" SIZE=3  
  onChange="validate(this, 18, 99)">
```

In general, `this` refers to the calling object in a method. When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When

the user clicks the button, the value of the Text object is set to the form's name. The button's onClick event handler uses this.form to refer to the parent form, myForm.

```
<FORM NAME="myForm">  
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">  
<P>  
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"  
onClick="this.form.text1.value=this.form.name">  
</FORM>
```

## Deleting Objects

You can remove an object by using the delete operator. The following code shows how to remove an object.

```
myobj=new Number()  
delete myobj // removes the object and returns true
```

---

## 8.8. Predefined Core Objects

### 8.8.1 Boolean Object

---

The Boolean object is a wrapper around the primitive Boolean data type. Use the following syntax to create a Boolean object:

```
booleanObjectName = new Boolean(value)
```

Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object. Any object whose value is not undefined or null, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. See “if...else Statement” on page 80 for more information.

---

### 8.8.2. Date Object

---

JavaScript does not have a date data type. However, you can use the Date object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties. JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and

both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00. The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

To create a Date object:

```
dateObjectName = new Date([parameters])
```

where dateObjectName is the name of the Date object being created; it can be a new object or a property of an existing object.

The parameters in the preceding syntax can be any of the following:

```
today = new Date().
```

A string representing a date in the following form: “Month day, year hours:minutes:seconds.” For example, Xmas95 = new Date("December 25, 1995 13:30:00"). If you omit hours, minutes, or seconds, the value will be set to zero.

A set of integer values for year, month, and day. For example, Xmas95 = new Date(1995,11,25). A set of values for year, month, day, hour, minute, and seconds. For example, Xmas95 = new Date(1995,11,25,9,30,0).

### **Methods of the Date Object**

The Date object methods for handling dates and times fall into these broad categories: “set” methods, for setting date and time values in Date objects. “get” methods, for getting date and time values from Date objects. “to” methods, for returning string values from Date objects. parse and UTC methods, for parsing Date strings.

With the “get” and “set” methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a getDay method that returns the day of the week, but no corresponding setDay method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

Seconds and minutes: 0 to 59

Hours: 0 to 23

Day: 0 (Sunday) to 6 (Saturday)

Date: 1 to 31 (day of the month)

Months: 0 (January) to 11 (December)

Year: years since 1900

For example, suppose you define the following date:

```
Xmas95 = new Date("December 25, 1995")
```

Then `Xmas95.getMonth()` returns 11, and `Xmas95.getFullYear()` returns 95. The `getTime` and `setTime` methods are useful for comparing dates. The `getTime` method returns the number of milliseconds since January 1, 1970, 00:00:00 for a `Date` object.

For example, the following code displays the number of days left in the current year:

```
today = new Date()
endYear = new Date(1995,11,31,23,59,59,999) // Set day and month
endYear.setFullYear(today.getFullYear()) // Set year to this year
msPerDay = 24 * 60 * 60 * 1000 // Number of milliseconds per day
daysLeft = (endYear.getTime() - today.getTime()) / msPerDay
daysLeft = Math.round(daysLeft) //returns days left in the year
```

This example creates a `Date` object named `today` that contains today's date. It then creates a `Date` object named `endYear` and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between `today` and `endYear`, using `getTime` and rounding to a whole number of days.

The `parse` method is useful for assigning values from date strings to existing `Date` objects. For example, the following code uses `parse` and `setTime` to assign a date value to the `IPOdate` object:

```
IPOdate = new Date()
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

### **Using the Date Object: an Example**

In the following example, the function `JSClock()` returns the time in the format of a digital clock.

```
function JSClock() {
    var time = new Date()
    var hour = time.getHours()
    var minute = time.getMinutes()
    var second = time.getSeconds()
    var temp = "" + ((hour > 12) ? hour - 12 : hour)
    temp += ((minute < 10) ? ":0" : ":") + minute
    temp += ((second < 10) ? ":0" : ":") + second
    temp += (hour >= 12) ? " P.M." : " A.M."
    return temp
}
```

The JSClock function first creates a new Date object called time; since no arguments are given, time is created with the current date and time. Then calls to the getHours, getMinutes, and getSeconds methods assign the value of the current hour, minute and seconds to hour, minute, and second. The next four statements build a string value based on the time. The first statement creates a variable temp, assigning it a value using a conditional expression; if hour is greater than 12, (hour - 13), otherwise simply hour. The next statement appends a minute value to temp. If the value of minute is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to temp in the same way. Finally, a conditional expression appends “PM” to temp if hour is 12 or greater; otherwise, it appends “AM” to temp.

---

### 8.8.3. Function Object

---

The predefined Function object specifies a string of JavaScript code to be compiled as a function.

To create a Function object:

```
functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)
```



functionObjectName is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as window.onerror. arg1, arg2, ... argn are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm". functionBody is a string specifying the JavaScript code to be compiled as the function body.

Function objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the function statement. See the *Client-Side JavaScript Reference* for more information.

The following code assigns a function to the variable setBGColor. This function sets the current document's background color.

```
var setBGColor = new Function("document.bgColor='antiquewhite'")
```

To call the Function object, you can specify the variable name as if it were a function. The following code executes the function specified by the setBGColor variable:

```
var colorChoice="antiquewhite"

if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

1. document.form1.colorButton.onclick=setBGColor
2. <INPUT NAME="colorButton" TYPE="button"  
VALUE="Change background color"  
onClick="setBGColor()">

Creating the variable setBGColor shown above is similar to declaring the following function:

```
function setBGColor() {
    document.bgColor='antiquewhite'
}
```

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

The inner function can be accessed only from statements in the outer function. The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

---

#### 8.8.4. Math Object

---

The predefined Math object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141...), which you would use in an application as Math.PI

Similarly, standard mathematical functions are methods of Math. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write Math.sin(1.56)

Note that all trigonometric methods of Math take arguments in radians. The following table summarizes the Math object's methods. Unlike many other objects, you never create a Math object of your own. You always use the predefined Math object.

**Table 8.1 Methods of Math**

Method	Description
abs	Absolute value
sin, cos, tan	Standard trigonometric functions; argument in radians
acos, asin, atan	Inverse trigonometric functions; return values in radians
exp, log	Exponential and natural logarithm, base e
ceil	Returns least integer greater than or equal to argument
floor	Returns greatest integer less than or equal to argument
min, max	Returns greater or lesser (respectively) of two arguments
pow	Exponential; first argument is base, second is exponent
round	Rounds argument to nearest integer
sqrt	Square root

It is often convenient to use the `with` statement when a section of code uses several math constants and methods, so you don't have to type "Math" repeatedly. For example,

```
with (Math) {  
  a = PI * r*r  
  y = r*sin(theta)  
  x = r*cos(theta)  
}
```

---

### 8.8.5. Number Object

---

The `Number` object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
biggestNum = Number.MAX_VALUE  
smallestNum = Number.MIN_VALUE  
infiniteNum = Number.POSITIVE_INFINITY  
negInfiniteNum = Number.NEGATIVE_INFINITY  
notANum = Number.NaN
```

You always refer to a property of the predefined `Number` object as shown above, and not as a property of a `Number` object you create yourself. The following table summarizes the `Number` object's properties.

---

### 8.8.6. String Object

---

The `String` object is a wrapper around the string primitive data type. Do not confuse a string literal with the `String` object. For example, the following code creates the string literal `s1` and also the `String` object `s2`:

```
s1 = "foo" //creates a string literal value  
s2 = new String("foo") //creates a String object
```

You can call any of the methods of the `String` object on a string literal value—JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `String.length` property with a string literal. You should use string

literals unless you specifically need to use a String object, because String objects can have counterintuitive behavior. For example:

```
s1 = "2 + 2" //creates a string literal value
s2 = new String("2 + 2")//creates a String object
eval(s1) //returns the number 4
eval(s2) //returns the string "2 + 2"
```

A String object has one property, `length`, that indicates the number of characters in the string. For example, the following code assigns `x` the value 13, because “Hello, World!” has 13 characters:

```
myString = "Hello, World!"
x = mystring.length
```

A String object has two types of methods: those that return a variation on the string itself, such as `substring` and `toUpperCase`, and those that return an HTML-formatted version of the string, such as `bold` and `link`.

For example, using the previous example, both `mystring.toUpperCase()` and `"hello, world!".toUpperCase()` return the string “HELLO, WORLD!”.

The `substring` method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, `mystring.substring(4, 9)` returns the string “o, Wo.” See the `substring` method of the String object in the *Client-Side JavaScript Reference* for more information.

The String object also has a number of methods for automatic HTML formatting, such as `bold` to create boldface text and `link` to create a hyperlink.

For example, you could create a hyperlink to a hypothetical URL with the `link` method as follows:

```
mystring.link("http://www.helloworld.com")
```

The following table summarizes the methods of String objects.

**Table 7.2 Methods of String**

<b>Method</b>	<b>Description</b>
anchor	Creates HTML named anchor
big, blink, bold, fixed, italics, small, strike, sub, sup	Creates HTML formatted string
charAt, charCodeAt	Returns the character or character code at the specified position in string
indexOf, lastIndexOf	Returns the position of specified substring in the string or last position of specified substring, respectively
link	Creates HTML hyperlink
concat	Combines the text of two strings and returns a new string
fromCharCode	Constructs a string from the specified sequence of ISO-Latin-1 codeset values
split	Splits a String object into an array of strings by separating the string into substrings
slice	Extracts a section of an string and returns a new string.
substring, substr	Returns the specified subset of the string, either by specifying the start and end indexes or the start index and a length
match, replace, toLowerCase,	search Used to work with regular expressions
toUpperCase	Returns the string in all lowercase or all uppercase, Respectively

### **8.9. Object Manipulation Statements**

JavaScript uses the for...in and with statements to manipulate objects.

#### **for...in Statement**

The `for...in` statement iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

```
for (variable in object) {  
    statements }
```

**Example.** The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function dump_props(obj, obj_name) {  
    var result = ""  
    for (var i in obj) {  
        result += obj_name + "." + i + " = " + obj[i] + "<BR>"  
    }  
    result += "<HR>"  
    return result  
}
```

For an object `car` with properties `make` and `model`, result would be:

```
car.make = Ford  
car.model = Mustang
```

### **with Statement**

The `with` statement establishes the default object for a set of statements. JavaScript looks up any unqualified names within the set of statements to determine if the names are properties of the default object. If an unqualified name matches a property, then the property is used in the statement; otherwise, a local or global variable is used.

A `with` statement looks as follows:

```
with (object) {  
    statements  
}
```

**Example.** The following with statement specifies that the Math object is the default object. The statements following the with statement refer to the PI property and the cos and sin methods, without specifying an object. JavaScript assumes the Math object for these references.

```
var a, x, y
var r=10
with (Math) {
    a = PI * r * r
    x = r * cos(PI)
    y = r * sin(PI/2)
}
```

---

### 8.10. Let us Sum Up

---

A JavaScript object has properties associated with it. You access the properties of an object with a simple notation:

*objectName.propertyName*

Both the object name and property name are case sensitive. An array is an ordered set of values associated with a single variable name. Properties and arrays in JavaScript are intimately related; in fact, they are different interfaces to the same data structure.

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation.

To define an object type, create a function for the object type that specifies its name, properties, and methods.

You can add a property to a previously defined object type by using the prototype property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object.

A *method* is a function associated with an object. You define a method the same way you define a standard function. Then you use the following syntax to associate the function with an existing object:

```
object.methodname = function_name
```

where object is an existing object, methodname is the name you are assigning to the method, and function\_name is the name of the function.

JavaScript has a special keyword, this, that you can use within a method to refer to the current object

### **Boolean Object**

The Boolean object is a wrapper around the primitive Boolean data type. Use the following syntax to create a Boolean object:

```
booleanObjectName = new Boolean(value)
```

### **Date Object**

JavaScript does not have a date data type. However, you can use the Date object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties. JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00. The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

To create a Date object:

```
dateObjectName = new Date([parameters])
```

where dateObjectName is the name of the Date object being created; it can be a new object or a property of an existing object.

### **Function Object**

The predefined Function object specifies a string of JavaScript code to be compiled as a function.

To create a Function object:

```
functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)
```

functionObjectName is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as window.onerror. arg1, arg2, ... argn are arguments to be used by the function as formal argument names.



## Math Object

The predefined Math object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141...), which you would use in an application as Math.PI

## Number Object

The Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
biggestNum = Number.MAX_VALUE  
smallestNum = Number.MIN_VALUE  
infiniteNum = Number.POSITIVE_INFINITY  
negInfiniteNum = Number.NEGATIVE_INFINITY  
notANum = Number.NaN
```

You always refer to a property of the predefined Number object as shown above, and not as a property of a Number object you create yourself. The following table summarizes the Number object's properties.

## String Object

The String object is a wrapper around the string primitive data type. Do not confuse a string literal with the String object. For example, the following code creates the string literal s1 and also the String object s2:

```
s1 = "foo" //creates a string literal value  
s2 = new String("foo") //creates a String object
```

## Object Manipulation Statements

### for...in Statement

The for...in statement iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. A for...in statement looks as follows:

```
for (variable in object) {  
    statements }
```

## **with Statement**

The with statement establishes the default object for a set of statements. JavaScript looks up any unqualified names within the set of statements to determine if the names are properties of the default object. If an unqualified name matches a property, then the property is used in the statement; otherwise, a local or global variable is used.

A with statement looks as follows:

```
with (object){  
    statements  
}
```

---

### **8.11. Lesson end Activities**

---

What is JavaScript Object?

1. How to create and delete an Object?
2. What is the purpose of JavaScript Constructor?

---

### **8.12 Check Your Progress**

---

1. Write a JavaScript program to implement for each and with statements.
2. Write short notes on JavaScript predefined objects.

---

### **8.13. Reference**

---

1. [WWW.W3C](http://www.w3c.org)
2. Thomas A. Powell, "The Complete Reference HTML and XHTML", fourth Edition, Tata McGraw Hill.
3. Deitel, Deitel, Nieto, "Internet and World Wide Web – How to Program", Pearson Education Asia, 2003

## Cascading Style Sheet (CSS)

---

### Contents

#### 9.0. Aim and Objective

#### 9.1.Introduction

#### 9.2. Structure vs Presentation

#### 9.3. Inline Style

#### 9.4. Creating Style Sheet with the Style Element

#### 9.5. Conflicting Style

#### 9.6. linking External Style Sheets

#### 9.7. Positioning Elements

#### 9.8. Background

#### 9.9. Element Dimension

#### 9.10. Text Flow and the Box Model

#### 9.11. User Style sheet

#### 9.12. Let us Sum Up

#### 9.13. Lesson end Activities

#### 9.14. Check your progress

#### 9.15 .Reference

---

### 9.0. Aim and Objective

---

- To take control of the appearance of a web site by creating style sheets
- To use stylesheets to separate presentation from the content

---

### 9.1. Introduction

---

HTML has been primarily concerned with representing the structure of documents online. By this, I mean that it allows the author to identify headings, paragraphs, lists, etc., but it does not provide (very many) facilities for specifying how the information can be presented (fonts, colors, spacing, text flow, etc.).

In the early days, this was fine; the Web was used mostly by technical people to publish articles and documents without much emphasis on how the documents looked. With the explosion of interest in the Web, both the audience and the authors have changed. Many people writing and designing for the Web want to exert much more control over the presentation of their documents. This brings the issue of structure versus presentation into sharp relief.

---

## 9.2. Structure vs Presentation

---

### Structural

This is an article in a journal. It has a title, an author, and an abstract. The body of the article is divided into sections. Each section has a title and may include subsections. Most of the article is comprised of paragraphs, but lists, figures, and other elements are interspersed. The most fundamental question in structural markup is, "what is it?" What structural significance does it have? Is it a filename, or a paragraph, or a list item? Is it a chapter title, or a person, place, or thing?

### Presentation

This is a typeset page. The page begins with a centered title in 18 point ITC Garamond small caps. The author's name appears below the title in italics, also centered. Following the author's name is 40 points of white space followed by the abstract, also set in italics and with the bold, centered title "Abstract." The body of the article appears in two columns below the abstract. It is introduced by a heading in 15 point Franklin Gothic Book Compressed. Most of the article is comprised of paragraphs set in 9/12 ITC Garamond Light. The main text wraps around other elements that appear interspersed in the text. The most fundamental question in presentation markup is, "what does it look like?" Is it green, or bold, or does it blink? Does it move, is it in a box, does it stand out, or is it hard to see?

Both of these answers are correct and useful.

The structural view of a document is useful because it provides us with context. Using the structural view, you can answer questions like "where is the

section on font styles?" or build a table of contents with first- and second-level section headings, or identify the list of authors in a journal.

The presentation view is useful because we have expectations about how information will be presented. We expect books, journals, marketing information, advertisements, annual reports, and technical bulletins to *look different* (even when they have similar structure). In addition, many institutions have a distinctive look and feel that they expect to appear in all their published documents.

### **HTML is the Structure**

The problem is that HTML, the primary markup language used to code documents on the World Wide Web, is really only useful for expressing the structure of a document.

It is possible to exert some control over the presentation, by employing tables and a variety of other tricks, but doing so blurs the structural view of the document. Adding new presentational tags to HTML isn't going to help, either. Presentational tags further blur the underlying structure of the document and could lead rapidly to multiple, incompatible HTML variants.

### **Style Sheets are the Presentation**

Style sheets, which should become commonplace on the Web over the next few months, provide a means of associating presentational information with the structural elements of a document in a way that does not corrupt the underlying structure of the document.

A style sheet is a set of guidelines for the browser indicating how the various elements of a document should be presented. For example, the following set of instructions constitute a style sheet for web documents:

- The document background should be blue.
- Top-level headings should be in 20 point Bold Arial (or Helvetica, or at least a sans-serif face).
- Body text should be 10 point Times Roman. Body text should be white; links should be light red; visited links should be yellow.
- Block quotations should be set in 8 point Times Italic. The body text should be black and the background white.

- Warnings should be indented on both sides and set in yellow.
- Itemized lists should use a fancy bullet.

---

### 9.3.Inline Style

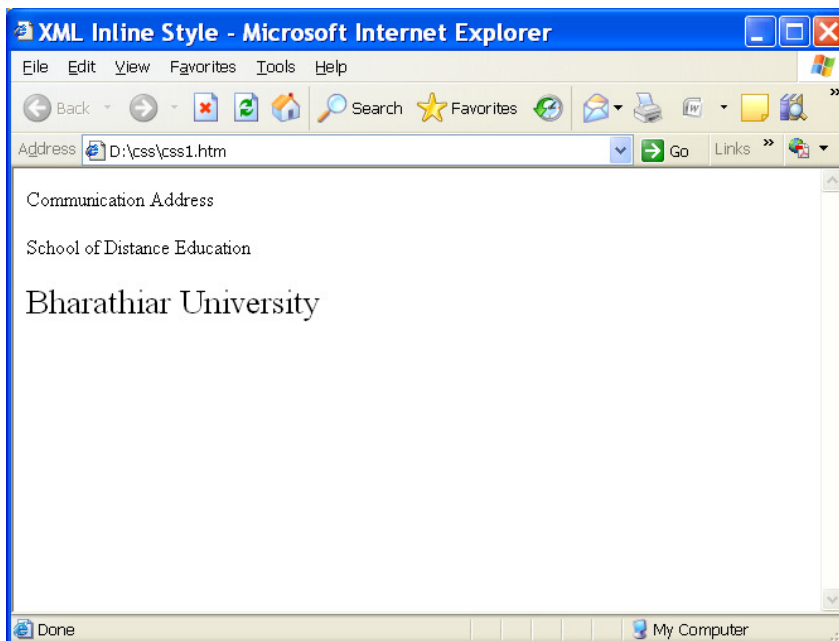
---

There are many ways to declare styles for a document. In Inline Style individual element's style is declared using the style attribute.

Example

```
<html>
<head>
<title> XML Inline Style</title>
</head>
<body>
<p> Communication Address </p>
<p style ="font-size: 20pt"> School of Distance Education</p>
<p style = font-size:20pt; color: #0000ff"> Bharathiar University </p>
</body>
</html>
```

Output:



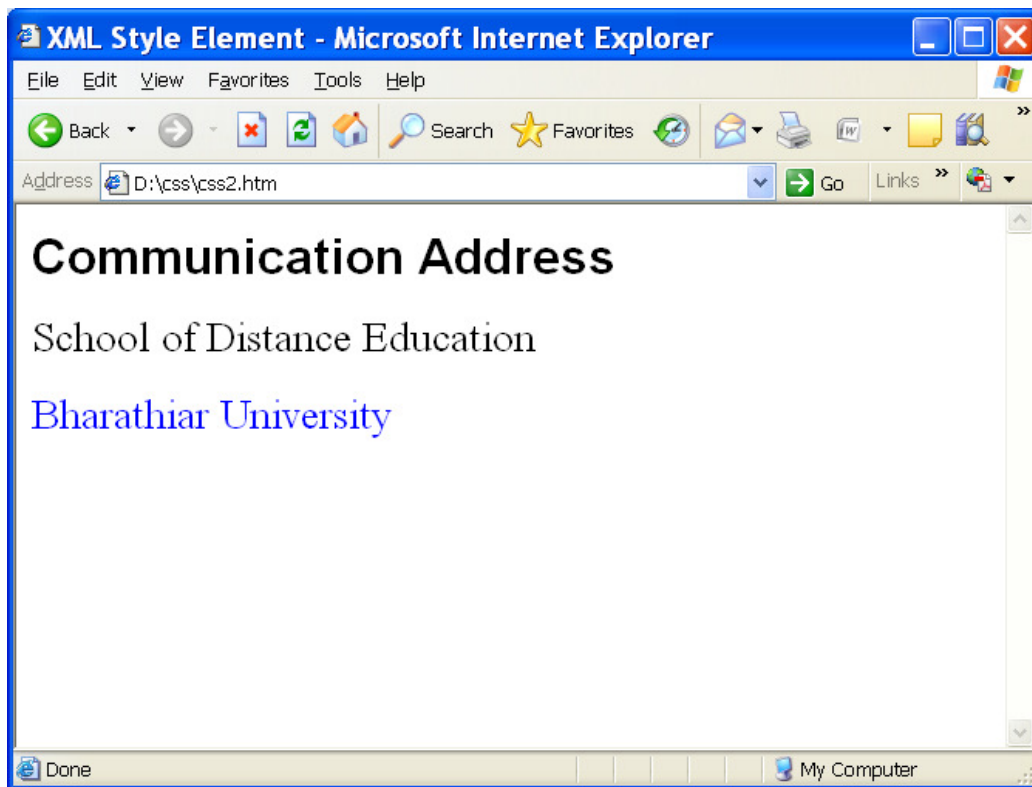
## 9.4.Creating Style Sheets with the Style Element

If we declare styles in the head of the document. These styles may be applied to the entire document.

Example

```
<html>
<head>
<title> XML Style Element </title>
<style type = "text/css">
em    { background-color : #8000ff; color:white }
h1    { font-family: arial, sans-serif }
p     { font-size : 20pt }
.special { color: blue }
</style>
</head>
<body>
<h1> Communication Address </h1>
<p>  School of Distance Education</p>
<p class = "special"> Bharathiar University </p>
</body>
</html>
```

Output :



---

## 9.5. Conflicting Styles

---

CSS style sheets are cascading because styles may be defined by user, an author and a user agent. Styles defined by authors take precedence over styles defined by the user, and styles defined by the user take precedence over styles defined by the user agent. Styles defined for parent and ancestor elements are also inherited by child and descendant elements.

Example

```
<html>

<head>

<title> XML More Style Element </title>

<style type = "text/css">

a.nodect      { text-decoration: none }

a:hover      { text-decoration : underline; color: red; background-color :
#ccffcc }

li em { color: red; font-weight:bold }

ul      { margin-left: 75px }
```

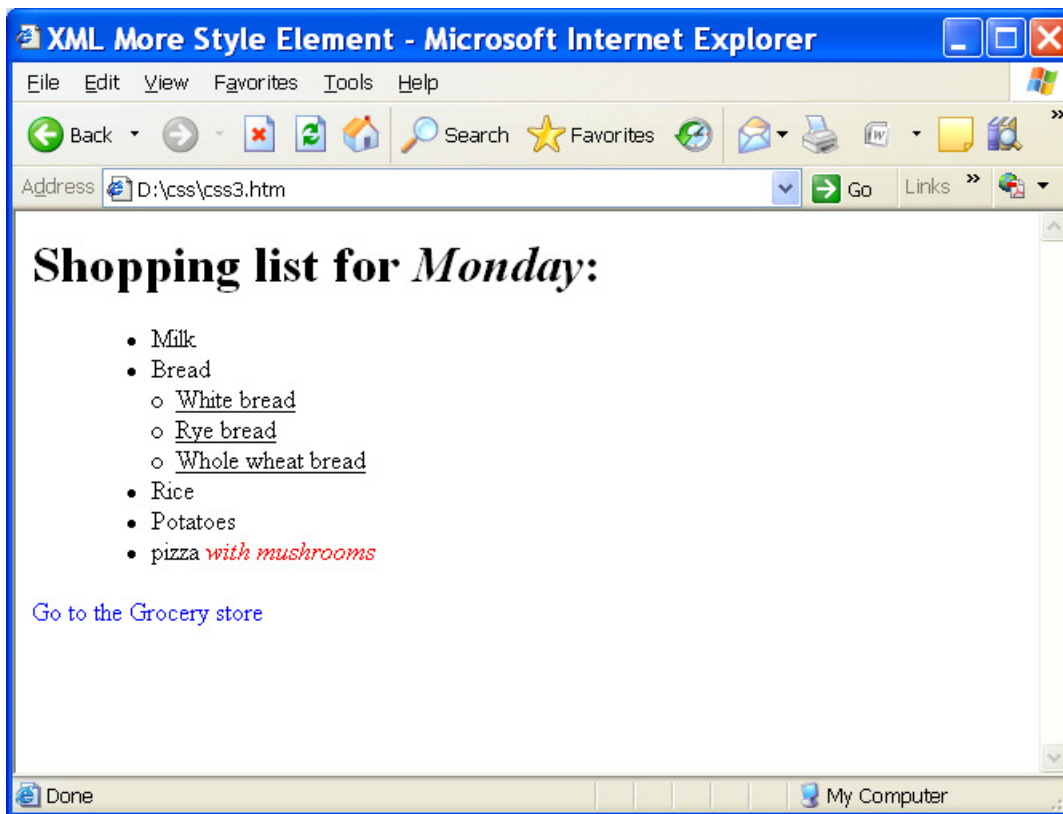


```

ul ul { text-decoration : underline; margin-left: 15px }
</style>
</head>
<body>
<h1> Shopping list for <em>Monday</em>:</h1>
<ul>
<li>Milk</li>
<li>Bread
    <ul>
    <li> White bread</li>
    <li>Rye bread</li>
    <li>Whole wheat bread</li>
    </ul>
</li>
<li>Rice</li>
<li>Potatoes</li>
<li>pizza <em> with mushrooms </em> </li>
</ul>
<p><a class = "nodec" href = "http://food.com"> Go to the Grocery store
</a> </p>
</body>
</html>

```

Output :




---

## 9.6. Linking External Style Sheets

---

Style sheets are an efficient way to give a document a uniform theme. With external linking, you can give your whole web site a uniform look – separate pages on your site can all use the same style sheet, and you only need to modify only a single file to make changes to styles across your whole web site.

Example

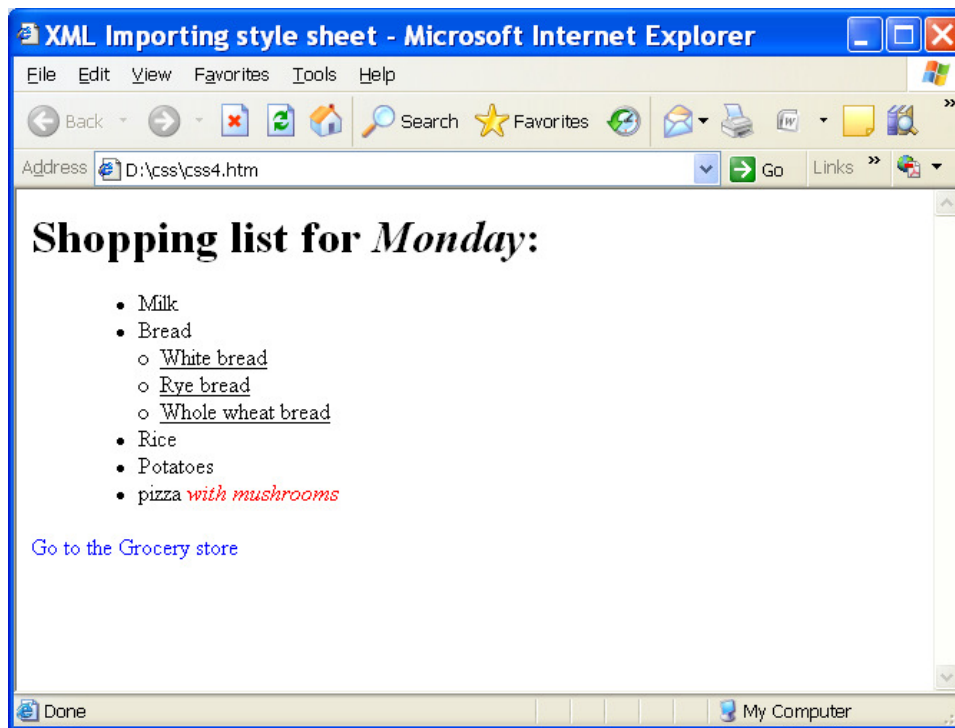
File name styles.css

```
a.nodect      { text-decoration: none }
a:hover       { text-decoration : underline; color: red; background-color :
#ccffcc }
li em { color: red; font-weight:bold }
ul           { margin-left: 75px }
ul ul       { text-decoration : underline; margin-left: 15px }
```

file name : css4.htm

```
<html>
<head>
<title> XML Importing style sheet </title>
<link rel = "stylesheet" type = "text/css" href = "styles.css">
</head>
<body>
<h1> Shopping list for <em>Monday</em>:</h1>
<ul>
<li>Milk</li>
<li>Bread
    <ul>
        <li> White bread</li>
        <li>Rye bread</li>
        <li>Whole wheat bread</li>
    </ul>
</li>
<li>Rice</li>
<li>Potatoes</li>
<li>pizza <em> with mushrooms </em> </li>
</ul>
<p><a class = "nodec" href = "http://food.com"> Go to the Grocery store
</a> </p>
</body>
</html>
```

Output :




---

## 9.7. Positioning element

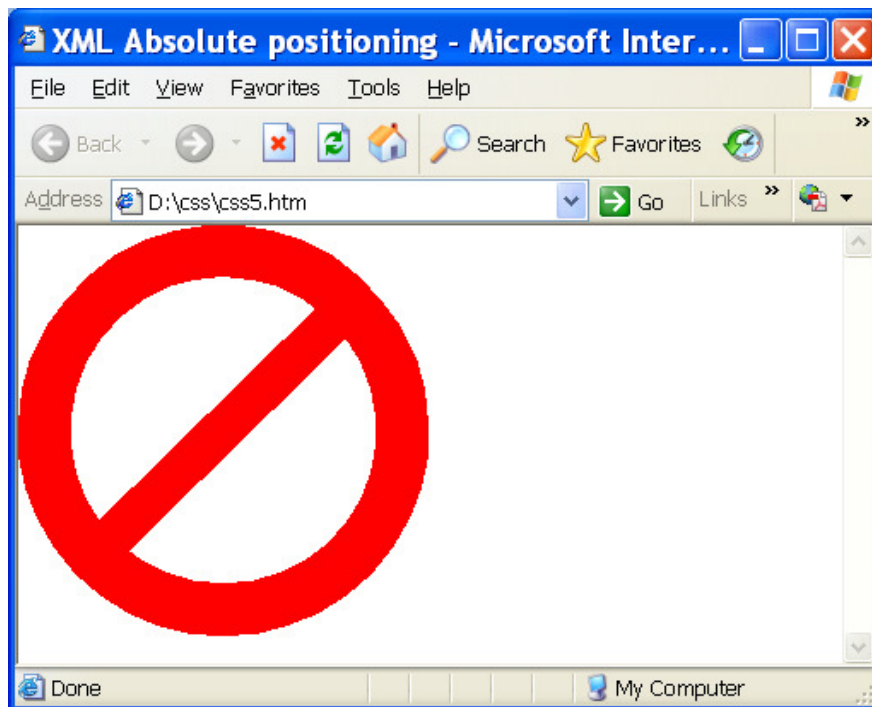
---

Controlling the positioning of elements in an HTML document was difficult; positioning was basically up to the browser. CSS introduces the position property and a capability called absolute positioning, which gives authors greater control over how documents are displayed.

Example : Absolute positioning

```
<html>
<head>
<title> XML Absolute positioning </title>
</head>
<body>
<p> <img src = "d:/css/slash.gif" style = "position : absolute; top: 0px;
left: 0px; z-index: 1" alt = "First positioned image"></p>
<p> <style = "position : absolute; top: 500px; left: 500px; z-index: 3; font-
size: 20pt;"> Positioned Text</p>
<p> <img src = "d:/css/hood.gif" style = "position : absolute; top: 300px;
left: 400px; z-index: 2" alt = "Second positioned image"></p>
</body>
</html>
```

Output



Relative Positioning

```
<html>
<head>
<title> XML Relative positioning </title>
<style type = "text/css">
p      { font-size: 1.3em; font-family : verdana, arial, sans-serif }
span   { color: red; font-size : .6em; height : 1em }
.super { position: relative; top: -1ex }
.sub   { position : relative; bottom : -1ex }
.shiftright { position: relative; left: -1ex}
.shiftright { position: relative; right: -1ex }
</style>
</head>
<body>
```

```

<p> The text at the end of this sentence <span class = "super"> is
in superscript</span>. </p>

<p> The text at the end of this sentence <span class = "sub"> is in
subscript</span>. </p>

<p> The text at the end of this sentence <span class = "shiftleft">
is shifted left</span>. </p>

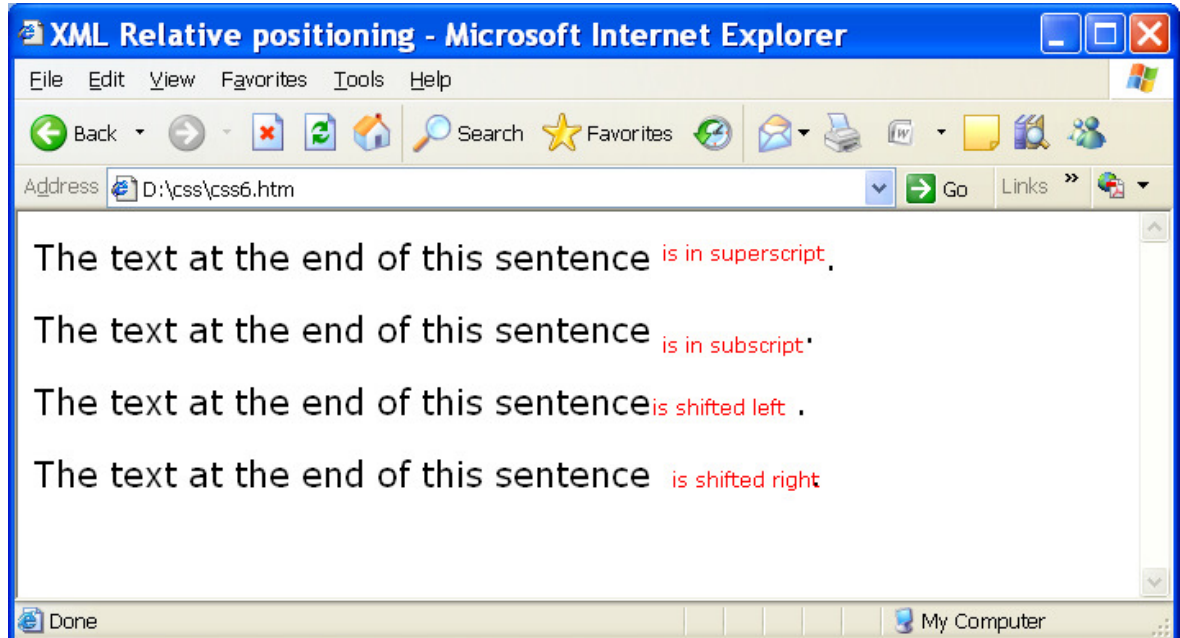
<p> The text at the end of this sentence <span class = "shiftright">
is shifted right</span>. </p>

</body>

</html>

```

Output :




---

## 9.8 Back ground

---

CSS also gives control over the backgrounds of elements. You can also add background images to your documents using CSS.

Example

```

<html>

<head>

<title> XML - Background Image </title>

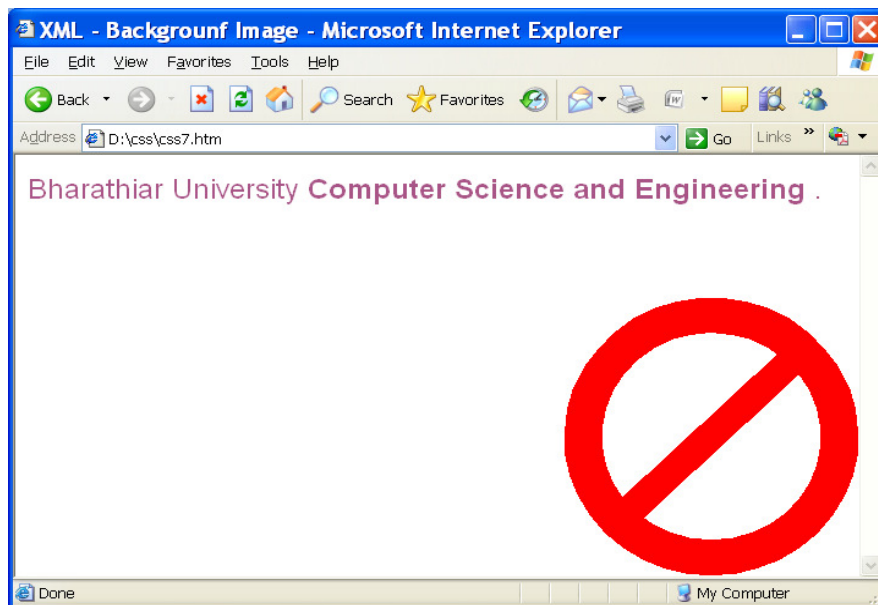
```

```

<style type = "text/css">
body  { background-image : url(slash.gif);
        background-position : bottom right;
        background-repeat : no-repeat;
        background-attachment: fixed; }
{ font-size: 18pt;
  color: #aa5588;
  text-indent: 1em;
  font-family: arial, sans-serif;}
.dark {font-weight : bold}
</style>
</head>
<body>
<p> Bharathiar University <span class = "dark"> Computer
Science and Engineering </span>. </p>
</body>
</html>

```

Output :



---

## 9.9 Element Dimension

---

The dimension of each element on the page can be specified using CSS.

Example :

```
<html>
<head>
<title> XML - Box Dimension </title>
<style type = "text/css">
div    { background-color: #ffccff; margin-bottom : .5em }
</style>
</head>
<body>
<div style = "width: 20%"> Bharathiar University </div>
<div style = width: 80%; text-align: center"> School of Computer Science
and Engineering. </div>
<div style = width: 20%; height : 30% ; overflow: scroll"> The Director,
School of Distance Education, Bharathiar University, Coimbatore - 641
046.</div>
</body>
</html>
```



---

## 9.10. Text Flow and the Box Model

---

A browser normally places text and elements on screen in the order they appear in the HTML document. However, as we saw with absolute positioning, it is possible to remove elements from the normal flow of text. Floating allows you to move an element to one side of the screen – other content in the document will then flow around the floated element. In addition, each block-level element has a box drawn around it, known as the box model – the properties of this box are easily adjusted. In addition to the text, whole element can be floated to left or right of a document.

Example

```
<html>

<head>

<title> XML Flowing text Around Floating Elements </title>

<style type = "text/css">

    div { background-color: #ffccff; margin-bottom: .5em; font-size:
1.5.em;Width: 50% }

    p      { text-align:justify; }

</style>

</head>

<body>

<div style = "text-align:center">XML programming.</div>

<div style = "float:right; margin: .5em; text-align:right"> Cascading Style
Sheets</div>

<p> A browser normally places text and elements on screen in the order
they appear in the HTML document. However, as we saw with absolute
positioning, it is possible to remove elements from the normal flow of
text. Floating allows you to move an element to one side of the screen –
other content in the document will then flow around the floated element.

</p>
```

```
<div style = "float:right; padding:.5em; text-align:right"> Floating  
Elements </div>
```

```
<p>In addition, each block-level element has a box drawn around it,  
known as the box model – the properties of this box are easily adjusted.  
In addition to the text, whole element can be floated to left or right of a  
document.</span> </p>
```

```
</body>
```

```
</html>
```

Example 2:

```
<head>
```

```
<title> XML – Borders</title>
```

```
<style type = "text/css">
```

```
Body {      background-color:#ccffcc }
```

```
Div   { text-align:center; margin-bottom: 1em; padding : .5em }
```

```
.thick { border-width: thick }
```

```
.medium { border-width: medium }
```

```
.thin { border-width: thin }
```

```
.groove { border-style: groove }
```

```
.inset { border-style: inset }
```

```
.outset { border-outset: outset }
```

```
.red { border-color:red}
```

```
.blue { border-color:blue}
```

```
</style>
```

```
</head>
```

```
<body>
```

```
<div class ="thick groove">This text has border</div>
```

```
<div class ="medium groove">This text has border</div>
```

```
<div class ="thin groove">This text has border</div>
```

```
<p class = "thin red inset"> A thin red line ... </p>
```

```
<p class = "medium blue outset"> and a thicker blue line </p>
```

```
</body>
```

```
</html>
```

---

### 9.11. User Style Sheets

---

Users have the option to define their own user style sheets to format pages based on their own preference – for eg. Visually impaired people might want to increase the text size on all pages they view.

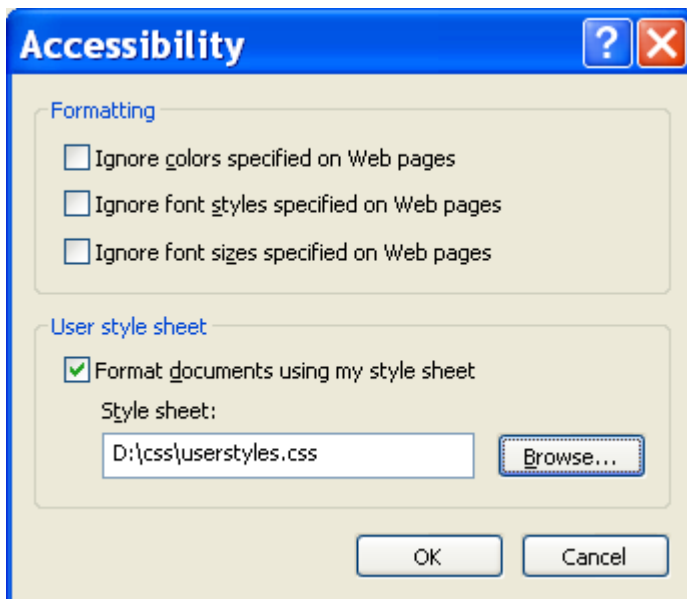
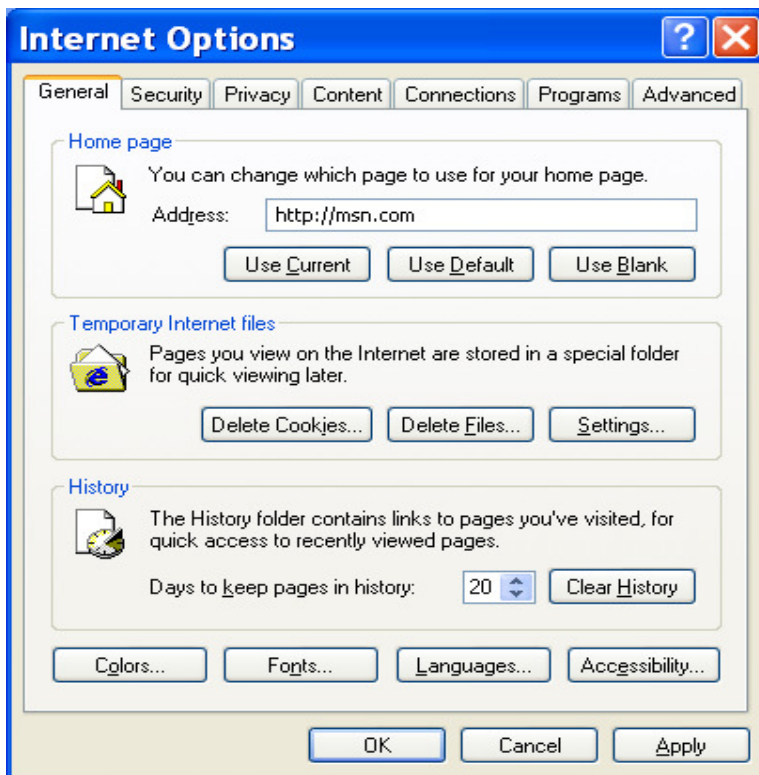
Example : user.html

```
</head>
<title> XML user style </title>
<style type = "text/css">
.note { font-size : 1.5em }
</style>
</head>
<body>
<p> Thanks for visiting my web page. I hope you enjoy it. </p>
<p class = "note"> Please note: This site will be moving soon. Please
check the periodically for updates </p>
</body>
</html>
```

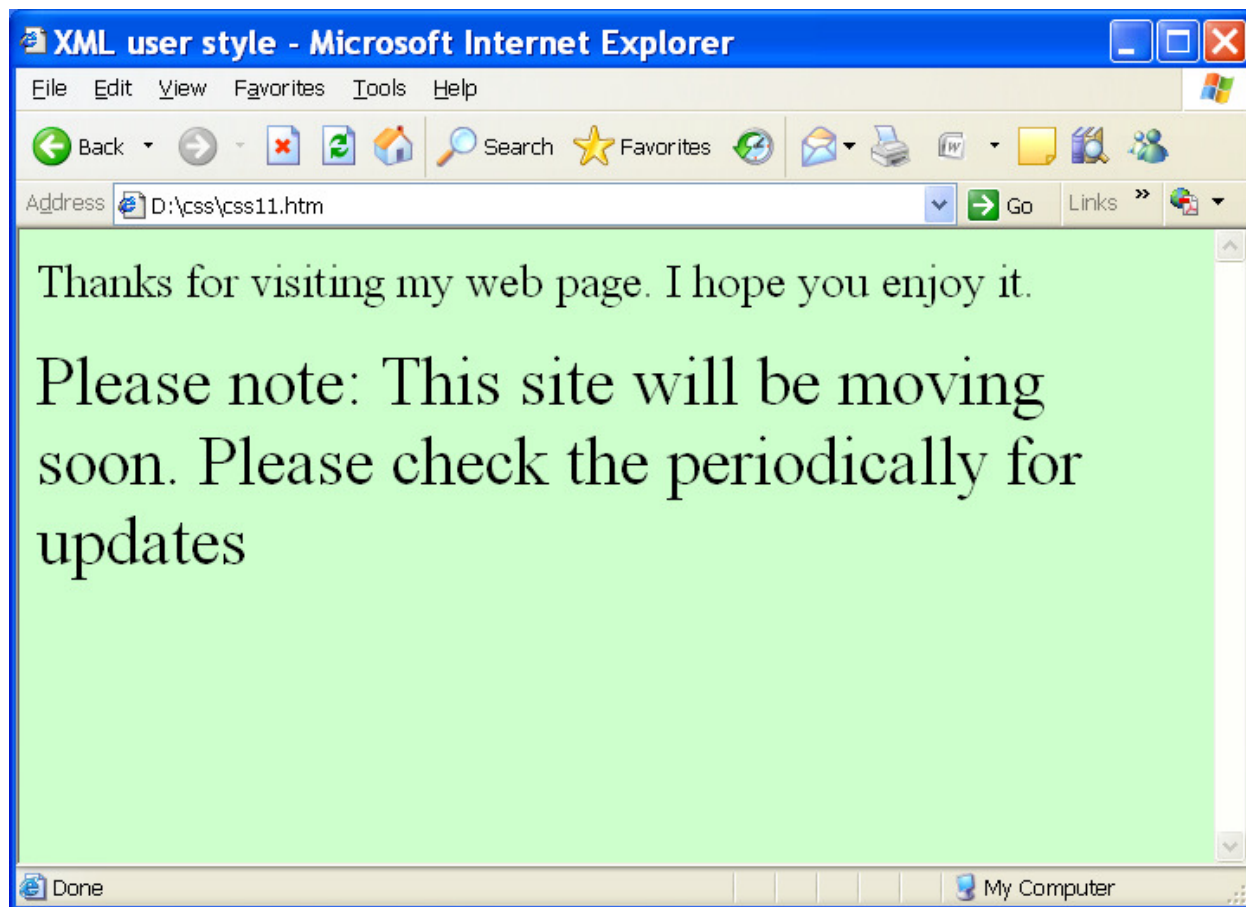
File Name : userstyle.htm

```
body { font-size:20pt; background-color: #ccffcc }
a      { color : red }
```

In this case the user have to modify the browser options



Output :



---

### 9.12. Let us Sum Up

---

HTML has been primarily concerned with representing the structure of documents online. By this, I mean that it allows the author to identify headings, paragraphs, lists, etc., but it does not provide (very many) facilities for specifying how the information can be presented (fonts, colors, spacing, text flow, etc.).

#### **HTML is the Structure**

The problem is that HTML, the primary markup language used to code documents on the World Wide Web, is really only useful for expressing the structure of a document.

It is possible to exert some control over the presentation, by employing tables and a variety of other tricks, but doing so blurs the structural view of the document. Adding new presentational tags to HTML isn't going to help, either.

Presentational tags further blur the underlying structure of the document and could lead rapidly to multiple, incompatible HTML variants.

### **Style Sheets are the Presentation**

Style sheets, which should become commonplace on the Web over the next few months, provide a means of associating presentational information with the structural elements of a document in a way that does not corrupt the underlying structure of the document.

A style sheet is a set of guidelines for the browser indicating how the various elements of a document should be presented. For example, the following set of instructions constitute a style sheet for web documents:

- The document background should be blue.
- Top-level headings should be in 20 point Bold Arial (or Helvetica, or at least a sans-serif face).
- Body text should be 10 point Times Roman. Body text should be white; links should be light red; visited links should be yellow.
- Block quotations should be set in 8 point Times Italic. The body text should be black and the background white.
- Warnings should be indented on both sides and set in yellow.
- Itemized lists should use a fancy bullet.

### **Inline Style**

There are many ways to declare styles for a document. In Inline Style individual element's style is declared using the style attribute.

### **Creating Style Sheets with the Style Element**

If we declare styles in the head of the document. These styles may be applied to the entire document.

### **Conflicting Styles**

CSS style sheets are cascading because styles may be defined by user, an author and a user agent. Styles defined by authors take precedence over styles defined by the user, and styles defined by the user take precedence over styles defined by the user agent. Styles defined for parent and ancestor elements are also inherited by child and descendant elements.

## **Linking External Style Sheets**

Style sheets are an efficient way to give a document a uniform theme. With external linking, you can give your whole web site a uniform look – separate pages on your site can all use the same style sheet, and you only need to modify only a single file to make changes to styles across your whole web site.

### **Positioning element**

Controlling the positioning of elements in an HTML document was difficult; positioning was basically up to the browser. CSS introduces the position property and a capability called absolute positioning, which gives authors greater control over how documents are displayed.

### **Back ground**

CSS also gives control over the backgrounds of elements. You can also add background images to your documents using CSS.

## **Element Dimension**

The dimension of each element on the page can be specified using CSS.

## **Text Flow and the Box Model**

A browser normally places text and elements on screen in the order they appear in the HTML document. However, as we saw with absolute positioning, it is possible to remove elements from the normal flow of text. Floating allows you to move an element to one side of the screen – other content in the document will then flow around the floated element. In addition, each block-level element has a box drawn around it, known as the box model – the properties of this box are easily adjusted. In addition to the text, whole element can be floated to left or right of a document.

## **User Style Sheets**

Users have the option to define their own user style sheets to format pages based on their own preference – for eg. Visually impaired people might want to increase the text size on all pages they view.

---

**9.13. Lesson end Activities**

---

1. What is the need for CSS?
2. Write a CSS program for user style.

---

**9.14. Check your Progress**

---

1. Write a CSS program for Element dimension.
2. Write a CSS program for linking external style sheets.

---

**9.15. Reference**

---

1. XML How to Program , deitel, Nieto, Lin and Sadhu, Pearson Education,2004



## **Dynamic HTML: Object Model and Collections**

---

### **Contents**

#### **10.0 Aim and Objective**

#### **10.1 Introduction**

#### **10.2. Collection all and Children**

#### **10.3. Dynamic Style and Positioning**

#### **10.4. Cross-frame Referencing**

#### **10.5. Navigator Object**

#### **10.6. Let us Sum Up**

#### **10.7. Lesson end Activities**

#### **10.8. Check your Progress**

#### **10.9. Reference**

---

#### **10.0 Aim and Objective**

---

- **To use the Dynamic HTML – Object Model and Collection**
- **To understand DHTML object hierarchy**
- **To use all and children collection**
- **To use dynamic styles and dynamic positioning**
- **To use frames collection**
- **To use the navigator object**

---

#### **10.1. Introduction**

---

The object model gives access to all elements of a Web page, whose properties and attributes can thus be retrieved or modified by scripting. The value of the id attribute of an element becomes the name of the object representing that element. The various HTML attributes of the element become properties of this object (which can be modified).

For example, the value of the `innerText` property of a `p` element is the text within that element. So, if we have a `P` element with `id` `pText`, we can dynamically change the rendering of this element with, e.g.,

```
pText.innerText = "Good bye!";
```

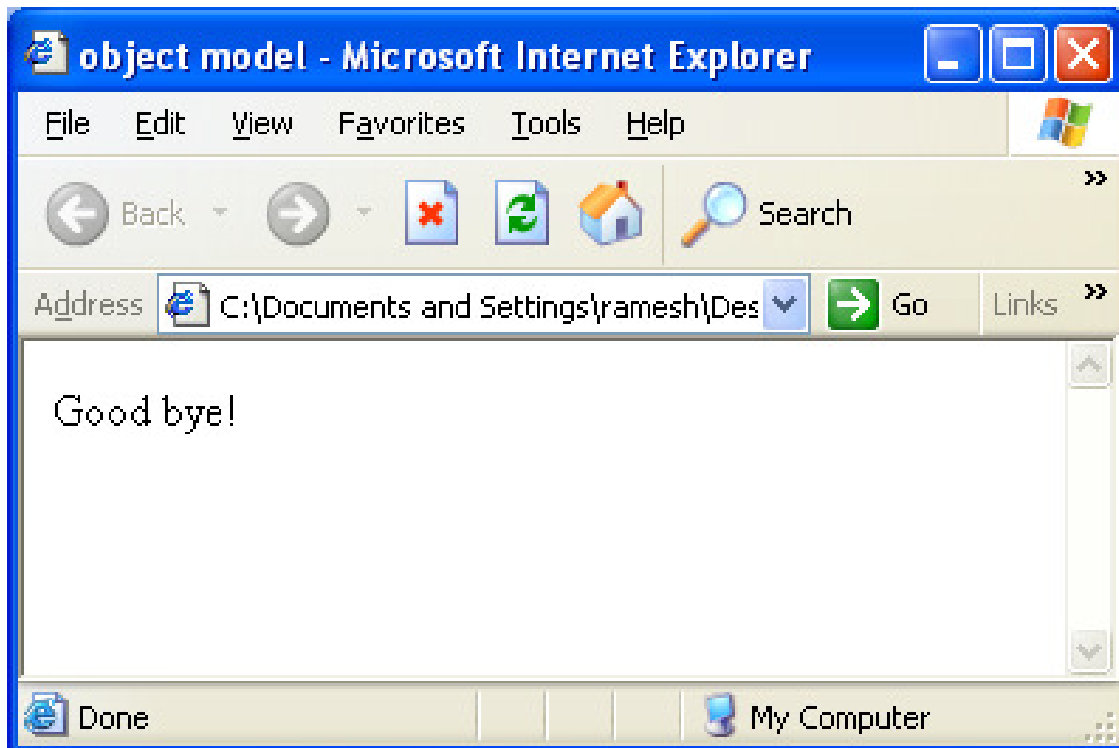
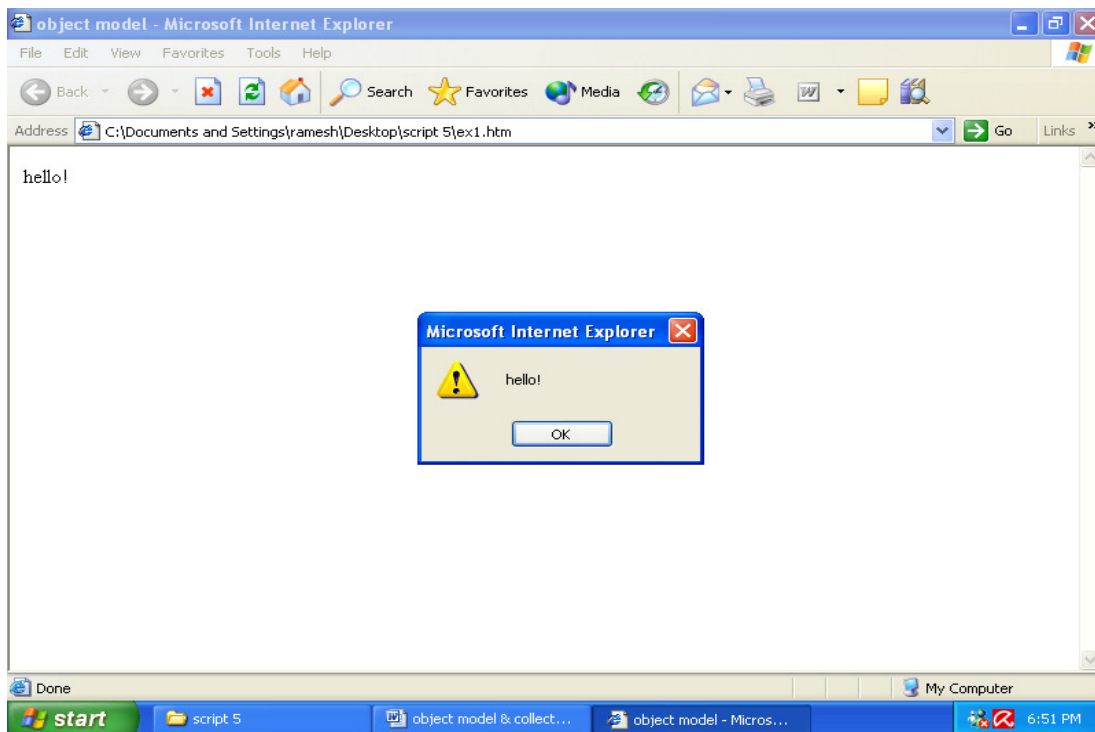
This is dynamic content.

In the following example, the function (not the window method) `alert` is used to pop up an alert box.

*Example: 1*

```
<html>
<head>
<title>object model</title>
<script type = "text/javascript">
function start()
{
alert( pText.innerText );
pText.innerText = "Good bye!";
}
</script>
</head>
<body onload = "start()">
<p id = "pText">hello!</p>
</body>
</html>
```

Output :



---

## 10.2. Collection all and children

---

A *collection* is an array of related objects on a page. The `all` collection of an element (syntactically a property) is a collection of all the elements in it in order of appearance. This gives us reference even to elements that lack ID attributes. Like all collections, it has a `length` property.

For example,

`document.all[ i ]` references the *i*th element in the document.

The `innerHTML` property of a `p` element is like the `innerText` property but may contain HTML formatting. The `tagName` property of an element is the name of the HTML element.

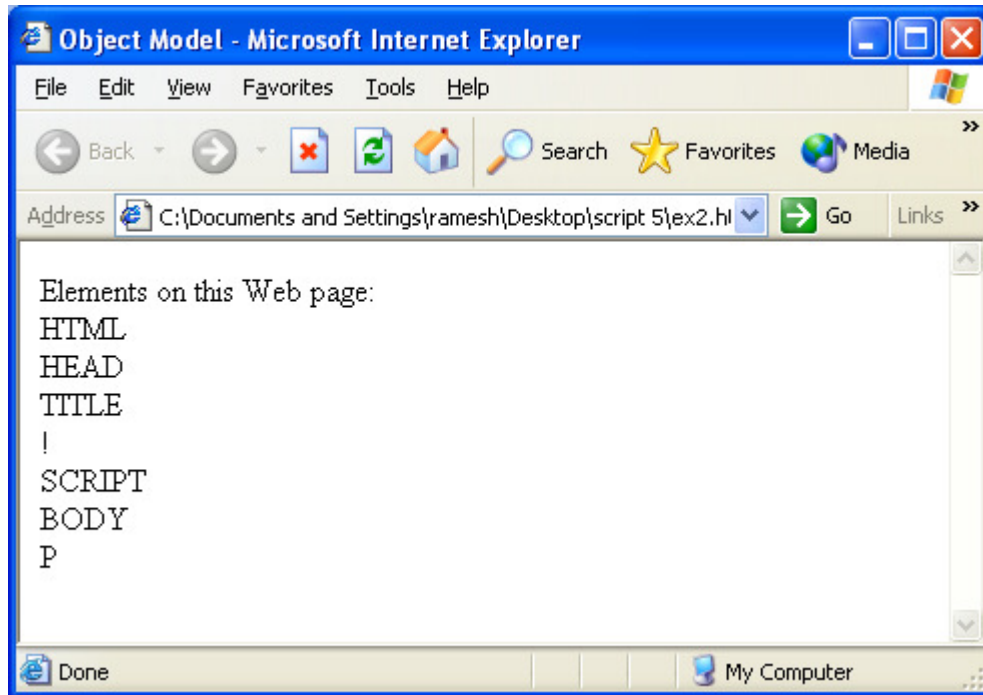
Example: 2

```
<html>
<!-- Using the all collection -->
<head>
<title>Object Model</title>
<script type = "text/javascript">
var elements = "";
function start()
{
for ( var loop = 0;
loop < document.all.length; ++loop )
elements += "<BR>" +
document.all[ loop ].tagName;
pText.innerHTML += elements;
}
</script>
</head>
<body onload = "start()">
<p id = "pText">Elements on this Web page:</p>
```

</body>

</html>

Output :



Note that the tagName property of a comment is !.

The children collection for an object is like the all collection but contains only the next level down in the hierarchy. For example, an HTML element has a head and a body child. In the following example, function child(object) does a preorder traversal of the part of the object hierarchy rooted at object. For every object with children, it appends on to global variable elements

The script adds ul and li tags to display the elements in a hierarchical manner on the page. <li>, tags represent the name of the HTML element represented by the object, <ul>,tag represent □similar information for the children (iteratively) and more distant descendants (recursively) of the object, and □</ul>.

The body tag is

```
<body onload = "child( document.all[ 1 ] );  
myDisplay.outerHTML += elements;">
```

When the page is loaded, this calls child, passing it the second object in the hierarchy.

(The first element is the comment at the top of the file.)

When control returns from the call, the string in global variable elements (containing the hierarchical description of the objects) is appended to the value of the outerHTML property of P element myDisplay. Property outerHTML is like innerHTML but includes the enclosing tags.

*Example: 3*

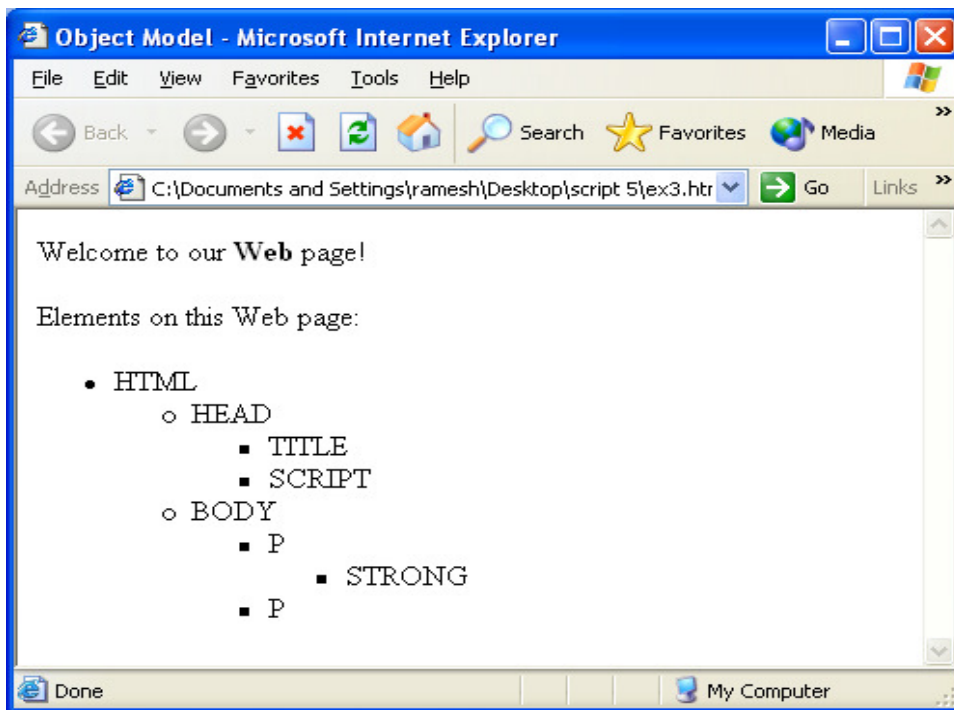
```
<!-- The children collection -->
<head>
<title>Object Model</title>
<script type = "text/javascript">
var elements = "<ul>";
function child( object )
{
var loop = 0;
elements += "<LI>" + object.tagName + "<UL>";
for( loop = 0; loop < object.children.length;
loop++ )
if ( object.children[loop].children.length )
child( object.children[ loop ] );
else
elements += "<LI>" +
object.children[ loop ].tagName
+ "</LI>";
elements += " </UL> ";
}
</script>
</head>
```

```

<body onload = "child( document.all[ 1 ] );
myDisplay.outerHTML += elements;">
<p>Welcome to our <strong>Web</strong> page!</p>
<p id = "myDisplay">
Elements on this Web page:
</p>
</body>
</html>

```

Output :




---

### 10.3. Dynamic Styles and Positioning

---

We can change an element's style dynamically. Most HTML elements have a style object as a property. The names of the properties of this object used in JavaScript are generally their HTML names modified to avoid the “-“ (seen as subtraction in JavaScript) – e.g., HTML JavaScript background-color backgroundColor border-width borderWidth font-family fontFamily .

We can make assignments to these properties, dynamically changing the element's rendering – e.g.,

```
document.body.style.fontSize = 16;
```

Suppose an element's CSS position property is declared to be either absolute or relative. Then we can move it by manipulating any of the top, left, right, or bottom CSS properties of its style object.

This is *dynamic positioning*.

Example

Suppose in the body we have

```
<p id = "pText1"
style = "position: absolute; top: 35">
XXX</p>
```

and in the script we have

```
pText1.style.left = 100;
```

Then the rendering XXX of the pText1 element will be shifted right 100 pixels.

We can also change the class attribute of an element by assigning the name of a class we have defined to the element's className property.

Example

Suppose in the body we have `<p id = "pText2">CCC</p>` in the style sheet we have defined `.bigText { font-size: 2em }` and in the script we have `pText2.className = "bigText";`

Then the rendering of XXX will be twice as large as the surrounding text.

*Example: 4*

```
<html>
<head>
<title>Dynamic Styles</title>
<style type = "text/css">
.bigText { font-size: 2em }
</style>
<script type = "text/javascript">
```

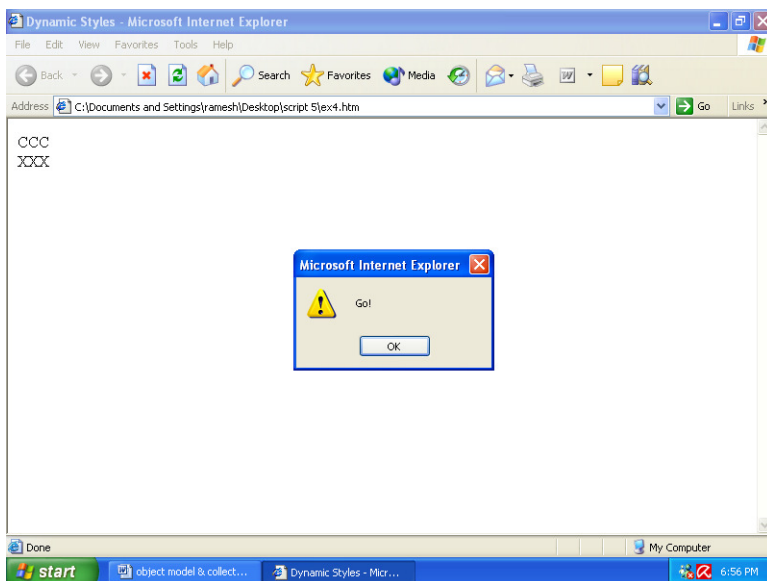


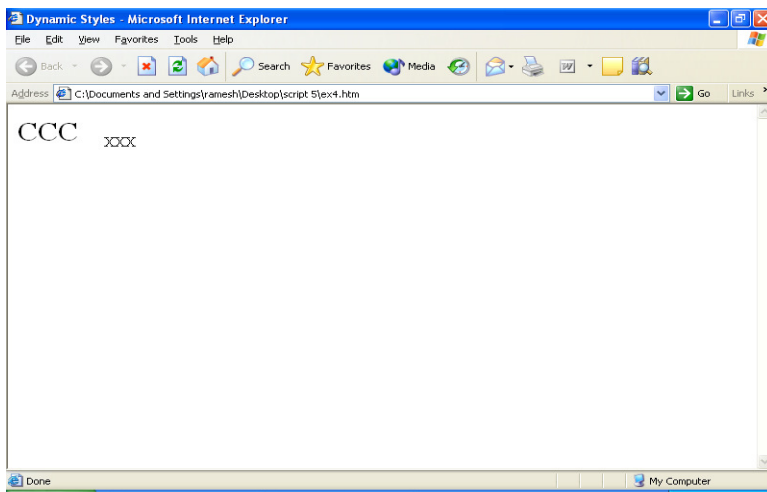
```

function start()
{
    alert( "Go!" );
    document.body.style.fontSize = 16;
    pText1.style.left = 100;
    pText2.className = "bigText";
}
</script>
</head>
<body onload = "start()">
<p id = "pText1"
STYLE = "position: absolute; top: 35">
XXX</p>
<p id = "pText2">CCC</p>
</body>
</html>

```

Output :





The initial rendering is:

After the alert dialog box is dismissed, the rendering is:

To get perceptible dynamic effects, we need some way to control when element attributes and properties are changed. The `setInterval` method of the window object, used as `window.setInterval( "function_name()", msec)`, invokes function *function\_name* every *msecs* milliseconds.

Method `setTimeout` has the same parameters, but it waits *msecs* milliseconds before invoking *function\_name* only once. Both `setInterval` and `setTimeout` return values, which can be assigned to variables. Method `clearInterval` takes the value returned by `setInterval` and terminates the timed function's executions.

Method `clearTimeout` takes the value returned by `setTimeout` and stops the timer before it fires (if it hasn't already).

Example

```
timer = window.setInterval( "f()", 1000);
```

...

```
window.clearInterval( timer );
```

Example: 5

```
<html>
<head>
<title>Dynamic Positioning</title>
```

```

<script language = "javascript">
var speed = 5;
var count = 10;
var direction = 1;
var firstLine = "Text growing";
var fontStyle =
[ "serif", "sans-serif", "monospace" ];
var fontStylecount = 0;

```

```

function start()
{
window.setInterval( "run()", 100 );
}

```

Continued next page

```

function run()
{
count += speed;
if ( ( count % 200 ) == 0 ) {
speed *= -1;
direction = !direction;
pText.style.color =
( speed < 0 ) ? "red" : "blue" ;
firstLine =
( speed < 0 ) ? "Text shrinking" :
"Text growing";
pText.style.fontFamily =
fontStyle[ ++fontStylecount % 3 ];
}
}

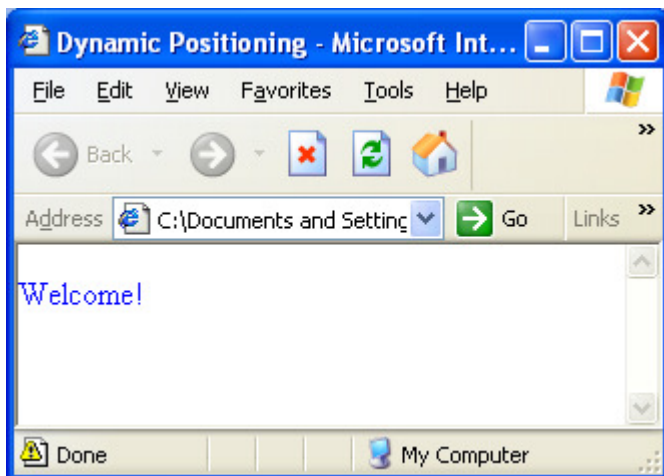
```

```

pText.style.fontSize = count / 3;
pText.style.left = count;
pText.innerHTML = firstLine +
"<BR> Font size: " +
count + "px";
}
</script>
</head>
<body onload = "start()">
<p id = "pText"
STYLE = "position: absolute; left: 0;
font-family: serif; color: blue">
Welcome!</p>
</body>
</html>

```

Output :




---

## 10.4. Cross-Frame Referencing

---

With frame sets, we have the problem of referencing elements that are on different pages. In a page loaded into a frame, parent references the parent page (containing a frame element with the current page's URL as the value of its src attribute). Then parent frames is the collection of frames in the parent of the

current page. We can reference any page that is the source of some frame in the parent's frame set either □by using the ordinal (0 to the number of frames minus one) of the frame within the frame set as an index or by using the value of the name attribute of the desired frame as an argument.

Example

```
parent.frames[ 1 ]  
parent.frames( "lower" )
```

This gets us to the document level of the page. If no page is loaded into the selected frame, the frame's document object is still defined – designating the space rendered for that frame in the rendering of the parent.

Example

```
parent.frames( "lower" ).document.writeln("<p>lower</p>");
```

*Example: 6*

The following is a page that defines a frame set. One of its frames has "frameset2.html" as the value of its SRC attribute.

```
<html>  
<head>  
<title>Frames collection</title>  
</HEAD>  
<frameset rows = "100, *">  
<frame src = "frameset2.html" name = "upper">  
<frame src = "" name = "lower">  
</frameset>  
</html>
```

The following is file frameset2.html:

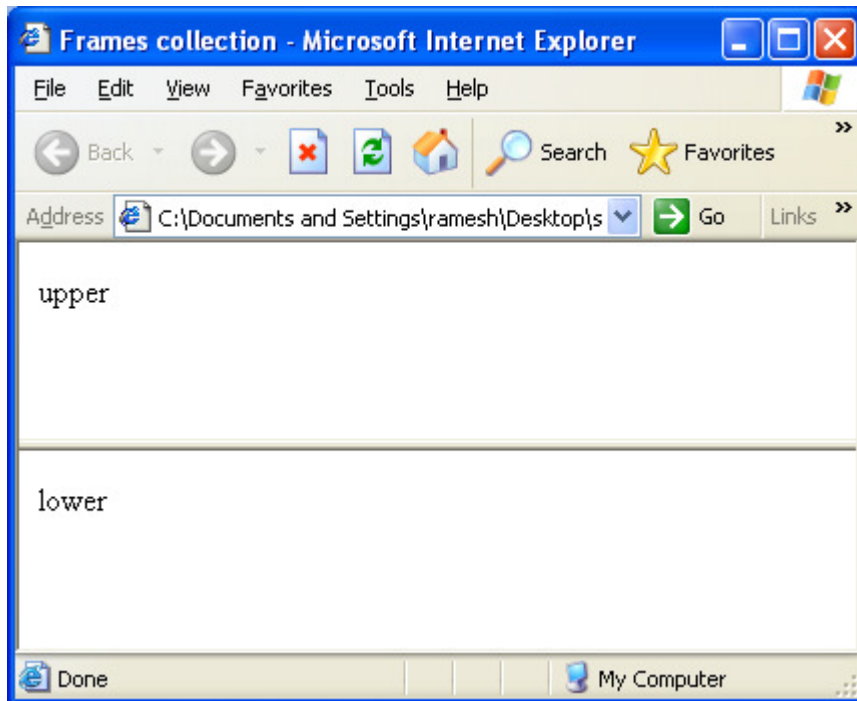
```
<html>  
<head>  
<title>The frames collection</title>  
<script type = "text/javascript">  
function start()
```

```

{
parent.frames( "lower" ).document.write(
"<p>lower</p>" );
}
</script>
</head>
<body onload = "start()">
<p>upper</p>
</body>
</html>

```

Output




---

## 10.5. The Navigator Object

---

Both Netscape's Navigator and Microsoft's Internet Explorer support the navigator object. It contains information about the browser that's viewing the page. Property `navigator.appName` is "Microsoft Internet Explorer" if the application is Internet Explorer and "Netscape" if the application is Netscape's

Navigator. Property navigator.appVersion is a string of various information, starting with the version number.

For the following example, note that document.location is the URL of the document being viewed.

*Example: 7*

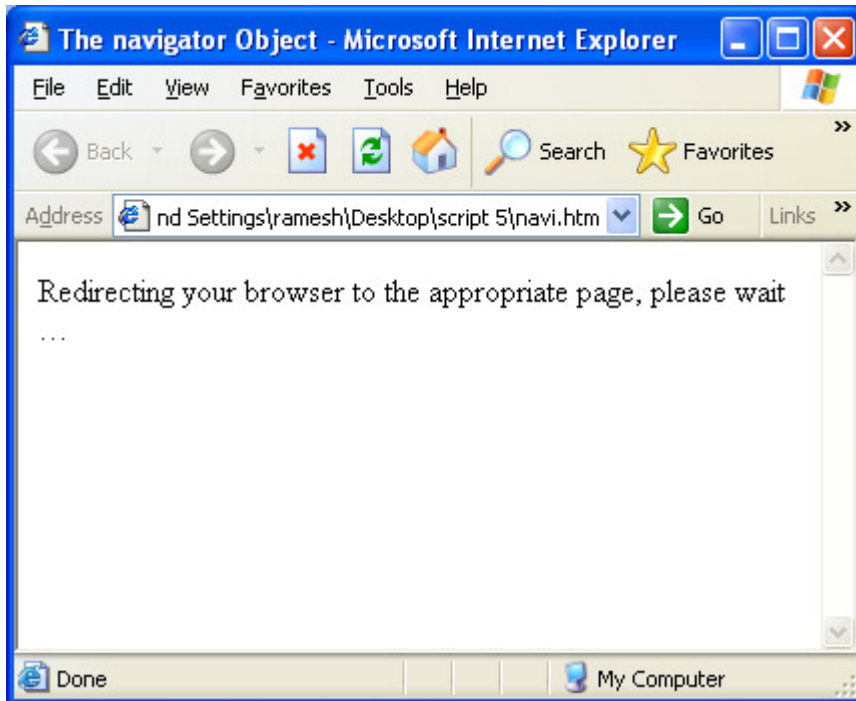
```
<html>
<head>
<title> The navigator Object</title>
<script type = "text/javascript">

function start()
{
if ( navigator.appName == "Microsoft Internet Explorer" ) {
if ( navigator.appVersion.substring(1,0) >= "4" )
document.location = "newIEVersion.html";
else
document.location = "oldIEVersion.html";
}
else
document.location = "NSVersion.html";
}

</script>
</head>

<body onload = "start()">
<p> Redirecting your browser to the appropriate page, please wait ... </p>
</body>
</html>
```

Output :



---

## 10.6 Let us Sum Up

---

The object model gives access to all elements of a Web page, whose properties and attributes can thus be retrieved or modified by scripting. The value of the id attribute of an element becomes the name of the object representing that element. The various HTML attributes of the element become properties of this object (which can be modified).

### Collection all and children

A *collection* is an array of related objects on a page. The all collection of an element (syntactically a property) is a collection of all the elements in it in order of appearance. This gives us reference even to elements that lack ID attributes. Like all collections, it has a length property.

The children collection for an object is like the all collection but contains only the next level down in the hierarchy. For example, an HTML element has a head and a body child. In the following example, function child(object) does a preorder traversal of the part of the object hierarchy rooted at object. For every object with children, it appends on to global variable elements



## Dynamic Styles and Positioning

We can change an element's style dynamically. Most HTML elements have a style object as a property. The names of the properties of this object used in JavaScript are generally their HTML names modified to avoid the “-“ (seen as subtraction in JavaScript) – e.g., HTML JavaScript background-color backgroundColor border-width borderWidth font-family fontFamily .

## Cross-Frame Referencing

With frame sets, we have the problem of referencing elements that are on different pages. In a page loaded into a frame, parent references the parent page (containing a frame element with the current page's URL as the value of its src attribute). Then parent frames is the collection of frames in the parent of the current page. We can reference any page that is the source of some frame in the parent's frame set either □by using the ordinal (0 to the number of frames minus one) of the frame within the frame set as an index or by using the value of the name attribute of the desired frame as an argument.

## The Navigator Object

Both Netscape's Navigator and Microsoft's Internet Explorer support the navigator object. It contains information about the browser that's viewing the page. Property navigator.appName is “Microsoft Internet Explorer” if the application is Internet Explorer and “Netscape” if the application is Netscape's Navigator. Property navigator.appVersion is a string of various information, starting with the version number.

---

### 10.7. Lesson end Activities

---

1. What is JavaScript Object model?
2. Write a JavaScript programe to positioning the text in the middle of the browser.

---

### 10.8. Check your Progress

---

1. Write a JavaScript programe to use more than two frames.
2. Write a programe to display the version of your browser.

---

## **10.9. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.w3c.com](http://www.w3c.com)

## Dynamic HTML - Event Model

---

### Contents

- 11.0. Aim and Objective
- 11.1. Introduction
- 11.2. Event Onclick
- 11.3. Event Onload
- 11.4. Error Handling with Onerror
- 11.5. Tracking the mouse with Event onmousemove
- 11.6. Rollover with onmouseover and onmouseout
- 11.7. Form processing with onfocus and onblur
- 11.8. Form processing with onsubmit and onreset
- 11.9. Event Bubbling
- 11.10. Let us Sum Up
- 11.11. Lesson End Activities
- 11.12. Check your Progress
- 11.13. Reference

---

### 11.0. Aim and Objectives

---

- To understand Event handling using mouse

---

### 11.1. Introduction

---

Dynamic HTML the pages can be controlled by the event models. This makes the web application more responsive and user friendly and can reduce server load. With the event model, scripts can respond to a user who is moving the mouse, scrolling up or down the screen or entering keystrokes. Content becomes more dynamic while interfaces become more intuitive.

---

### 11.2. Event onclick

---

One of the most common events is onclick. When the user clicks a specific item with the mouse, the onclick event fires. Using the for property of the script element allows you to specify the element to which the script applies.

Example 1

```
<html>

<head>

<title> DHTML Event Model – onclick</title>


<script type = "text/javascript" for ="para" event = "onclick">
alert("Hi there");
</script>
</head>


<body>
<p id = "para"> Click on this text!</p>
<input type = "button" value = "Click Me"
onclick = "alert("Hi Again")"/>
</body>
</html>
```

---

### 11.3. Event onload

---

The onload event fires whenever an element finishes loading successfully. Frequently, this event is used in the body element to initiate a script after the page loads into the client. The script called by the onload event updates a timer that indicates how many seconds have elapsed since the document was loaded.

Example 2

```
<html>

<head>
```

```

<title> DHTML Event Model – onload</title>

<script type = “text/javascript” >

Var seconds = 0;

Function startTimer() {
Window.setInterval(“updateTime()”,1000);
}

Function updateTime() {
Seconds++;
soFar.innerText = seconds;
}

</script>

</head>


<body  onload = “startTime()”>

<p> Second you have spent viewing this page so far: <strong id =
“soFar”>0</strong></p>

</body>

</html>

```

---

#### 11.4. Error Handling with onerror

---

The web is a dynamic medium. Sometimes a script refers to objects that existed at a specified location when the script was written, but the location changes at a later time, rendering the script invalid. The error dialog presented by the browser in such a case can be confusing to the user. To prevent this dialog box from displaying and to handle errors more elegantly, scripts can use the onerror event to execute specialized error handling code. To display and to handle errors more elegantly, scripts can use the onerror event to execute specialized error handling code. The function handleError should execute when an onerror occurs in the window object. The following example misspelled function name alrrt create an error and informed by the function handleError.

### Example 3

```
<html>
<head>
<title> DHTML Event Model – onerror</title>
<script type = “text/javascript”>
Window.onerror = handleError;
Function doThis() {
    Alrrt(“Hi”);
}
Function handleError (errType, errURL, errLineNum)
{
Window.status = “Error: “+errType + “on line”+ errLineNum;
Return true;
}
</script>
</head>
<body>
<input id = “mybbutton” type = “button” value = “ Click Me” onclick =
“doThis()”/>
</body>
</html>
```

The `handleError` function accepts three parameters from the `onerror` event, which is one of the few events that passes parameters to an event handler. The parameters are the type of error that occurred, the URL of the file that had the error and the line number on which the error occurred. This function return true to the event handler to indicate the the error has been handled. This prevent browser’s default response.

---

## 11.5. Tracking the Mouse with Event onmousemove

---

Event onmousemove fires repeatedly whenever the user moves over the Web page.

Example 4

```
<html>
<head>
<title> DHTML Event Model – onmousemove event </title>
Function updateMouseCoordinates()
{
Coordinates.innerText = event.srcElement.tagName + " (" + event.offsetX +
"," + event.offsetY + ")";
}
</script>
</head>
<body style = "background-color:wheat" onmousemove =
"updateMouseCoordinates()">
<span id = "coordinates">(0,0) </span><br/>
<img src = "deitel.gif" style = "position:absolute; top : 100; left:100" alt =
"Deitel"/>
</body>
</html>
```

The offsetX and offsetY properties of the event object give the location of the mouse cursor relative to the top-left corner of the object on which the event was triggered. The properties of the event object contain information about any events that occur on your page and are used to create Web pages that are truly dynamic and responsive to the user.

---

## 11.6. Rollovers with onmouseover and onmouseout

---

When the mouse cursor moves over an element, an onmouseover event occurs for that element. When the mouse cursor leaves the element, an

onmouseout event occurs for that element. If the image is large or the connection is slow, downloading causes a noticeable delay in the image update.

#### Example 5

```
<html>

<head>

<title> DHTML Event Model – onmouseover and onmouseout </title>

captionImage1 = new Image();
captionimage1.src = "caption.gif";
captionImage2 = new Image();
captionImage2.src = "caption2.gif";
function mOver()
{
    If ( event.srcElement.id == "tableCaption") {
        Event.srcElement.src = captionImage2.src;
        Return;
    }
    If (event.srcElement.id)
        Event.srcElement.style.color = event.srcElement.id;
}
function mOut()
{
    If ( event.srcElement.id == "tableCaption") {
        Event.srcElement.src = captionImage1.src;
        Return;
    }
    If (event.srcElement.id)
        Event.srcElement.innerText = event.srcElement.id;
}
Document.onmouseover = mOver;
```



```

Document.onmouseout = mOut;
</script>
</head>
<body style = "background-color:white">
<h1>Guess the Hex Code's Actual Color</h1>
<p> Can you tell a color from its hexadecimal RGB code value? Look at
the hexcode

Guess the color. To see what color it corresponds to, move the mouse
over the

Hex code. Moving the mouse out will display the color name. </p>
<table style = "width 50%; border-style; groove;
Text-align:center; font-family:monospace;
Font-weight:bold">
<caption>
<img src = "caption1.gif" id = "tableCaption" alt = "Table Caption"/>
</caption>
<tr>
    <td id = "Black">#000000</td>
    <td id = "Blue">#0000FF</td>
    <td id = "Magenta">#FF00FF</td>
    <td id = "Gray">#808080</td>
</tr>
<tr>
    <td id = "Green">#008000</td>
    <td id = "Lime">#00FF00</td>
    <td id = "Maroon">#800000</td>
    <td id = "Navy">#000080</td>
</tr>

```

```

<tr>
    <td id = "Olive">#8080000</td>
    <td id = "Purple">#800080</td>
    <td id = "Red">#FF0000</td>
    <td id = "Silver">#C0C0C0</td>
</tr>

<tr>
    <td id = "Cyan">#00FFFF</td>
    <td id = "Teal">#008080</td>
    <td id = "Yellow">#FFFF00</td>
    <td id = "White">#FFFFFFF</td>
</tr>
</table>
</body>

```

---

### 11.7. Form processing with onfocus and onblur

---

The onfocus and onblur events are particularly useful when dealing with form elements that Allow user input. The onfocus event fires when an element gains and onblur fires when an element loses focus, which occurs when another control gains the focus.

#### Example 6

```

<html>

<head>

<title> DHTML Event Model – onfocus and onblur</title>

<script type = "text/javascript">

Var helpArray =

[ " Enter your name in this input box,",

  "Enter your email address in this input box"+

  "in the format user@domain.",

]

```

```

Function helpText(messageNum)
{
    myForm.helpBox.value = helpArray [messageNum];
}
</script>
</head>
<body>
<form id = "myform" action = "">
Name : <input type = "text" name = "name" onfocus = "helpText(0)"
onblur = "helpText(6)" /> <br/>
Email : <input type = "text" name = "email" onfocus = "helpText(1)"
onblur = "helpText(6)" /> <br/>
Click here if you like this site
<input type = "checkbox" name = "like" onfocus = "helpText(2)" onblur =
"helpText(6)" /><br /> <hr/>
Any comments? <br/>
<textarea name = "comments" row = "5" cols = "45" onfocus =
"helpText(3)" onblur = "helpText(6)" /> <br/>
<input type = "submit" value = "Submit" onfocus = "helpText(4)" onblur =
"helpText(6)" />
<input type = "reset" value = "reset" onfocus = "helpText(5)" onblur =
"helpText(6)" />
<textarea name = "helpBox" style = "position: absolute;
This text area provides context-sensitive help.
</textarea>
</form>
</body>
</html>

```

---

## 11.8. Form processing with onsubmit and onreset

---

These events fire when a form is submitted or reset, respectively. The function form submit executes in response to the user submitting the form.

Example 7

```
<html>
<head>
<title> DHTML Event Model – onfocus and onblur</title>
<script type = "text/javascript">

Var helpArray =
[ " Enter your name in this input box,",
  "Enter your email address in this input box"+
  "in the format user@domain.",
]
Function helpText(messageNum)
{
    myForm.helpBox.value = helpArray [messageNum];
}
Function formSubmit() {
    Window.event.returnValue = false;
    If(confirm("Are you sure want to submit?"))
        Window.event.returnValue = true;
}
Function formReset() {
    Window.event.returnValue = false;
    If(confirm("Are you sure want to reset?"))
        Window.event.returnValue = true;
}
</script>
```

```

</head>

<body>

<form id = "myform" onsubmit = "formSubmit()" onreset = "formReset()"
action = ">

Name : <input type = "text" name = "name" onfocus = "helpText(0)"
onblur = "helpText(6)" /> <br/>

Email : <input type = "text" name = "email" onfocus = "helpText(1)"
onblur = "helpText(6)" /> <br/>

Click here if you like this site

<input type = "checkbox" name = "like" onfocus = "helpText(2)" onblur =
"helpText(6)" /><br /> <hr/>

Any comments? <br/>

<textarea name = "comments" row = "5" cols = "45" onfocus =
"helpText(3)" onblur = "helpText(6)"></textarea> <br/>

<input type = "submit" value = "Submit" onfocus = "helpText(4)" onblur =
"helpText(6)" />

<input type = "reset" value = "reset" onfocus = "helpText(5)" onblur =
"helpText(6)" />

<textarea name = "helpBox" style = "position: absolute; right: 0 top:0
readonly = "true".rows = "4" cols = "45">

This text area provides context-sensitive help.

</textarea>

</form>

</body>

</html>

```

---

### 11.9. Event Bubbling

---

Event bubbling is the process whereby events fired in child elements “bubble” up to their parent elements. When a child event is fired, the event is first delivered to the child’s event handler, then to the parent’s event handler. This might result in event handling that was not intended. If you intend to

handle an event in a child element, you might need to cancel the bubbling of the event in the child element's event-handling code by using the `cancelBubble` property of the event object.

#### Example 8

```
<html>
<head>

<title> DHTML Event Model – Event Bubbling</title>
<script type = "text/javascript">
Function documentClick()
{
    Alert("You clicked in the document");
}
Function paragraphClick(value)
{
    Alert("You clicked the text");
    If (value)
        Event.cancelBubble = true;
}

    Document.onClick = documentClick;
</script>
</head>
<body>
<p onclick = "paragraphClick(false)"> Click here</p>
<p onclick = "paragraphClick( true) "> Click here, too</p>
</body>
</html>
```

---

## 11.10. Let us Sum Up

---

Dynamic HTML the pages can be controlled by the event models. This makes the web application more responsive and user friendly and can reduce server load. With the event model, scripts can respond to a user who is moving the mouse, scrolling up or down the screen or entering keystrokes. Content becomes more dynamic while interfaces become more intuitive.

### **Event onclick**

One of the most common events is onclick. When the user clicks a specific item with the mouse, the onclick event fires. Using the for property of the script element allows you to specify the element to which the script applies.

### **Event on load**

The onload event fires whenever an element finishes loading successfully. Frequently, this event is used in the body element to initiate a script after the page loads into the client. The script called by the onload event updates a timer that indicates how many seconds have elapsed since the document was loaded.

### **Error Handling with onerror**

The web is a dynamic medium. Sometimes a script refers to objects that existed at a specified location when the script was written, but the location changes at a later time, rendering the script invalid. The error dialog presented by the browser in such a case can be confusing to the user. To prevent this dialog box from displaying and to handle errors more elegantly, scripts can use the onerror event to execute specialized error handling code. To prevent this dialog box from displaying and to handle errors more elegantly, scripts can use the onerror event to execute specialized error handling code. The function handleError should execute when an onerror occurs in the window object. The following example misspelled function name alrrt create an error and informed by the function handleError.

### **Tracking the Mouse with Event onmousemove**

Event onmousemove fires repeatedly whenever the user moves over the Web page.

### **Rollovers with onmouseover and onmouseout**

When the mouse cursor moves over an element, an onmouseover event occurs for that element. When the mouse cursor leaves the element, an onmouseout event occurs for that element. If the image is large or the connection is slow, downloading causes a noticeable delay in the image update.

### **Form processing with onfocus and onblur**

The onfocus and onblur events are particularly useful when dealing with form elements that allow user input. The onfocus event fires when an element gains and onblur fires when an element loses focus, which occurs when another control gains the focus.

### **Form processing with onsubmit and onreset**

These events fire when a form is submitted or reset, respectively. The function form submit executes in response to the user submitting the form.

### **Event Bubbling**

Event bubbling is the process whereby events fired in child elements “bubble” up to their parent elements. When a child event is fired, the event is first delivered to the child’s event handler, then to the parent’s event handler. This might result in event handling that was not intended. If you intend to handle an event in a child element, you might need to cancel the bubbling of the event in the child element’s event-handling code by using the cancelBubble property of the event object.

---

## **11. 11. Lesson End Activities**

---

1. What is the difference about onClick and onload event?
2. How to track the mouse position?

---

## **11.12. Check your progress**

---

1. Write a Javascript program to print a message for mouse over an image.
2. Write a JavaScript program for form processing with onsubmit and onreset.

---

## **11.13. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J. Deitel and A.B. Goldberg, Prentice Hall.
2. [www.w3c.com](http://www.w3c.com)



## Java Script - Filters and Transitions

---

### Contents

- 12.0.Aim and Objective
- 12.1. Introduction
- 12.2. Flip Filters
- 12.3.Chroma filter
- 12.4.Image masks
- 12.5.Image filters
- 12.6.Invert Filter
- 12.7.Gray Filter
- 12.8.Adding Shadows to text
- 12.9.Creating gradients with alpha
- 12.10.Making Text glow
- 12.11.Creating Motion with blur
- 12.12.Wave Filter
- 12.13. Let us Sum UP
- 12.14. Lesson End Activities
- 12.15. Check your progress
- 12.16. Reference

---

### 12.0. Aim and Objective

---

- To use filters to achieve special effects
- To be able to create animated visual transitions
- To be able to modify filters dynamically, using DHTML

---

### 12.1.Introduction

---

Filters and transitions are specified with the CSS filter property. Applying filters to text and images causes changes that are persistent. Transitions are temporary; applying a transition allows you to transfer from one page to another with a pleasant visual effect, such as random dissolve. Filter and Transistion do not add content to your pages, rather, they present existing

content in an engaging manner to capture the user's attention. Each of the visual effects achievable with filters and transitions is programmable, so these effects can be adjusted dynamically by programs that respond to user-initiated events, such as mouse click and keystrokes.

The images used in this chapter are :

Flood.jpg



Sunface.gif



Slash.gif



---

## 12.2. Flip Filters : flipv and fliph

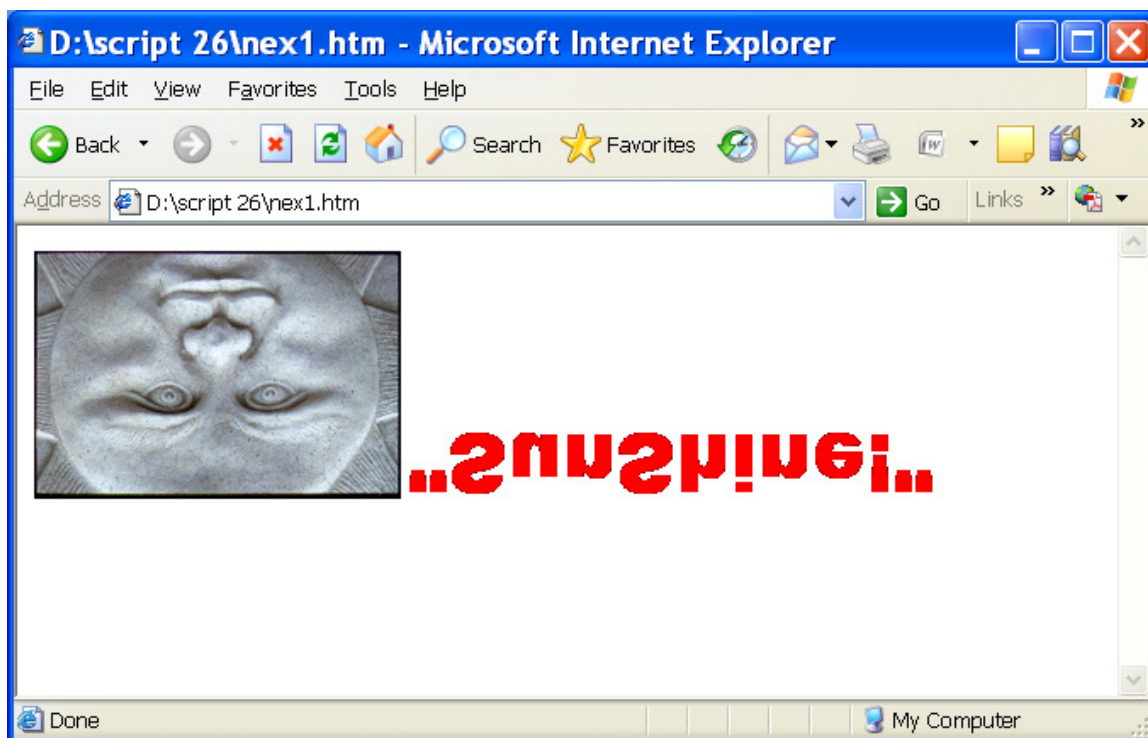
---

The flipv and fliph filters mirror text or images vertically and horizontally. The following example demonstrates this.

Example

```
<html>
<head>
<title> Flip Filter</title>
    
    <span style="width: 300; height: 50; font-size: 36pt; font-family: Arial
    Black; color: red; Filter: FlipV">"SunShine!" </span>
</body>
</html>
```

Output :



If the filter is fliph, which flips the affected object horizontally and if flipv then it affected the object vertically.

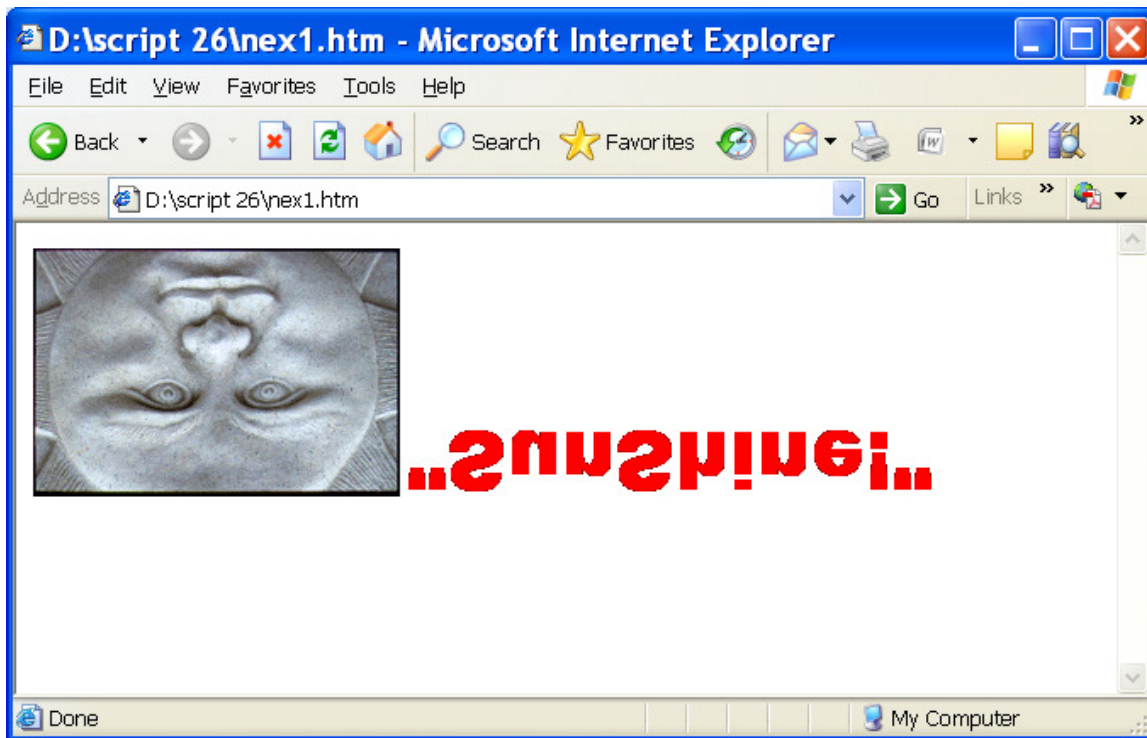
### 12.3. Chroma filter

The Chroma filter applies transparency effect dynamically, without using a graphics editor to hard-code transparency into the image. From the output of the following example the transparency of an image, using object model scripting based on a user selection from a select element.

Example

```
<html>
<head>
<title> Chroma Filter</title>
<style type ="text/javascript">
    
    "SunShine!" <span style="width: 580; height: 50; font-size: 36pt; font-
family: Arial Black; color: red; Filter: Chroma(Color =
#FF0000)">"SunShine!"</span>
</body>
</html>
```

Output :



Each filter has a property named `enabled`. If this property is set to `true`, the filter is applied. If it is set to `false`, the filter is not applied. `Onchange` event fires whenever the value of a form field changes. The expression `this.value` represents the currently selected value in the select GUI component, which is passed to function `changeColor`.

---

## 12.4 image masks

---

Applying the mask filter to an image allows you to create an effect in which an element's background is a solid color and its foreground is transparent, so the image or color behind it shows through. This is known as an image mask. The following example adds the mask filter to a `div` element which overlaps an image. The foreground of the `div` element is transparent, so you can see the background image through the letters in the foreground. Parameters always specified in the format `param = value`.

Example

```
<html>

<head>

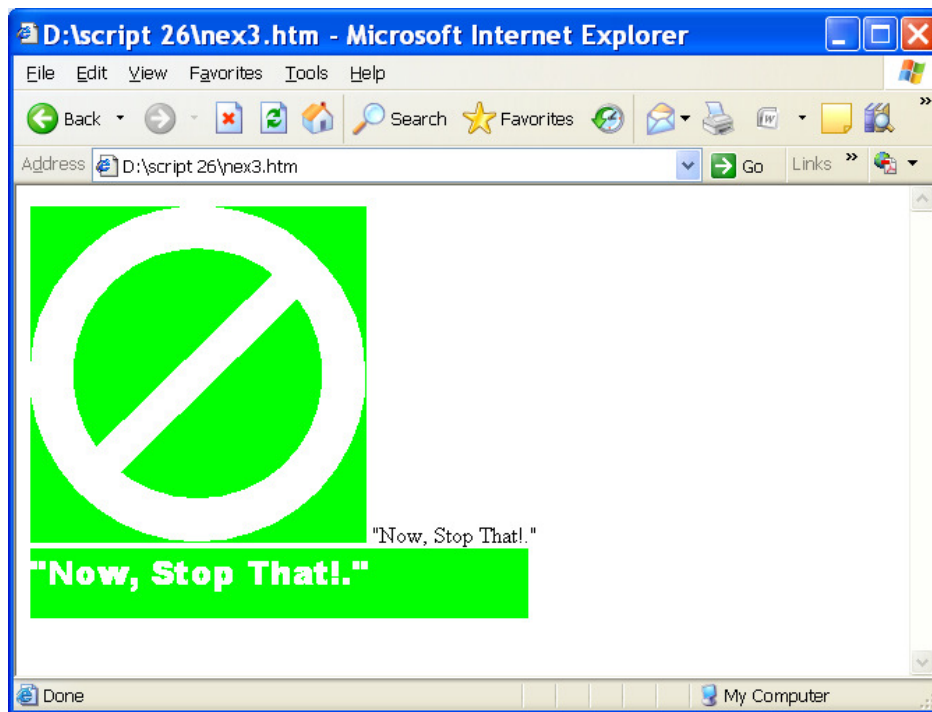
<title>Mask Filter</title>

    
    "Now, Stop That!."
    <span style="width: 357; height: 50; font-size: 18pt; font-family: Arial
    Black; color: red; Filter: Mask(Color=#00FF00)">"Now, Stop
    That!."</span>

</body>

</html>
```

Output :



---

## 12.5. Image filters : invert, gray and xray

---

These filters apply simple image effect to image or text. The invert filter applies a negative image effect – dark areas become light and light areas become dark. The gray filter applies a grayscale image effect, in which all color is stripped from the image and all that remains is brightness data. The Xray filter applies an x-ray effect, which basically is an inversion of the grayscale effect.

The following example illustrate in detail.

### **Xray Filter**

Example

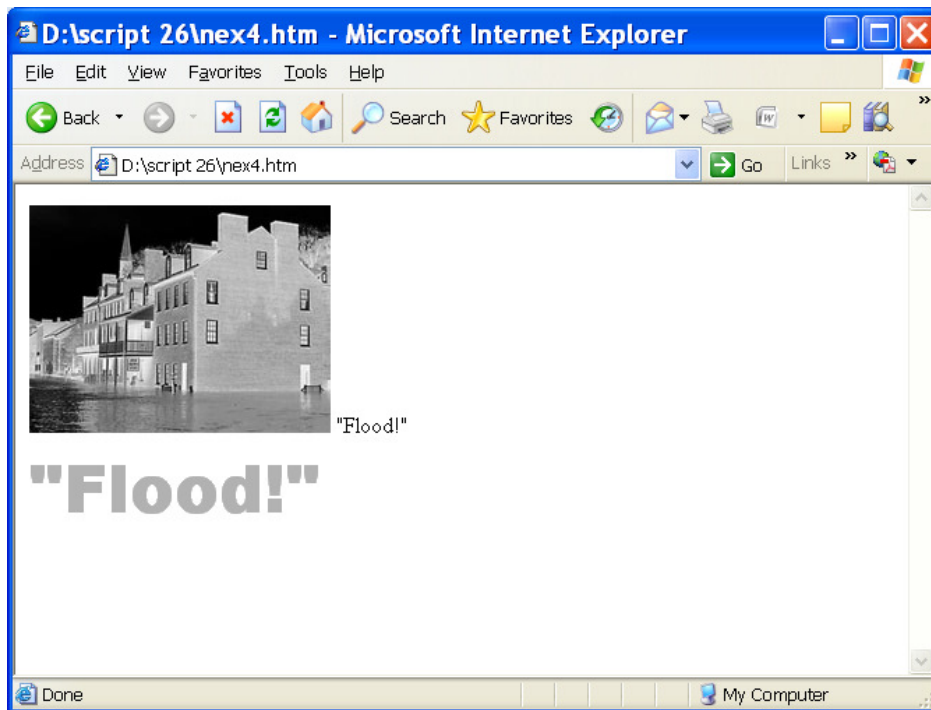
```
<html>
<head>
<title> Xray Filter</title>

"Flood!"
<span style="width: 357; height: 50; font-size: 36pt; font-family: Arial
Black; color: red; Filter: Xray">"Flood!" </span>
```

</head>

</html>

Output:



---

## 12.6. Invert Filter

---

Example :

<html>

<head>

<title> invert Filter </title>



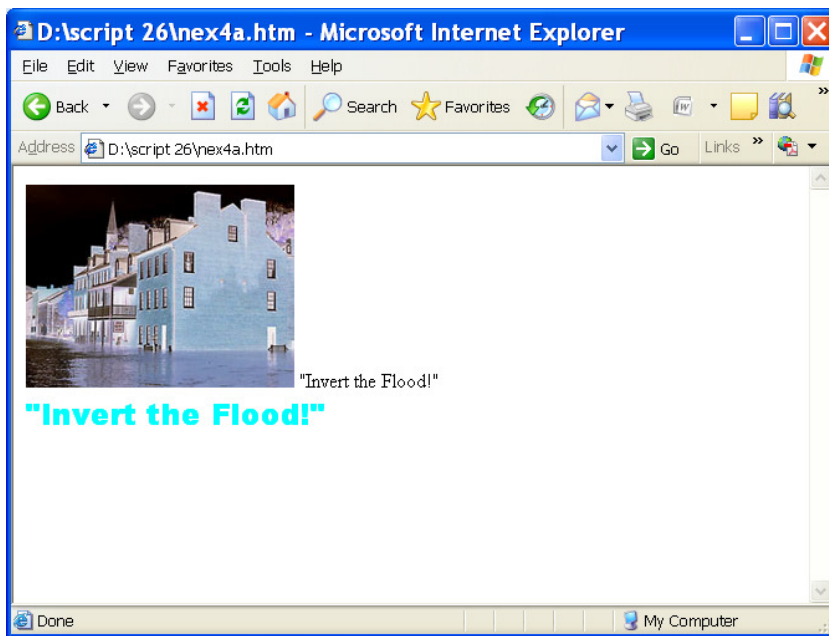
"Invert the Flood!"

<span style="width: 387; height: 50; font-size: 18pt; font-family:  
Arial Black; color: red; Filter: Invert">"Invert the Flood!"</span>

</head>

</html>

Output :




---

## 12.7 Gray Filter

---

Example

```
<html>
```

```
<head>
```

```
<title> Gray Filter</title>
```

```
    
```

```
    "Colorless Flood"
```

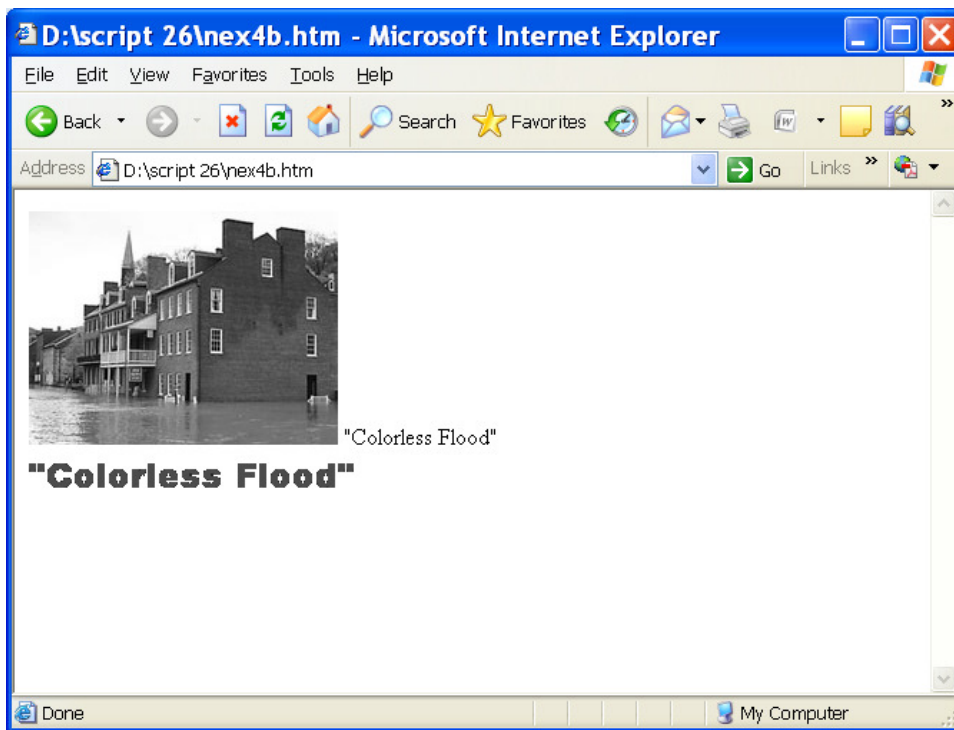
```
    <span style="width: 300; height: 50; font-size: 18pt; font-family: Arial  
    Black; color: red; Filter: Gray">"Colorless Flood" </span>
```

```
</head>
```

```
</html>
```

Output :





## 12.8. Adding Shadows to text

A simple filter that adds depth to your text is the shadow filter. This filter create a shadowing effect that gives your text a three-dimensional appearance.

Example

```
<html>
<head>
<title> Shadow Filter </title>

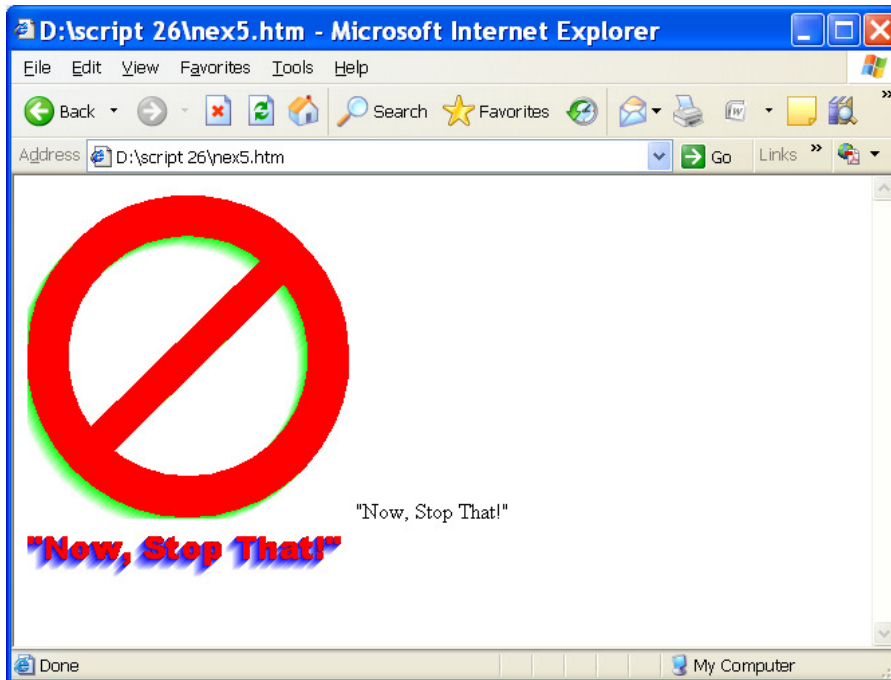


"Now, Stop That!"

<span style="width: 357; height: 50; font-size: 18pt; font-family: Arial
Black; color: red; Filter: Shadow(Color=#0000FF, Direction=225)">"Now,
Stop That!"</span>

</head>
</html>
```

Output :



Property direction of the shadow filter determines in which direction the shadow effect is applied – this can be set to one of eight direction expressed in angular notation

- 0 up
- 45 above-right
- 90 right
- 135 below-right
- 180 below
- 225 below-left
- 270 left
- 315 above-left

Property color specifies the color of the shadow that is applied to the text.

---

## 12.9. Creating gradients with alpha

---

Alpha filter used to get gradient effect. It also used to transparency effects notachievable with the chroma filter.

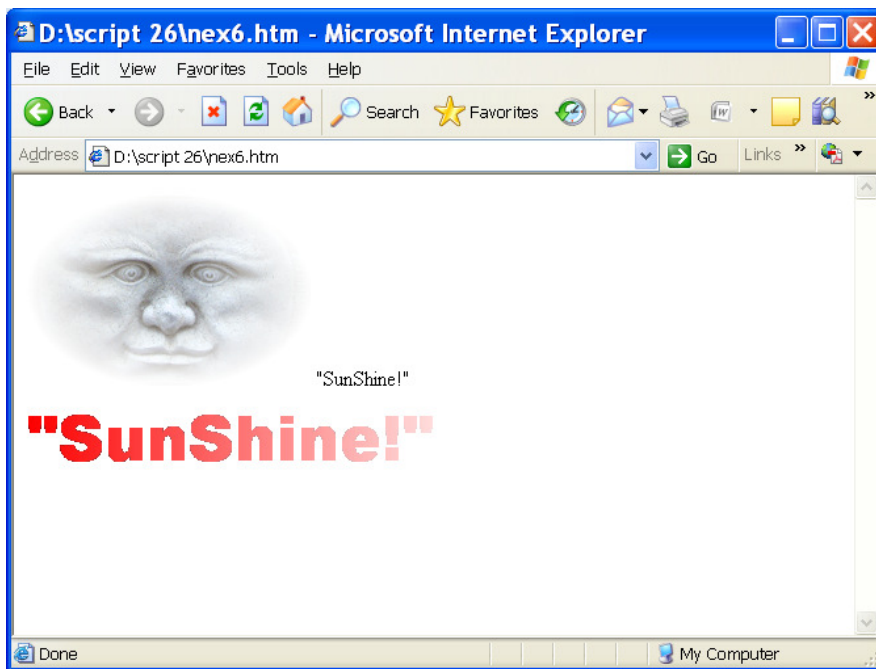
The style property of the filter determines in what opacity style is applied. Opacity refers to the color saturation of an image. The various style values create different transitions from opaque to transparent. A style value of 0 applies uniform opacity, a value of 1 applies a linear gradient, a value of 2 applies a circular gradient and a value of 3 applies a rectangular gradient.

The opacity and finishopacity properties are both percentages that determine at what percent opacity the specified gradient starts and finishes, respectively. Additional attributes are startX, startY, finishX and finishY. These specify at what x-y coordinates the gradient starts and finishes in that element.

Example

```
<html>
<head>
  
  "SunShine!"
  <span style="width: 357; height: 50; font-size: 36pt; font-family: Arial
  Black; color: red; Filter: Alpha(Opacity=100, FinishOpacity=0, Style=1,
  startX=0, startY=0, FinishX=580, FinishY=0)">"SunShine!"</span>
</head>
</html>
```

Output :




---

### 12.10. Making Text glow

---

The glow filter adds an aura of color around text. The color and strength can both be specified.

---

### 12.11. Creating Motion with blur

---

The blur filter creates an illusion of motion by blurring text or images in a certain directions.

Example :

```
<html>
```

```
<head>
```

```
<title> Blur Filter </title>
```

```

```

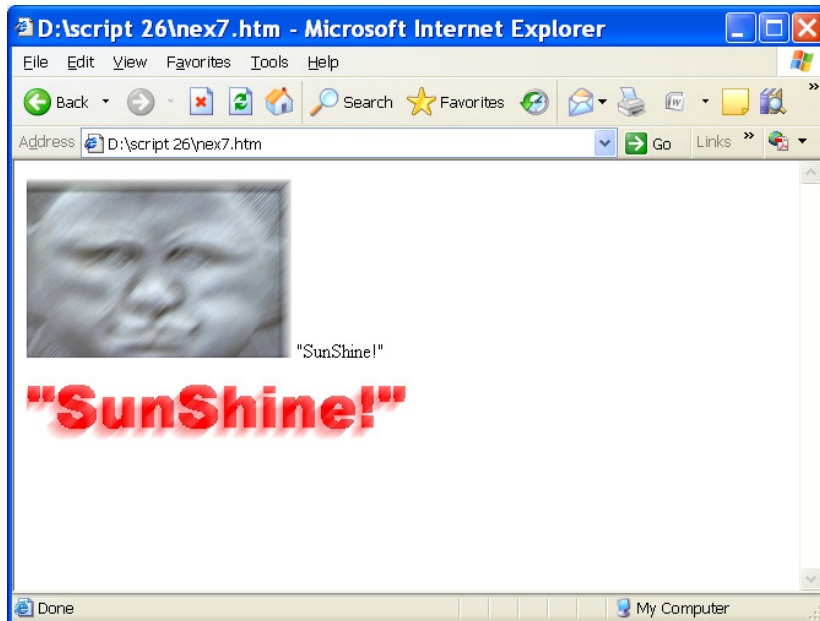
```
"SunShine!"
```

```
<span style="width: 357; height: 50; font-size: 36pt; font-family: Arial Black; color: red; Filter: Blur(Add = 1, Direction = 225, Strength = 10)">"SunShine!"</span>
```

</head>

</html>

Output :



---

## 12.12. Wave Filter

---

Example

<html>

<head>

<title> Wave Filter</title>



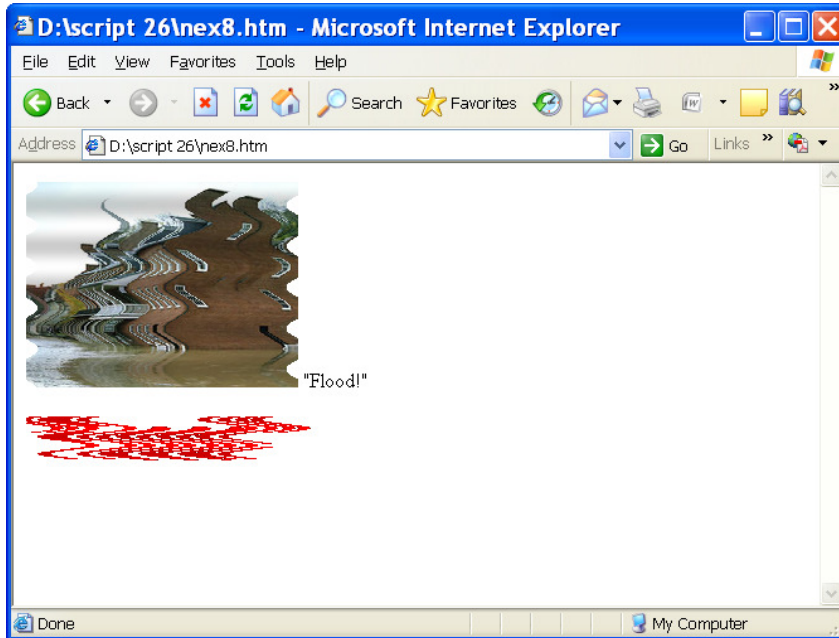
"Flood!"

<span style="width: 357; height: 50; font-size: 36pt; font-family: Arial Black; color: red; Filter: Wave(Add=0, Freq=5, LightStrength=20, Phase=20, Strength=20)">"Flood!"</span>

</head>

</html>

Output:



---

### 12.13. Let us Sum UP

---

Filters and transitions are specified with the CSS filter property. Applying filters to text and images causes changes that are persistent. Transitions are temporary; applying a transition allows you to transfer from one page to another with a pleasant visual effect, such as random dissolve. Filter and Transition do not add content to your pages, rather, they present existing content in an engaging manner to capture the user's attention. Each of the visual effects achievable with filters and transitions is programmable, so these effects can be adjusted dynamically by programs that respond to user-initiated events, such as mouse click and keystrokes.

#### **Flip Filters : flipv and fliph**

The flipv and fliph filters mirror text or images vertically and horizontally.

#### **Chroma filter**

The Chroma filter applies transparency effect dynamically, without using a graphics editor to hard-code transparency into the image. From the output of the following example the transparency of an image, using object model scripting based on a user selection from a select element.

## **Image masks**

Applying the mask filter to an image allows you to create an effect in which an element's background is a solid color and its foreground is transparent, so the image or color behind it shows through. This is known as an image mask. The following example adds the mask filter to a div element which overlaps an image. The foreground of the div element is transparent, so you can see the background image through the letters in the foreground. Parameters always specified in the format param = value.

## **Image filters : invert, gray and xray**

These filters apply simple image effect to image or text. The invert filter applies a negative image effect – dark areas become light and light areas become dark. The gray filter applies a grayscale image effect, in which all color is stripped from the image and all that remains is brightness data. The Xray filter applies an x-ray effect, which basically is an inversion of the grayscale effect.

## **Adding Shadows to text**

A simple filter that adds depth to your text is the shadow filter. This filter create a shadowing effect that gives your text a three-dimensional appearance.

## **Creating gradients with alpha**

Alpha filter used to get gradient effect. It also used to transparency effects notachievable with the chroma filter.

## **Making Text glow**

The glow filter adds an aura of color around text. The color and strength can both be specified.

## **Creating Motion with blur**

The blur filter creates an illusion of motion by blurring text or images in a certain directions.

---

### **12.14. Lesson end Activities**

---

1. Write a program for wave filter.
2. Write a program for Text glow.

---

**12.15. Check your Progress**

---

1. List all the filters available in JavaScript.

---

**12.16. Reference.**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.w3c.com](http://www.w3c.com)



---

## Lesson 13

# VBScript - I

---

### Content

#### 13.0. Aim and Objective

#### 13.1. Introduction

#### 13.2. Statements and Expressions

#### 13.3. Variables & Procedure

#### 13.4. Data Types

#### 13.5. Operators

#### 13.6. Control Structure

#### 13.7. Arrays

#### 13.8. Let us Sum Up

#### 13.9. Lesson End Activities

#### 13.10. Check Your Progress

#### 13.11. Reference

---

#### 13.0. Aim and Objective

---

- The differences between Visual Basic and VBScript
- The syntax of the VBScript language
- Program flow in VBScript
- Interaction between the program and the user in VBScript

---

#### 13.1. Introduction

---

**VBScript** is derived from **Visual Basic**, which has origins in **GW-Basic** that was shipped with all early versions of MS-DOS. GW-Basic evolved from the original Dartmouth BASIC (Beginners All-purpose Symbolic Instruction Code) developed in the 1960's.

VBScript is an interpreted language - sort of. Like many other contemporary "interpreted" languages.

VBScript is a scripting language for HTML pages on the World Wide Web and corporate intranets. If you know Visual Basic, you'll probably not have any trouble at all with VBScript.

---

### 13.2. Statements and Expressions

---

An *expression* is a chunk of VBScript syntax in the form of a calculation that uses operators to combine numbers, strings, and variables and evaluate them to some value. An expression can be used any place where a variable is used.

In VBScript statements are terminated at the end of a line when you hit the **Enter** key and your text editor inserts the Carriage Return (ASCII 13) and LineFeed (ASCII 10) characters. If you have a statement that is too long to fit on one line without scrolling off the right edge of the screen you can use the underscore as a *line continuation character*. For line continuation the underscore must be the very last character on the line without even a space following it, and it must be between keywords, variable names, operators, and quoted strings. Multiple statements separated by a colon can be included in a single line.

---

### 13.3. Variables and Procedures

---

Unlike languages that define types and sizes of numbers (integers, floating point, etc.), characters, strings, etc. as distinct data types, VBScript treats all data as one of two fundamental types: numbers or strings. Most of the time VBScript figures out how to make the conversions based on context. When you need explicit control over data types VBScript has conversion functions and data type functions that return the type of variables and expressions.

Variables are used to store data items that can change (e.g. "vary") during the lifetime of a program. All variables have one variable type called *variant*. The data is actually stored in numbered memory addresses in the computer's Random Access Memory (RAM), however variables in high-level languages like VBScript provide convenient names for the memory addresses.

Variables and memory addresses are similar to a house where you deliver the newspaper: it might have a lot number, plot number, section number, and township number for the official county engineer records; but you know it as "123 Main St.," "Mr. Smith's house," or maybe "the big red house on the corner with the mean dog."

Variables in VBScript do not have to be "declared" before they are used, they spring to life when you first use them. This might sound convenient, but it is now generally considered a bad idea, although it persists for historical reasons and backward compatibility with earlier versions of Basic. VBScript supports the **Dim** statement, however, that allows you to explicitly declare the variables you intend to use. The **Option Explicit** is also supported, which *forces* you to use the **Dim** statement for all variables. When used, **Option Explicit** must be the first executable statement in the program.

VBScript variable names are *not case sensitive*. The variables **TotalPrice**, **totalprice**, and **TOTALPRICE** all refer to the same variable. Variable names can be made up of letters, digits, and the underscore character only (no spaces); must begin with a letter; and cannot be the same as VBScript built-in key words (such as **For**, **While**, etc.).

If *variables* are used to store data items that can change then it should follow that *constants* are used to store data items that cannot change. VBScript supports the **Const** key word to declare *symbolic constants*. A common naming convention borrowed from languages like C and Java (which are case sensitive) is to use all upper case for symbolic Constants. Remember, however, that VBScript is not case sensitive.

You should understand some simple concepts before you begin to program. Among these concepts are variables and procedures.

A program is made up of one or more procedures. A procedure is an instruction block in VBScript. Regular procedures, or Subs, simply act on data, but a special procedure called a Function returns a value to the procedure that called it. Be aware that all your scripts will be made up of procedures and Function blocks.

VBScript programs will begin with a `<SCRIPT>` tag and end with a `</Script>` tag. Your procedures and Functions will work the same way. The procedure is created and it is ended. Your program code goes in between.

The first thing you'll notice in the VBScript code block is the

```
<SCRIPT LANGUAGE="VBS">
```

Keep in mind that you must close out your script code block with a `</SCRIPT>` tag. If you don't add this tag, your script usually won't run and you won't know what's going on because you aren't going to see any error messages. Errors like this are tough to track down because if your code block is large, you might start hacking away at perfectly good code to see if you can figure out what's wrong. An error like this can waste a lot of time because nothing you try works.

The next tag that you should get used to adding to a script block is the comment tag. You won't run into a problem with most browsers if you don't add this tag, but it's a good idea to get used to adding it anyway. If nothing else, this tag helps you to see your scripting blocks more easily in your HTML documents. Like the `<script>` tag, the comment tag should have a closing tag, although not having a closing tag will not affect your script in a browser that recognizes the `<SCRIPT>` tag.

Procedures make up most of the rest of a script code block. They have opening and closing lines and are accessible from one another. The first procedure run in a script depends on what events take place within the HTML page. An event is usually an action taken by a user, such as clicking on a button, but other events can also be triggered within an HTML page. Other possible events include timer events, mouse movement, and messages sent from controls.

To review, an HTML script is enclosed by the `<SCRIPT>` `</SCRIPT>` tag pair. The `LANGUAGE="VBS"` attribute sets the scripting language to VBScript. The VBScript script code should be written inside a comment tag, `<!-- -->`. Procedures are blocks of code within the script that make up the program.

---

## 13.4. Data Types

---

A variable is a name that represents the contents of some space in memory. Thinking of a variable in terms of memory space will help you to understand an important concept in programming: data types.

All variables in VBScript are of the type variant. A variant is a variable that is simply the name for a piece of data. The variant type doesn't differentiate between types of data until the time comes to process that data. This means that a variable can represent literally any value or type of value.

### Subtypes of Variant Types

The variant type has a number of subtypes. Let's go through each of the subtypes and discuss how a variable of any given subtype will take up space in memory.

#### Boolean

One of the most basic data types in programming is the Boolean data type. This subtype in VBScript can have a value of either true or false. The Boolean type takes up very little space in memory.

#### Byte

The byte subtype can be a whole, positive number in the range of 0 to 255. Like the Boolean subtype, the byte subtype takes up very little space in memory.

#### Integer

The integer subtype takes up 2 bytes in memory and can be an integer value in the range of -32,768 to 32,767. An extra byte of storage makes a big difference in the value that a variable can hold.

#### Long

The long variable subtype is 4 bytes in size and can be a whole number in the range of -2,137,483,648 to 2,137,483,647.

#### Single

The single subtype contains a single-precision, floating-point number. Precision refers to the number of bytes of fractional value storage allotted to the variable. A single-precision number allots 2 bytes

for fractional value storage. It takes 4 bytes to store a variable of the subtype single. The range of values that a single can hold is -3.402823E38 to -1.401298E-45 for negative values and 1.401298E-45 to 3.402823E38 for positive values.

### **Double**

The double subtype takes up 8 bytes of storage, 4 bytes of which are reserved for a fractional value. The double subtype is extremely precise and is usually reserved for advanced mathematical and scientific operations. It has a range of -1.79769313486232E308 to -4.94065645841247E-324 for negative values and 4.94065645841247E-324 to 1.7976931348632E308 for positive values.

### **String**

The string subtype is a group of up to approximately 2 billion characters. The size of the string variable depends on the length of the string.

### **Date**

The date subtype is a number that represents a date in the range of January 1, 100 to December 31, 9999.

### **Empty**

The empty subtype is returned for variants that aren't initialized. If the variable is a number, the value is 0. If it's a string, the value is "".

### **Object**

The object subtype contains the name of an OLE Automation object. The object can then be manipulated using the variable.

### **Error**

The error subtype contains an error number. You can use the generated error number to generate an expanded explanation of the error.

## **Using Variables**

Now that you know what variables are all about, let's take a look at how they are used in your VBS scripts.

## Declaring Your Variables

Strictly speaking, you don't need to declare your variables in VBScript. You could simply set the variables that you need on the fly:

```
MyString="This is my string"
```

One of problems with setting the value of variables without first declaring them is that the variables become difficult to track. For example, you set the value of a variable somewhere in code, but you can't remember where the variable started. In addition to just being good programming practice, declaring your variables will make it easier for you to read and maintain your code.

To declare a variable in a VBS script, you use the Dim statement. Dim stands for dimension:

```
<SCRIPT LANGUAGE="VBS">
```

```
<!--
```

```
Option Explicit
```

```
Dim MyString
```

```
MyString="This is my string"
```

```
-->
```

```
</SCRIPT>
```

Notice that in addition to declaring our variable with the Dim statement, we added the Option Explicit statement to the beginning of the code. Adding Option Explicit will require you to declare all variables in your script. Using this statement is completely up to you. If you use Option Explicit but don't declare a variable, your script will generate an error when run.

You can declare more than one variable at a time. Just use the Dim statement and put a comma between every new variable name:

```
<SCRIPT LANGUAGE="VBS">
```

```
<!--
```

```
Dim Name, Address, City, State, Zip
```

```
-->
```

```
</SCRIPT>
```

## Assignment

To assign a value to a variable, place the variable on the left followed by an equals sign and the value on the right:

```
Name="Ramesh Thanappan"
```

Remember that variables in VBScript are variants. VBScript determines the nature of the variable when you run the script. There are really only two types of data: strings and numbers. String data is always held in quotation marks.

## Scope

The scope of a variable refers to where in the script the variable is available for processing. A variable declared inside a procedure is limited in scope to that procedure. If a variable is declared outside of the procedures in the script, it is available to all the procedures in the script. Listing 2.3 illustrates the scope of variables in VBS scripts.

### Scope of variables in VBS scripts.

```
<HTML>
<HEAD>
  <TITLE>VBS Variable Scope</TITLE>
</HEAD>
<BODY>
<H1>Tester Page for VBS</H1>
<HR COLOR="BLUE">
  <INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Local
Variable">
  <INPUT TYPE="SUBMIT" NAME="Btn2" VALUE="Script Wide
Variable">
  <INPUT TYPE="SUBMIT" NAME="BTN3" VALUE="Out of Scope">
<SCRIPT LANGUAGE="VBS">
Option Explicit
Dim MyGlobal
```



```

MyGlobal="Access Ok"
Sub Btn1_OnClick()
    Dim MyLocal
    MyLocal="Local access Ok!"
    MsgBox MyLocal,0,"Btn1 Clicked"
End Sub
Sub Btn2_OnClick
    MsgBox MyGlobal,0,"Btn2 Clicked"
End Sub
Sub Btn3_OnClick
    MsgBox MyLocal,0,"Btn3 Clicked"
End Sub
-->
</SCRIPT>
</BODY>
</HTML>

```

If you run the code, you'll see that you can access the local variable by clicking Btn1. If you click Btn2, you'll access the variable that we declared globally. Finally, if you click Btn3, you'll get an error because the Btn3\_OnClick procedure tries to access the variable declared in the Btn1\_OnClick procedure.

When discussing the scope of a variable, you'll often hear the term lifetime. Lifetime refers to the amount of time that the variable exists in memory. At the procedure level, the lifetime of the variable is as long as it takes to run through the procedure. At the script level, the variable is live for as long as the script is live. You can extend the lifetime of a procedure-level variable by declaring the variable as static. A static variable will retain its value between calls to the procedure. To declare a variable as static, simply use the Static keyword:

```
Static MyVariable
```

## Scalar

The types of variables we have talked about so far have been scalar variables. A scalar variable has only one value assigned to it at a time. Depending on the types of programs you write, you'll probably use scalar variables often in your scripts. When modeling the world, though, you often need to work with sets of values. To work with these values, you need to use a different type of variable known as an array.

## Naming Conventions

The names that you give variables in VBScript are the same as for any other named item in a script. Variable names must begin with an alphabetical character:

```
' This is Ok
```

```
Dim MyVariable
```

```
' This is not Ok
```

```
Dim @Variable
```

In addition, variables cannot contain embedded periods and are limited to 255 characters. Finally, you cannot use the same name for two different values in the same scope. It's okay to have the same variable name declared in many different procedures, but the same variable name cannot be declared globally, and it cannot be declared twice in the same procedure.

When you declare variables, you should consider a few naming conventions. Table 13.1 lists these conventions.

**Table 13.1. Naming conventions in VBScript.**

<i>Variable Subtype</i>	<i>Prefix</i>
Boolean	bln
Byte	byt
Date	dtm
Double	dbl

Error	err
Integer	int
Long	lng
Object	obj
Single	sng
String	str

To use these naming conventions properly, you need to know the probable data subtype of the variables you declare. To name a variable that you know will contain a string, use the prefix along with a descriptive name for the variable:

```
Dim strName
```

The same rules follow for values that you know will be of a specified subtype:

```
Dim dblMiles
```

```
Dim blnEmpty
```

```
Dim intTotal
```

Using a convention such as this can save you time in the long run. It's much easier to see that you're about to make a mistake when you see an equation like `strValue + dblMiles` than if you simply saw `Value + Miles`.

The scope of a variable also has a naming convention. If you have a script-level variable, you can add an `s` to the prefix:

```
Dim sdblMyNumericValue
```

We'll talk more about coding conventions throughout the book. As new conventions are introduced, they will be incorporated into the sample code.

## Constants

A constant is a named value that doesn't change throughout a script. For example, the value of the speed of light through room-temperature air at sea level is about 760 miles per hour. To use this constant in a script, you simply declare it, assign it, and use it like a variable. To differentiate a value that you

want to be a constant in your script, you should use your own naming conventions, so that you don't try to reassign the value of the constant that you've created.

```
'vbSOL Speed of Light
```

```
Dim vbSol
```

```
vbSol = 760
```

Once you set the value of the constant, you can use it in your script as if it were the actual number. Keep in mind though that constants in VBScript are essentially just variables that you don't change in your program. Unlike other languages that allow you to set the value of a constant to static, there's no way to make a variable in VBScript unchangeable.

---

## 13.5 Operators

---

VBScript uses many familiar or intuitive operators, such as **+**, **-**, **\***, and **/** for addition, subtraction, multiplication, and division. VBScript has an additional division operator, **\**, which performs integer division.  $8 / 5$  is 1.6, where  $8 \setminus 5$  is 1. The exponentiation operator is **^**. An operator supported in virtually all programming languages that may be new is the *modulus division* operator. Modulus division returns the remainder rather than the quotient. In VBScript the modulus operator is **MOD**.

VBScript has one operator for strings, the **&** (ampersand) operator to *concatenate* (combine) strings. VBScript supports many additional operations for strings implemented by functions. For example, you can extract parts of a string with **Mid()**; convert strings to upper and lower case with **UCase()** and **LCase**; remove leading white space, trailing white space, or both with **LTrim()**, **RTrim()**, and **Trim()**; and convert strings to various data types with **CInt()** (integer), **CDBl()** (double precision), **CCur()** (currency), etc.

Comparisons are tested with the familiar **<** and **>** symbols used in mathematics for "less than" and "greater than". "Less than or equal to" and "greater than or equal to" use the **<=** and **>=** operators respectively. Equality is tested with the **=** operator and inequality ( "not equal to" ) is tested with the **<>** operator. Note that **=** is used for both assignment *and* test for equality; VBScript knows the difference based on context.

Table 13.2 contains the operators you can use in VBScript programs.

**Table 13.2. VBScript operators.**

<i>Operator</i>	<i>Purpose</i>
+	Addition
And	Logical And
&	Concatation operator
/	Division
Eqv	Logical Equivalence
^	Exponential
Imp	Logical Implication
\	Integer Division
Is	Logical Is (Refer to same object)
Mod	Modulus Operator (Remainder)
*	Multiplication
-	Negation and Subtraction
Not	Logical Not
Or	Logical Or
Xor	Logical Xor

You'll be using some of the logic operators in the next sections. The mathematical operators are used for math operations and, in some cases, concatenation. For example:

MyString = "Now is the time" & " for all good men..."

is functionally equivalent to

MyString = "Now is the time" + " for all good men..."

## **13.6 Control Structure**

### **Decision-Making in Programs**

Decision-making in programs is what programming is all about. Keep in mind that computers aren't really that good at thinking things through. You have to tell the program what to do every step of the way. Humans use a method of thinking that has what are known as fuzzy logic characteristics. If you were to ask more than one person to name the color of a particular scarf, you might get the answers red, pink, orange, and scarlet. If you ask a computer to name the color of an element on the screen, you'll get back the exact color name. Taking it one step further, if you ask the same group of people if the scarf is red, they all might answer "Yes." If you ask a computer if the screen element is red, it will return a yes value only if the item is exactly red.

Always keep in mind the limitations of the machine. Write good questions and you'll get good answers.

#### **If...Then...Else**

The If...Then...Else statement is one of the basic capabilities of a computer language. This statement tests for the truth of a statement and, depending on the answer, processes information or moves on to another piece of code. Let's look at an example of an If...Then...Else statement in a program and then talk about what's happening in the code.

#### **If...Then...Else in a VBS script.**

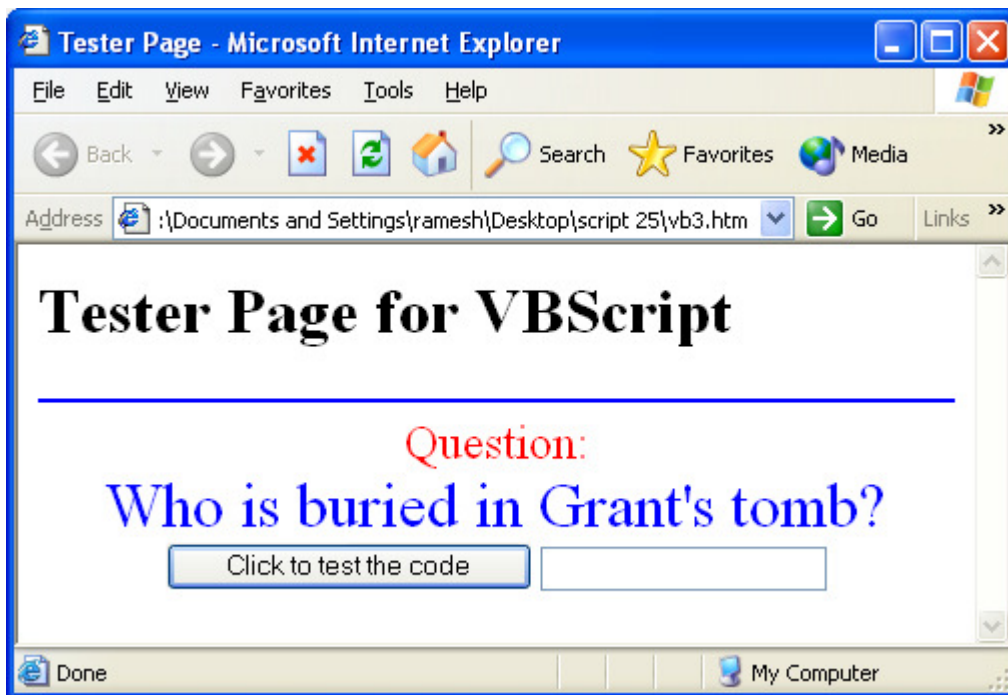
```
<HTML>
<HEAD>
<TITLE>Tester Page</TITLE>
</HEAD>
<BODY>
<H1>Tester Page for VBScript</H1>
<HR COLOR="BLUE">
```

```

<CENTER>
<FONT COLOR=RED SIZE=5>Question:</FONT><BR>
<FONT COLOR=BLUE SIZE=6>Who is buried in Grant's tomb?</FONT><BR>
<INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Click to test the code">
<INPUT TYPE="TEXT" NAME="Txt1">
</CENTER>
<SCRIPT LANGUAGE="VBSCRIPT">
<!--
Sub Btn1_OnClick()
Dim Message
If Txt1.Value = "Grant" then
Message="You're right!"
Else
Message="Try again"
End If
MsgBox Message, 0,"Tester Result"
End Sub
-->
</SCRIPT>
</BODY></HTML>

```

Output :



The example in the above program is simple yet powerful. First of all, it's our first useful program. We answer a question, and the computer tells us if we are right or wrong. Immediate feedback and interaction are what it's all about. Let's break down the Btn1\_OnClick procedure to see what's happening up close.

When Btn1 is clicked, the If...Then...Else statement asks if the value of the Txt1 text input box is Grant. If the answer to that question is true, the program sets the value of the variable Message to You're right. If the answer is false, the value of Message is set to Try again. Notice that there is absolutely no room for error when answering this question. If the user enters grant, GRANT, or US Grant, the program will skip to the Else part of the program, and the message will be "Try again." Later you'll learn how to make If...Then...Else statements a little more tolerant.

If you have a simple If...Then statement, the statement can be just a single line:

If x=4 then y=12

When the statement takes more than one line, you need to close it off with an End If statement.



## **For...Next**

The For...Next statement is used to run a block of code statements a certain number of times:

```
Dim x
For x = 1 to 10
Edit1.Value = x
Next
```

In this example, the counter starts at 1 and repeats Edit1.Value = x 10 times. You can also specify the way that the value of x is counted, using the Step keyword. You can use an If...Then statement to exit the loop, if necessary:

```
Dim x
For x = 5 To 50 Step 5
Edit1.Value=x
If x > 20 Then Exit For
Next
```

In this case, the counter starts at 5 and stops at 50 for a total of 10 iterations. The loop is exited after five iterations, because x is greater than 20.

You could also use the Step value to count down, using negative numbers. Starting with a higher number and counting down to a lower one would look something like this:

```
Dim x
For x = 10 to 1 Step -1
Edit1.Value = x
Next
```

## **Do...Loop**

Another common looping statement is Do...Loop. A Do...Loop statement is usually better to use than a For...Next and Exit For combination. The Do...Loop statement allows you to create a loop that runs an infinite number of times. You need to be careful when using this statement-you don't want to accidentally get your script into a loop that it can't get out of:

```
Dim x
```

```
x=1
```

```
Do Until x = 25
```

```
x = x + 1
```

```
Loop
```

In this example, the place where you need to be careful is in Do Until x = 25. If the initial value of x was 30, the code would loop continuously without the x = 25 value ever being met.

There are two ways to test for a true value in the Do...Loop statement. You can use the Until keyword to repeat the loop until a condition is true or the While keyword to repeat while the condition is true:

```
Dim x
```

```
x = 10
```

```
Do While x < 100
```

```
x = x + 10
```

```
Loop
```

By placing the While or Until keyword after the Loop statement, you can make sure that the code inside the loop is run at least once:

```
Dim x
```

```
x = 1
```

```
Do
```

```
x = x + 1
```

```
If x > 30 Then Exit Loop
```

```
Loop Until x = 30
```

The code in this Do...Loop block is run at least once before it's ended. We've also added a safety feature in the form of an Exit Loop command that's issued if x is found to be greater than 30. If x is greater than 30, there is no chance that the value will ever be 30.

## **For Each...Next**

The For Each...Next loop is used like the For...Next loop. It is used to test conditions in collections of objects. Objects have properties that are specified by keywords that are appended to the object name after a period. If I had an object with the property of color, you could access that property with MyObject.Color. Groups of objects are called collections, and you can test each of the objects for the truth of a statement using For Each...Next:

```
For Each Car in Lot
    if Car.Color = "Red" then
        MsgBox "Red car!", 0, "Answer"
    End If
Next
```

In this example, the collection of objects is called Lot, and the objects are of the type Car. We go through the cars in the lot, and when we get to a red one we see a message. You'll learn more about objects in Part II of this book.

## **While...Wend**

According to Microsoft, While...Wend is included in VBScript for those programmers who are familiar with its usage, but it's not well documented. Microsoft recommends using Do...Loop instead, but let's take a quick look at the While...Wend statement to become familiar with its usage:

```
Dim x
x = 1
While x < 10
    x = x + 1
Wend
```

The While...Wend statement repeats until the value of the While portion of the statement is true.

---

### 13.7. Array

---

An array is like a list. When you go to the grocery store, you often carry a list of the items that you need to purchase. If you were to assign the list to a scalar value, it might look something like this:

```
MyList = "peas, carrots, corn"
```

Although it's easy enough to have MyList contain the contents of the list, it's not very easy to figure out which item is which. That's where an array comes in. An array lets you easily specify items in the list and retrieve those items individually. Let's take a look at a simple array.

You declare an array variable in VBScript the same way you do a scalar variable. The difference is that you specify the number of items in the array:

```
Dim MyList(4)
```

Arrays in VBScript start their count at zero, so an array is declared with the highest count number in the array. In this case, MyList has a total of five items.

Looking at following example you can see that we've drawn five buttons on the page. An array variable, MyList, is declared with five items, and each item is assigned a different value. In the click event for each button, we attach the value of an array item to a local scalar variable called Item. We then call a procedure named ShowMessage, passing Item as a parameter. The ShowMessage procedure takes the value of Item and plugs it into the MsgBox function, which brings up the message box containing the value of the particular array item.

#### **Shopping list of items.**

```
<HTML>
<HEAD>
  <TITLE>Tester Page</TITLE>
</HEAD>

<BODY>
  <H1>Tester Page for VBScript</H1>
```

```
<HR COLOR="BLUE">  
  
  <INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Item 1"><BR>  
  <INPUT TYPE="SUBMIT" NAME="Btn2" VALUE="Item 2"><BR>  
  <INPUT TYPE="SUBMIT" NAME="Btn3" VALUE="Item 3"><BR>  
  <INPUT TYPE="SUBMIT" NAME="Btn4" VALUE="Item 4"><BR>  
  <INPUT TYPE="SUBMIT" NAME="Btn5" VALUE="Item 5"><BR>  
<SCRIPT LANGUAGE="VBScript">
```

```
<!--
```

```
Option Explicit
```

```
Dim MyList(4)
```

```
MyList(0)="Corn"
```

```
MyList(1)="Carrots"
```

```
MyList(2)="Peas"
```

```
MyList(3)="Chicken"
```

```
MyList(4)="Cake"
```

```
Sub Btn1_OnClick
```

```
  Dim Item
```

```
  Item=MyList(0)
```

```
  ShowMessage(Item)
```

```
End Sub
```

```
Sub Btn2_OnClick
```

```
  Dim Item
```

```
  Item=MyList(1)
```

```
  ShowMessage(Item)
```

```
End Sub
```

```
Sub Btn3_OnClick  
    Dim Item  
    Item=MyList(2)  
    ShowMessage(Item)  
End Sub
```

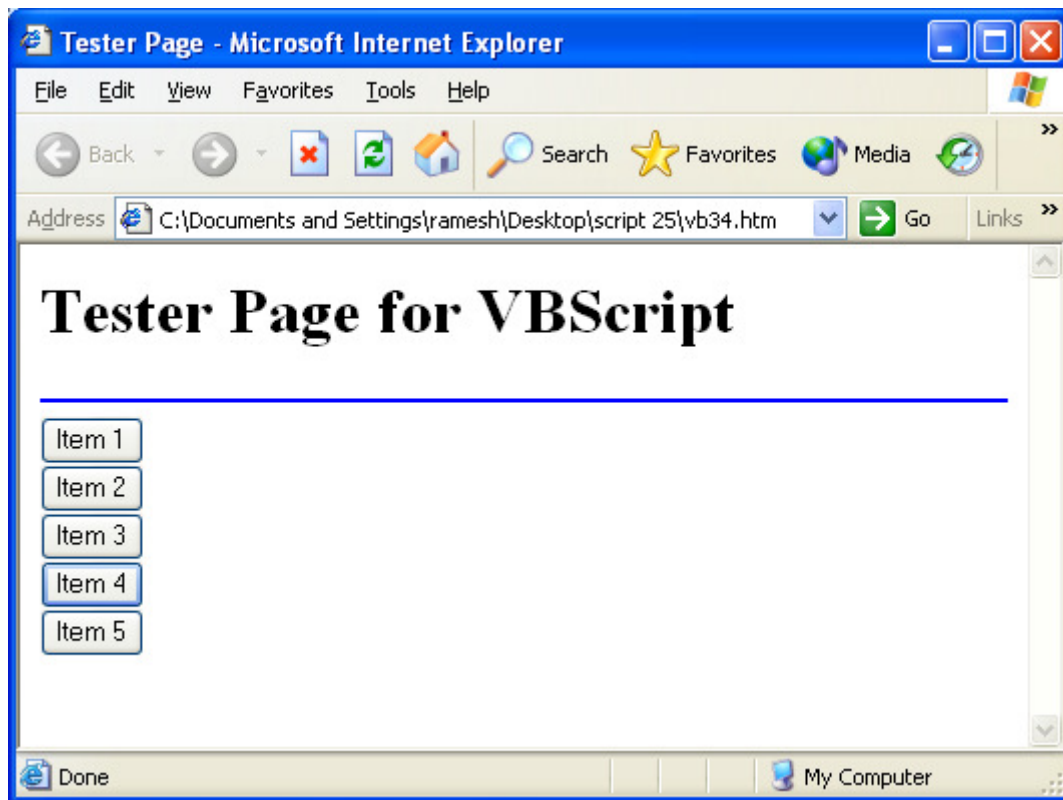
```
Sub Btn4_OnClick  
    Dim Item  
    Item=MyList(3)  
    ShowMessage(Item)  
End Sub
```

```
Sub Btn5_OnClick  
    Dim Item  
    Item=MyList(4)  
    ShowMessage(Item)  
End Sub
```

```
Sub ShowMessage(SelItem)  
    MsgBox SelItem,0,"Item picked"  
End Sub
```

```
-->  
</SCRIPT>  
</BODY>  
</HTML>
```

Output :



You can also create multidimensional arrays. A 2-D array is declared like this:

```
Dim MyArray(4,9)
```

This declaration creates a table containing five rows and 10 columns. You can pull a particular value from the array using `MyArray(row,column)`. The upper limit for multidimensional arrays is 60 dimensions.

You can declare what are called dynamic arrays that change sizes as the script is run. To declare a dynamic array, use the `Dim` or `ReDim` statement without adding a size value to the array name:

```
Dim MyDynamicArray()
```

To use this array, you would `ReDim` the array, adding a size value to the name:

```
ReDim MyDynamicArray(13)
```

You can `ReDim` an array as many times as needed.

An **array** is a type of variable that can hold a list of single variables that are referenced by their numerical index starting at 0. The index, also called a subscript, is an integer. Because indexing starts at 0 the first element in an array is `ArrayElement(0)`, the second is `ArrayElement(1)`, etc. The index of the

last element in an array will always be 1 less than the number of elements in the array and is called the *upper bound*. The *lower bound* is always 0. Arrays *must* be declared with a **Dim** statement (whether or not **Option Explicit** is used) indicating the upper bound so that the correct amount of memory can be reserved.

VBScript provides 2 built-in functions for working with arrays. **UBound()** returns the upper bound of an array. Since the upper bound of an array is always one less than the total elements in the array, we can determine the size.

The **Array()** function constructs a *dynamic array* from a list of expressions which can be assigned to a variable. This is called an "anonymous array" in some languages because the items in the list are never assigned to elements in a named array. They exist in a temporary unnamed array in memory, and that array is immediately assigned to a variable (typically an element of another array). However, because a reference to the "temporary" list still exists (in the form of array elements in the variable they were assigned to), the memory and its values remain accessible.

Arrays stored in elements of other arrays are the basis for more interesting and useful data structures. In many other languages these are called *structures* or *records*. Consider a database of Employee information where each employee has an ID, a first and last name, and number of hours worked.

A **fixed-size array**'s size does not change during program execution; a **dynamic array**'s size can change during execution. A dynamic array is also called a **redimmable array** (short for "re-dimensionable" array).

Attempting to access an index that is less than the lower bound or greater than the upper bound is an error.

Statement **ReDim** allocates memory for array dynamic All dynamic array memory must be allocated via ReDim. Dynamic arrays are more flexible than fixed-sized arrays, because they can be resized anytime by using ReDim to accommodate new data.

ReDim dynamic( 5 )



Dynamic arrays allow the programmer to manage memory more efficiently than do fixed-size arrays. Resizing dynamic arrays consumes processor time and can slow a program's execution speed. Attempting to use `ReDim` on a fixed-size array is an error.

Using `ReDim` without `Preserve` and assuming that the array still contains previous values is a logic error. Failure to `Preserve` array data can result in unexpected loss of data at runtime. Always double check every array `ReDim` to determine whether `Preserve` is needed.

Arrays can have multiple dimensions. VBScript supports at least 60 array dimensions, but most programmers will need to use only two-dimensional or three-dimensional arrays.

Referencing a two-dimensional array element `u(x, y)` incorrectly as `u(x)(y)` is an error. A multidimensional array is declared much like a one-dimensional array. For example, consider the following declarations

```
Dim b(2, 2), tripleArray(100, 8, 15)
```

which declare `b` as a two-dimensional array and `tripleArray` as a three-dimensional array.

Attempting to change the total number of array dimensions using `ReDim` is an error. Attempting to change the upper bound for any dimension except the last dimension in a dynamic multidimensional array (when using `ReDim Preserve`) is an error.

Memory allocated for dynamic arrays can be **deallocated** (released) at runtime using the keyword **Erase**. A dynamic array that has been deallocated must be redimensioned with `ReDim` before it can be used again. `Erase` can also be used with fixed-sized arrays to initialize all the array elements to the empty string. For example, the statement

```
Erase mDynamic
```

Accessing a dynamic array that has been deallocated is an error.

---

### 13.8. Let us Sum Up

---

**VBScript** is derived from **Visual Basic**, which has origins in **GW-Basic** that was shipped with all early versions of MS-DOS. GW-Basic evolved from the original Dartmouth BASIC (Beginners All-purpose Symbolic Instruction Code) developed in the 1960's.

An *expression* is a chunk of VBScript syntax in the form of a calculation that uses operators to combine numbers, strings, and variables and evaluate them to some value. An expression can be used any place where a variable is used. Variables are used to store data items that can change (e.g. "vary") during the lifetime of a program. All variables have one variable type called *variant*. Variables in VBScript do not have to be "declared" before they are used, they spring to life when you first use them. VBScript variable names are *not case sensitive*

VBScript programs will begin with a <SCRIPT> tag and end with a </Script> tag. The first thing you'll notice in the VBScript code block is the

```
<SCRIPT LANGUAGE="VBS">
```

Keep in mind that you must close out your script code block with a </SCRIPT> tag..

A variable is a name that represents the contents of some space in memory. Thinking of a variable in terms of memory space will help you to understand an important concept in programming: data types.

All variables in VBScript are of the type variant. A variant is a variable that is simply the name for a piece of data. The variant type doesn't differentiate between types of data until the time comes to process that data. This means that a variable can represent literally any value or type of value.

## Subtypes of Variant Types

Boolean

Byte

Integer

Long

Single

Double

String

Date

Empty

Object

Error

Strictly speaking, you don't need to declare your variables in VBScript. You could simply set the variables that you need on the fly:

```
MyString="This is my string"
```

To assign a value to a variable, place the variable on the left followed by an equals sign and the value on the right:

### Scalar

The types of variables we have talked about so far have been scalar variables. A scalar variable has only one value assigned to it at a time. Depending on the types of programs you write, you'll probably use scalar variables often in your scripts.

VBScript uses many familiar or intuitive operators, such as `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. VBScript has an additional division operator, `\`, which performs integer division. `8 / 5` is 1.6, where `8 \ 5` is 1. The exponentiation operator is `^`. An operator supported in virtually all programming languages that may be new is the *modulus division* operator. Modulus division returns the remainder rather than the quotient. In VBScript the modulus operator is **MOD**.

Decision-making in programs is what programming is all about.

### **If...Then...Else**

The If...Then...Else statement is one of the basic capabilities of a computer language. This statement tests for the truth of a statement and, depending on the answer, processes information or moves on to another piece of code.

### **For...Next**

The For...Next statement is used to run a block of code statements a certain number of times:

### **Do...Loop**

Another common looping statement is Do...Loop. A Do...Loop statement is usually better to use than a For...Next and Exit For combination. The Do...Loop statement allows you to create a loop that runs an infinite number of times.

### **For Each...Next**

The For Each...Next loop is used like the For...Next loop. It is used to test conditions in collections of objects. Objects have properties that are specified by keywords that are appended to the object name after a period.

### **While...Wend**

According to Microsoft, While...Wend is included in VBScript for those programmers who are familiar with its usage, but it's not well documented. Microsoft recommends using Do...Loop instead, but let's take a quick look at the While...Wend statement to become familiar with its usage:

### **Array**

An array is like a list. When you go to the grocery store, you often carry a list of the items that you need to purchase. You declare an array variable in VBScript the same way you do a scalar variable.

Statement **ReDim** allocates memory for array dynamic All dynamic array memory must be allocated via ReDim. Dynamic arrays are more flexible than fixed-sized arrays, because they can be resized anytime by using ReDim to accommodate new data.

ReDim dynamic( 5 )

---

### **13.9. Lesson end Activities**

---

1. What is the purpose of Variables?
2. What are the datatypes available in VBScript?
3. Explain Arrays in VBScript.

---

### **13.10. Check your Progress**

---

1. Describe different types of control statements available in VBScript.
2. Explain the operators available in VBScript.
3. Write a VBScript program for array multiplication.

---

### **13.11. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.microsoft.com](http://www.microsoft.com)



---

## Lesson 14

# VBScript - II

---

### Content

#### 14.0. Objective

#### 14.1.Introduction

#### 14.2. Sub Procedures and Function Procedures

#### 14.3. Built-in Functions

#### 14.4 Basic Functions

#### 14.5. String Functions

#### 14.6. Conversion Functions

#### 14.7. Math Functions

#### 14.8. Time and Date Functions

#### 14.9. Boolean Functions

#### 14.10. Class and Objects

#### 14.11. Let us Sum Up

#### 14.12. Lesson End Activities

#### 14.13. Check Your Progress

#### 14.14. Reference

---

#### 14.0. Objective

---

- Learn about procedures and how they are used in VBS scripts
- Learn about the difference between a Sub procedure and a Function procedure
- Learn about VBScript's intrinsic functions
- Learn how to utilize VBS functions in your own scripts

---

#### 14.1. Introduction - Procedure

---

Procedures are the logical parts into which a program is divided. The code inside a procedure is run when the procedure is called. A procedure can

be called with a Call statement in another procedure, or it can be triggered by an event such as a button click.

Events are triggered when messages are sent from the operating system to VBScript. In graphical operating systems, such as Windows, the way that different applications interact with the operating system is by sending and receiving. The operating system is the controlling force in the graphical environment, and the scripts that you write will depend in large part on the resources and messages made available to you from Windows.

---

## 14.2. Sub Procedures and Function Procedures

---

There are two types of procedures in VBScript: Sub procedures and Function procedures. Sub procedures are blocks of code that are wrapped in the Sub...End Sub keywords. A Sub can take arguments and process them within the Sub procedure. A Sub can call other procedures, but it can't return a value generated to the calling procedure directly.

A Function procedure works just like a Sub procedure. It can take arguments and call other procedures. Most importantly, Function procedures return a value to the calling procedure.

Let's take a look at some sample code. Following example contains three procedures. We'll use message boxes to track the program flow.

### **Flow.htm.**

```
<HTML>
<HEAD>
<TITLE>Tracking Procedures</TITLE>
</HEAD>
<BODY>
<H1>Tester Page for VBS</H1>
<HR COLOR="BLUE">
<INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Click to test the code">
<SCRIPT LANGUAGE="VBScript">
```



```

<!--
Sub Btn1_OnClick
    Dim Message, x
    x=100
    Message="Sub Btn1_OnClick"
    MsgBox Message, 0,"Procedure Result"
    Message = "Sub MySub"
    MySub(Message)
    x = ReturnCount(x)
    Message = "Function returned " + CStr(x)
    MsgBox Message, 0, "Procedure Result"
End Sub

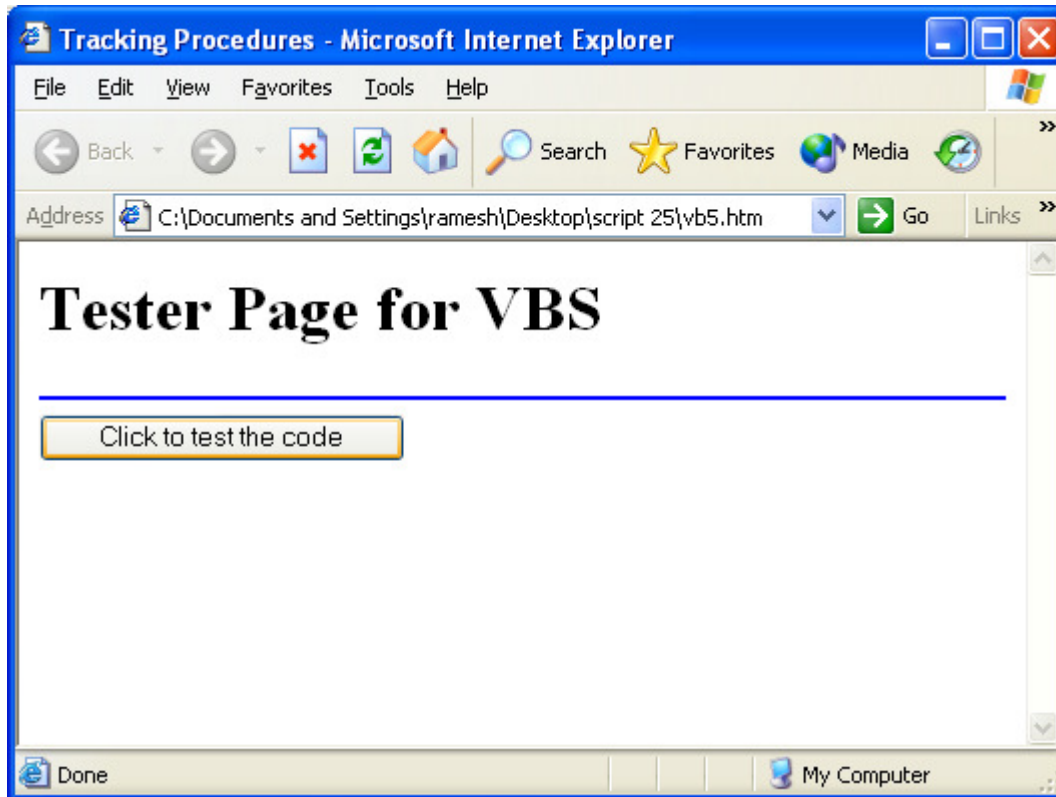
Sub MySub(Msg)
    MsgBox Msg, 0, "Procedure Result"
End Sub

Function ReturnCount(Num)
    ReturnCount = Num + 1
End Function
-->

</SCRIPT>
</BODY>
</HTML>

```

Output



Let's track Listing 3.1 and see where it's going. Using our tester template, we create two procedures to go with the Sub Btn1\_OnClick procedure that was already in place. The MySub procedure takes an argument called Msg and in turn passes that value to a MsgBox function. The other new procedure is the Function ReturnCount, which simply takes an argument, Num, and adds 1 to it.

If you take a look at how the program flows, you'll see that it all begins in Sub Btn1\_OnClick. The variable Message is assigned the name Sub Btn1\_OnClick, and a MsgBox function is called with Message as one of the arguments. Notice that the flow of the Sub Btn1\_OnClick procedure is halted until the message box is dismissed.

After the message box is closed, the value of Message is set to the name of the second procedure in the script, Sub MySub. The MySub procedure is called, with Message as an argument. The argument Message is then called in another MsgBox function call from within MySub. Again, program flow is stopped while the message box is displayed.

Finally, the variable `x`, having previously been set to a value of 100, is used as an argument in the call to the function `ReturnCount`. `ReturnCount` adds 1 to `x` and returns the value to the calling procedure. We then make `x` part of the `Message` variable. Notice that the argument we pass to the `MsgBox` function must be a string, so we need to convert the value of `x` to a string before it can be made part of `Message`. The message box is then displayed with the message "Function returned 101." When the message box is closed, `End Sub` is hit, and the script ends.

### **Arguments to Procedures**

Arguments can be used to pass data to either `Sub` or `Function` procedures. When calling a procedure, you can simply use the procedure name followed by arguments, which are separated by commas:

```
MyProcedure Arg1, Arg2, Arg3
```

Or you can use the `Call` statement, in which case the argument list must be enclosed by parentheses:

```
Call MyProcedure(Arg1, Arg2)
```

You can also call the procedure without the `Call` statement and still include parentheses.

### **Creating and Calling Functions**

When you create and use a function, the return value of the function is held by a variable with the name of the function. For example, if you create a function that converts pennies to dollars, you might name the function `Dollar`:

```
Function Dollar(Cents)
```

```
    Dollar = Cents/100
```

```
End Function
```

The function returns the value of `Dollar` to the calling procedure.

To call the `Dollar` function from another procedure, you must use a variable of some sort to hold the return value. If you call `Dollar` from the following code, the value `Dollars` will hold the returned value of the function:

```
Dollars = Dollar(2456)
```

Function procedures are great tools for working with data in a VBS script. When you create functions that you know you might want to use again, be sure to save them, so that you can easily reuse them in new scripts.

---

### **14.3. Built-in Functions**

---

In addition to creating your own function procedures for use in your scripts, you can use a number of intrinsic function procedures that are built into VBScript. These functions include string operations, conversions, math functions, time and date functions, and Boolean functions. Understanding these functions will benefit you greatly as you begin to write larger and more complex scripts.

---

### **14.5. Basic Functions**

---

The message box is one of the most useful functions in VBScript. The two types of message boxes available to you are the message box and the input box.

#### **InputBox**

The InputBox function makes it possible for you to get input from a user without placing a text control on the form. It takes seven possible parameters:

`InputBox(prompt, title, default, xpos, ypos, helpfile, context)`

All the arguments of the InputBox function are optional except prompt. Let's take a look at each of these parameters in detail.

#### **prompt**

The prompt argument is a string containing the question that you are asking your user:

`InputBox("What is your name?")`

#### **Title**

The Title argument determines the title of the dialog. This argument must be a string expression. If the title isn't specified, the title of the dialog defaults to the application name.

**default**

The default argument specifies a string that appears in the input box on the dialog when the dialog is displayed. Leave this parameter blank if you don't want the dialog to display default text.

**xpos and ypos**

The xpos and ypos arguments specify where in relation to the screen the dialog is displayed. These arguments are made in twips. The xpos argument specifies the horizontal value, and the ypos argument specifies the vertical value.

**helpfile and context**

The helpfile argument specifies a Windows Help file to open if the user presses the f1 button. If you specify a Help file, you also need to specify a context id.

The context argument specifies a Help context id number for the file called in the helpfile argument. The Help context id number opens the corresponding Help topic.

**len**

The len function returns the length of a string or the number of bytes in a variable.

**MsgBox**

The message box is useful when you want to notify a user that an event has occurred. You can specify the buttons shown in the dialog, and the function returns a value that indicates which button was clicked:

MsgBox(prompt, buttons, title, helpfile, context)

The helpfile and context arguments in the MsgBox function are optional.

**prompt**

The prompt argument is a string value of up to 1024 characters. As with the InputBox function, you can use the carriage return line-feed combination (Chr(13)& Chr(10)), a carriage return (Chr(13)), or a line feed (Chr(10)) to break the prompt into multiple lines:

Dim Message

Message = "This text is broken" + (Chr(13)& Chr(10)) + "into multiple lines."

MsgBox (Message, 0, "Message Title")

### **buttons**

The buttons argument is a VBScript constant or a numeric value that determines which buttons are displayed on the dialog. Table 14.1 lists the values and constants that you can use when calling this function.

**Table 14.1. Settings for the buttons argument.**

<i>Value</i>	<i>Buttons shown</i>
0	OK
1	OK, Cancel
2	Abort, Retry, Ignore
3	Yes, No, Cancel
4	Yes, No
5	Yes, No, Cancel
16	(Critical Message Icon)
32	(Warning Query Icon)
48	(Warning Message Icon)
64	(Information Message Icon)
0	(First button default)
256	(Second button default)
512	(Third button default)
0	(User must respond before continuing with application)
4096	(User must respond before continuing with the operating system)

**title**

The title argument is a string that specifies the text shown in the titlebar of the message box. If no title is specified, the title of the calling application is displayed.

**Helpfile and Context**

These arguments work the same as for InputBox. Helpfile specifies a Windows Help file, and context specifies the Help context id for the topic.

**Return Values**

The return values for the MsgBox function are the numeric or constant values of the button pressed. Table 14.2 lists the possible return values for the MsgBox function.

**Table 14.2. Possible return values for the MsgBox function.**

<i>Button Clicked</i>	<i>Value</i>
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

**Example : MsgBox return value.**

<HTML>

<HEAD>

<TITLE>The MsgBox Returns</TITLE>

</HEAD>

<BODY>

```

<H1>Tester Page for VBS</H1>
<HR COLOR="BLUE">
<INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Click to test the code">
<SCRIPT LANGUAGE="VBScript">

<!--
Sub Btn1_OnClick()
Dim strQuestion, intReturn
Do Until intReturn = 7
strQuestion = "Do you want to see another dialog?"
intReturn=MsgBox(strQuestion, 4, "Question" )
Loop
End Sub
-->

</SCRIPT>
</BODY>
</HTML>

```

In this script, the button on the page is clicked, and a yes/no message box is displayed asking whether the user wishes to see the message box repeated. If the answer stored in intReturn holds a value of Yes (6), then the dialog is redisplayed. If the user clicks No (7), the message box is closed and the script is ended until the button is pushed again.

### **VarType**

The VarType function returns the subvalue of a variable. This function is useful for verifying the contents of a variable or for checking the type of variable before trying to operate on it. Table 14.3 shows the possible return values for the VarType function.



**Table 14.3. Possible return values for the VarType function.**

<i>Subtype</i>	<i>Value</i>
(empty)	0
(null)	1
Integer	2
Long	3
Single	4
Double	5
Currency	6
Date	7
String	8
OLE Object	9
Error	10
Boolean	11
Variant	12
Non-OLE Object	13
Byte	17
Array	8192

You can use either the value or the constant to determine the return type. When the variable on which you're running the function is an array, the function returns the vbArray value added to the variable type in the array.

## 14.5. String Functions

The following string functions act on string data to allow you to manipulate and parse strings. Parsing textual data means dividing it into logical divisions. Let's take a look at the various string-related functions and then look at an example that utilizes some of the functions.

### **Asc**

The Asc function takes a string as a parameter and returns the ASCII character code for the first character in the string:

```
Dim strData, intCharCode  
strData = "This is a string"  
intCharCode = Asc(strData)
```

In this code example, the function returns the value of T, which is 84. If you run this function on an empty string, it generates a runtime error.

### **Chr**

The Chr function takes a character code as a parameter and returns the corresponding character. Keep in mind that the characters for codes 0 to 31 are nonprintable:

```
StrMyString = "This is my two line" + (Chr(13) + Chr(10)) + "text string example."
```

In this example line, the string has a carriage return and a line feed added to the middle of the text. This will cause a break in the string, displaying it on two lines.

### **InStr**

The InStr function is a powerful text-searching tool that finds the position of text substrings within strings of text. This is a fairly complex function, so you should first be familiar with the function's syntax:

```
position = InStr(startpos, string1, string2, type)
```

The InStr function returns the position of the substring within the string. In this case, the return value is held by the variable position. Let's go through the other arguments individually.

**Startpos**

The startpos argument is a numeric value that tells the function where to start searching. This is an important argument because if you're thinking about adding search functions to your own string operations, you'll need to adjust the number as you find multiple occurrences of the search string in a large body of text.

**String1**

The string1 argument is the string in which you are searching for the string2 string.

**String2**

String2 is the text for which you're searching.

**Type**

The type argument specifies the string comparison that you're performing. This argument can have a value of 0 (default) or 1. A type 0 comparison is a binary comparison. The function will return a position only in the case of an exact match. An argument of 1 will do a non-case-sensitive, textual search.

```
Dim strBigString, strSearchString, intReturn0, intReturn1
```

```
strBigString = "This is the BIG string"
```

```
strSearchString = "big"
```

```
intReturn0 = InStr( , strBigString, strSearchString, 0)
```

```
intReturn1 = InStr( , strBigString, strSearchString, 1)
```

In this sample, the intReturn0 variable is set to 0 because the function does not find the string "big" in strBigString. The variable intReturn1 is set to 13 because strSearchString is found in the non-case-sensitive search.

**Table14.4. Return values for the InStr function.**

<i>String Values</i>	<i>Return Values</i>
mainstring length is 0	0
mainstring is Null	Null
searchstring length is 0	startpos
searchstring is Null	Null
searchstring not found	0
searchstring found	(position of string in mainstring)

If the value of startpos is greater than the length of mainstring, the function returns a 0.

### **LCase**

The LCase function takes a string and converts all the characters to lowercase. It takes a string as an argument and returns the converted string.

### **Left**

The Left function returns a string containing the characters from the left side of the string, up to a specified number. The function takes two arguments, string and length:

```
Dim strMyString, strMain
```

```
strMain = "The rain in Spain..."
```

```
strMyString = Left(strMain, 15)
```

In this sample, the string variable strMyString would be set to The r, the first five characters of the string. If the number you specify in the length argument is greater than or equal to the length of the string, the function will return the entire string.

## **LTrim**

The LTrim function returns a copy of the string argument with the leading spaces removed from the string:

```
Dim strMyString, strMain
strMain = "  There are four leading spaces here."
strMyString = LTrim(strMain)
```

In this example, the function returns the string, "There are four leading spaces here."

## **Mid**

The Mid function returns a substring of a specified length from another string. This function takes three parameters: string, start, and length. The length argument is optional.

```
Dim strMyString, strMain
strMain = "Ask not what your country can do for you..."
strMyString = Mid(strMain, 8, 10)
```

In this example, we're looking for a string that starts at character 8 and is 10 characters in length. The string value contained in strMyString is equal to what your .

## **Right**

The Right function works like the Left function, but it returns a specified number of characters starting from the last character in the string. This function takes the number of characters as an argument and returns a string:

```
Dim strMyString, strMain
strMain = "How now brown cow?"
strMyString = Right(strMain, 10)
```

In this example, strMyString is set to the last 10 characters of strMain, or brown cow?.

## **RTrim**

The Rtrim function works like the Ltrim function. It removes trailing spaces from a string. It takes a single argument, string, and returns a string.

## Str()

The Str()function takes a number as an argument and returns a string value representing the number. The first character in the resulting string is always a space that acts as a placeholder for the sign of the number. If you want to use a numeric value in a string or a function that takes a string as a parameter, you'll need to use this function or the CStr function to make the conversion first.

## StrComp

The StrComp function takes two strings as arguments and compares them, based on the third argument, which defines the type of comparison:

```
Dim RetVal, strString1, strString2
```

```
strString1 = "This is a string."
```

```
strString2 = "This is a STRING."
```

```
RetVal = StrComp(strString1, strString2, 1)
```

The StrComp function returns a numeric value that indicates whether the items are the same. The comparison type has two possible values: 0 (default) is a binary comparison, and 1 is non-case-sensitive. Table 14.5 show the return values for the StrComp function.

**Table 14.5. Return values for the StrComp function.**

<i>Return Value</i>	<i>Description</i>
-1	String1 < String2
0	String1 = String2
1	String1 > String2
NULL	One of the strings is null

## String

The String function takes a number and character code argument and returns the character, repeated a number of times:

```
Dim strRetString
```

```
strRetString = String(3, 97)
```

This example returns the string "aaa". If the character code argument is greater than 255, the code is automatically converted to a valid character using the formula `charcode = (char Mod 256)`.

### **Trim**

The Trim function returns the string argument, with leading and trailing spaced removed.

### **UCase**

The UCase function converts all the characters in the string argument to uppercase.

### **Val**

The Val function takes a string argument and returns numbers from the string. The function stops retrieving numbers as soon as it hits a non-numeric character:

```
Dim MyNumber, strMyString
```

```
MyString = "300 South Street"
```

```
MyNumber = Val(strMyString)
```

In this example, the function returns the number 300. The Val function recognizes decimal points and radix prefixes. A radix prefix is a prefix that defines an alternative numbering system. The &O prefix is used for octal values and &H for hexadecimal.

## **14.6. Conversion Functions**

Conversion functions enable you to change the subtype of a variable. Although VBScript uses the variant type for all variables, in many cases an argument of a certain type is required. An example would be a function that takes a string argument. If you want to be able to use numeric data, you'll need to convert it to a string before calling the function.

### **CByte**

The Cbyte function converts an expression to the subtype byte.

## **CDbl**

The CDbl function converts an expression to the subtype double.

## **CInt**

The CInt function converts an expression to the subtype integer.

## **CLng**

The CLng function converts an expression to the subtype long.

## **CStr**

The CStr function returns a string from a valid expression. Table 3.6 lists the return values for various expression types.

**Table 14.6. Return values for the CStr function.**

<i>Return Value</i>	<i>Expression Type</i>
True or False	Boolean
Short-Date	Date
Runtime Error	Null
" "	Empty
Error(#)	Error
Number	Number

## **CVErr**

The CVErr function returns the subtype error. It takes any valid error number as an argument.

---

## **14.7. Math Functions**

---

Math functions enable you to perform operations on numbers in your VBS scripts. You'll find these functions fairly straightforward.

### **Abs**

The Abs function takes a number as a parameter and returns its absolute value. The absolute value of a number is the numerical value of



a number without considering its sign. An argument of -7 would return the value 7.

### **Array**

The Array function returns a variant value containing an array. The array can be of any subtype. This function takes a list of values separated by commas as a parameter.

### **Atn**

The Atn function returns the arctangent of a number, a trigonometric function that is used to determine angles in triangles. This function is the inverse of tangent (Tan), which calculates the ratio of sides in a right triangle. It takes a number as an argument.

### **Exp**

The Exp function takes a numeric argument and returns e (the base of natural logarithms) raised to a power.

### **Hex**

The Hex function returns a string value containing the value of an argument converted to hexadecimal form. If the argument is a fractional value, it is rounded to the nearest whole number before the function returns the string.

Keep in mind that this function returns a string. If you want to perform mathematical operations on the returned value, you must first convert the string back into a numerical value. Hexadecimal numbers are represented in VBScript using the &H prefix.

### **Int**

The Int function returns the whole number portion of an argument. If the argument is negative, Int returns the first integer value that is less than or equal to the argument.

### **Fix**

The Fix function works like the Int function, returning the whole number portion of an argument. The difference is that if the number

argument is negative, Fix returns the first integer value that is greater than or equal to the argument.

### **Log**

The Log function returns the natural logarithm of a numeric argument. The numeric argument that this function processes must be greater than zero.

### **Oct**

The Oct function returns a string representing the octal value of a numeric argument. If the numeric argument is fractional, it is rounded up before the function returns a value. As with the Hex function, the returned string must be converted back to numeric form before you can perform mathematical operations on it. To use an octal value mathematically, you use the &O prefix.

### **Rnd**

The Rnd function takes a numeric argument and returns a value between zero and one. The number generated depends on the numeric argument in relation to the values in Table 14.7.

**Table 14.7. Number value determines generation technique.**

<i>Number Value</i>	<i>Generates</i>
<0	Same number every time
>0	Next random number
=0	Last generated number
""	Next random value in sequence

If you want to generate a range of random numbers, use the following formula:

$\text{Int}((\text{upperbound} - \text{lowerbound} + 1) * \text{Rnd} + \text{lowerbound})$

**Example : Lotto.htm.**

```
<HTML>
<HEAD>
<TITLE>Play 3 Generator</TITLE>
</HEAD>
<BODY><CENTER>
<H1>Play 3 Generator</H1>
<HR COLOR="BLUE">
<INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Click for the lucky
numbers!"><BR><BR>
<INPUT TYPE="SUBMIT" NAME="BtnBall1" VALUE="">
<INPUT TYPE="SUBMIT" NAME="BtnBall2" VALUE="">
<INPUT TYPE="SUBMIT" NAME="BtnBall3" VALUE="">
<SCRIPT LANGUAGE="VBScript">

<!--
Sub Btn1_OnClick()
    BtnBall1.Value = RndBall()
    BtnBall2.Value = RndBall()
    BtnBall3.Value = RndBall()
End Sub

Function RndBall()
    RndBall = (Int((9-0+1)*Rnd+0))
End Function
-->

</SCRIPT>
</CENTER>
</BODY>
</HTML>
```

Output :



### **Sgn**

The Sgn function returns a numeric value representing the sign of a number argument. It returns 1 if the number is greater than zero, 0 if equal to zero, and -1 if less than zero.

### **Sqr**

The Sqr function returns the square root of a numeric argument. The argument value must be greater than or equal to zero.

### **Sin**

The Sin function returns the sine of an angle.

### **Tan**

The Tan function returns the tangent of an angle.

---

## **14.8. Time and Date Functions**

---

You'll find time and date functions extremely useful in customizing your Web pages. You can add automatic time and date

stamping to pages, and you can write programs that provide useful calendar functions.

### **Date**

The Date function takes no arguments and returns the current system date. In Figure 3.5, the Date function returns the date value in the tester page.

### **DateSerial**

The DateSerial function takes year, month, and day arguments and returns a variant of subtype date. The year argument can be any year between 100 and 9999. The month and day arguments are numeric values.

### **DateValue**

The DateValue function takes a string argument containing a valid date and returns a variant of subtype date. This is an extremely useful function, because it interprets a number of formatted date strings. For example, you could use "January 1, 1999," "1/1/99," or "Jan 1, 1999" as an argument. Once the date is a variant of subtype date, other date functions can be used on it.

### **Day**

The Day function takes a date argument and returns the day as a numeric value between 1 and 31.

### **Hour**

The Hour function takes a time argument and returns an hour value between 0 and 23.

### **Year**

The Year function takes a date value and returns the year.

### **Weekday**

The Weekday function takes a date and optionally a firstdayofweek argument and returns a numeric value representing the day of the week. If firstdayofweek isn't specified, the function defaults to Sunday. The settings for firstdayofweek are shown in Table 14.8.

**Table 14.8. Day constants.**

<i>Day</i>	<i>Numeric Value</i>
System *	0
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

\*This value is used only in the firstdayofweek argument and is not a return value.

Listing 3.4 takes a string that you enter, converts the string to date format, and then runs the Day function against the date. The return value is then converted to a string that sends the day of the week to the message box. There is no error checking built into the script, so a nonvalid date entry will result in a runtime error. Figure 3.6 shows the results of Listing 3.4.

**Example :. Day.htm.**

```
<HTML>
<HEAD>
<TITLE>Day of the week</TITLE>
</HEAD>
<BODY>
<H1>What day did it happen?</H1>
<HR COLOR="BLUE">
```

Enter any valid date and click the button to find out what day it was!<BR>

<INPUT TYPE="TEXT" NAME="TxtDate"><BR>

<INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Tell me the day of the week">

<SCRIPT LANGUAGE="VBScript">

<!--

Sub Btn1\_OnClick()

Dim DayVal, Message, MyDate

MyDate = DateValue(TxtDate.Value)

DayVal = Weekday(MyDate)

If DayVal = 1 then Message = "Sunday"

If DayVal = 2 then Message = "Monday"

If DayVal = 3 then Message = "Tuesday"

If DayVal = 4 then Message = "Wednesday"

If DayVal = 5 then Message = "Thursday"

If DayVal = 6 then Message = "Friday"

If DayVal = 7 then Message = "Saturday"

Message = "It happened on a " + Message + "."

MsgBox Message, 64, "When did it happen?"

End Sub

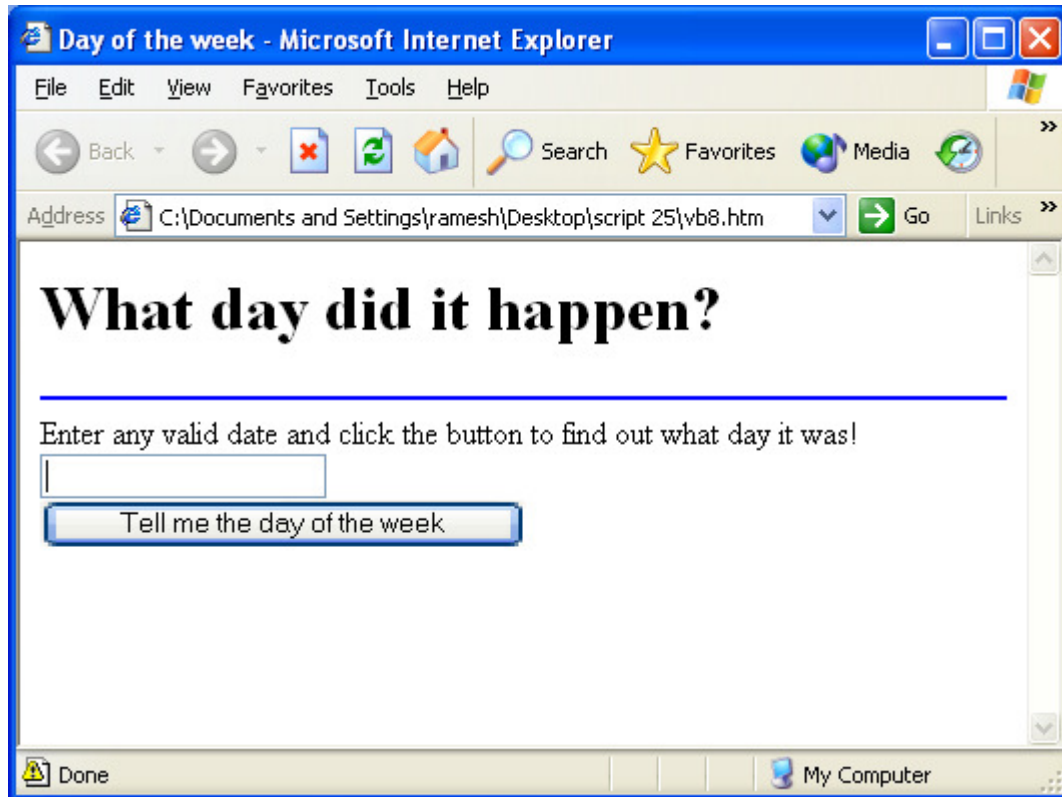
-->

</SCRIPT>

</BODY>

</HTML>

Output



### **Minute**

The Minute function retrieves the minute value of a time value.

### **Month**

The Month function returns a numeric value for the month from a valid date.

### **Now**

The Now function returns the current date and time from the client machine. This function takes no arguments.

### **Second**

The Second function returns the seconds value from a time value.

### **Time**

The Time function returns the current system time as a date subtype variant.



## TimeSerial

The TimeSerial function takes hours, minutes, and seconds as arguments and returns a variant of subtype date. The hour argument is an integer between 0 and 23.

## TimeValue

The TimeValue function takes a string containing a valid time and returns a variant of subtype date containing the time.

You can use this function to get input from the user and convert it to a date format. Valid values for the time argument include times from 12- and 24-hour clocks.

---

## 14.9. Boolean Functions

---

Boolean functions always return a value of true or false. Each of the functions listed in Table 14.9 tests for the truth of a condition.

**Table143.9. Boolean functions.**

<i>Function</i>	<i>Tests</i>
IsArray	Is variable an array?
IsDate	Is expression a date?
IsEmpty	Has the variable been initialized?
IsError	Is this an error value?
IsNull	Is this a null value?
IsNumeric	Is this a numeric value?
IsObject	Is this variable an object?

These Boolean functions are important because VBScript has little built-in error checking and no debugger other than Internet Explorer. You can use the Boolean functions to test data before trying to feed the data into functions where it might cause an error.

The following example shows the same program with a feature to check whether the data from the text input box is a valid date.

**Example : DayChk.htm.**

```
<HTML>
<HEAD>
<TITLE>Day of the week</TITLE>
</HEAD>
<BODY>
<H1>What day did it happen?</H1>
<HR COLOR="BLUE">
Enter any valid date and click the button to find out what day it was!<BR>
<INPUT TYPE="TEXT" NAME="TxtDate"><BR>
<INPUT TYPE="SUBMIT" NAME="Btn1" VALUE="Tell me the day of the week">
<SCRIPT LANGUAGE="VBS">

<!--
Sub Btn1_OnClick()
    Dim DayVal, Message, MyDate, blnCheck
    blnCheck = IsDate(TxtDate.Value)
    If blnCheck = True then
        MyDate = DateValue(TxtDate.Value)
        DayVal = Weekday(MyDate)
        If DayVal = 1 then Message = "Sunday"
        If DayVal = 2 then Message = "Monday"
        If DayVal = 3 then Message = "Tuesday"
```

```

    If DayVal = 4 then Message = "Wednesday"
    If DayVal = 5 then Message = "Thursday"
    If DayVal = 6 then Message = "Friday"
    If DayVal = 7 then Message = "Saturday"

    Message = "It happened on a " + Message + "."

    MsgBox Message, 64, "When did it happen?"

Else

Message = "You must enter a valid date."

MsgBox Message, 48, "Error"

End If

End Sub

-->

</SCRIPT>

</BODY>

</HTML>

```

---

## 14.10 Classes and Objects

---

Objects **encapsulate** (i.e., wrap together) data (**attributes**) and methods (**behaviors**); the data and methods of an object are intimately related. Objects have the property of **information hiding**. This phrase means that objects may communicate with one another, but they do not know how other objects are implemented— implementation details are hidden within the objects themselves. Surely it is possible to drive a car effectively without knowing the details of how engines and transmissions work.

In VBScript, the unit of object-oriented programming is the **Class** from which objects are **instantiated** (i.e., created). **Methods** are VBScript procedures that are encapsulated with the data they process within the “walls” of classes. VBScript programmers can create their own **user-defined types** called **classes**. Classes are also referred to as **programmer defined types**. Each class contains data as well as the set of methods that manipulate the data. The data components of a class are called **instance variables**. Just as an instance of a

Variant is called a **variable**, an instance of a class is called an **object**. The focus of attention in object-oriented programming with VBScript is on classes rather than methods.

This section explains how to create and use objects, a subject we call **object-based programming (OBP)**. VBScript programmers craft new classes and reuse existing classes. Software is then constructed by combining new classes with existing, well-defined, carefully tested, well-documented, widely available components. This kind of **software reusability** speeds the development of powerful, high-quality software. **Rapid applications development (RAD)** is of great interest today. Early versions of VBScript did not allow programmers to create their own classes, but VBScript programmers can now indeed develop their own classes, a powerful capability also offered by such object-oriented languages as C++ and Java.

Packaging software as classes out of which we make objects makes more significant portions of major software systems reusable. On the Windows platform, these classes have been packaged into class libraries, such as Microsoft's **MFC (Microsoft Foundation Classes)**, that provide C++ programmers with reusable components for handling common programming tasks, such as the creating and manipulating of graphical user interfaces.

Objects are endowed with the capabilities to do everything they need to do. For example, employee objects are endowed with a behavior to pay themselves. Video game objects are endowed with the ability to draw themselves on the screen. This is like a car being endowed with the ability to go faster (if someone presses the accelerator pedal), go slower (if someone presses the brake pedal) and turn left or turn right (if someone turns the steering wheel in the appropriate direction). The blueprint for a car is like a class. Each car is like an instance of a class. Each car comes equipped with all the behaviors it needs, such as "go faster," "go slower" and so on, just as every instance of a class comes equipped with each of the behavior instances of that class.

Classes normally hide their implementation details from the **clients** (i.e., users) of the classes. This is called **information hiding**.

As an example of information hiding, let us consider a data structure called a **stack**. Think of a stack in terms of a pile of dishes. When a dish is placed on the pile, it is always placed at the top (referred to as *pushing* the dish onto the stack). When a dish is removed from the pile, it is always removed from the top (referred to as *popping* the dish off the stack). Stacks are known as **last-in, first-out (LIFO) data structures**—the last item *pushed* (inserted) on the stack is the first item popped (removed) from the stack. So if we push 1, then 2, then 3 onto a stack, the next three pop operations will return 3, then 2, then 1.

The programmer may create a stack class and hide from its clients the implementation of the stack. Stacks can be implemented with arrays and other techniques, such as linked lists. A client of a stack class need not know how the stack is implemented. The client simply requires that when data items are placed in the stack with *push* operations, they will be recalled with *pop* operations in last-in, first-out order. Describing an object in terms of behaviors without concern for how the behaviors are actually implemented is called **data abstraction**, and VBScript classes define **abstract data types (ADTs)**. Although users may happen to know how a class is implemented, they should not write code that depends on these details. This allows a class to be replaced with another version without affecting the rest of the system, as long as the Public interface of the class does not change (i.e., every method still has the same name, return type and parameter list in the new class definition).

A primary activity in VBScript is creating new data types (i.e., **classes**) and expressing the interactions among **objects** of those classes.

An ADT actually captures two notions, a *data representation* of the ADT and the *operations allowed* on the data of the ADT. For example, subtype integer defines addition, subtraction, multiplication, division and other operations in VBScript, but division by zero is undefined. The allowed operations and the data representation of negative integers are clear, but the operation of taking the square root of a negative integer is undefined.

Access to Private data should be carefully controlled by the class's methods. For example, to allow clients to read the value of Private data, the class can provide a **get method**. It also called an **accessor** method or a **query** method.

To enable clients to modify Private data, the class can provide a **set** method. It also called a **mutator** method. Such modification would seem to violate the notion of Private data. But a *set* method can provide data validation capabilities (e.g., range checking) to ensure that the data is set properly and to reject attempts to set data to invalid values. A *set* method can also translate between the form of the data used in the interface and the form used in the implementation. A *get* method need not expose the data in raw format; rather, the *get* method can edit the data and limit the view of the data that the client will see.

The class designer need not provide set or get methods for each Private data member; these capabilities should be provided only when it makes sense and after careful thought.

Classes often provide Public methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) Private instance variables. These methods are special methods in VBScript called **Property Let**, **Property Set** and **Property Get** (collectively these methods and the internal class data they manipulate are called **properties**). More specifically, a method that sets variable `mInterestRate` would be named `Property Let InterestRate` and a method that gets the `InterestRate` would be called `Property Get InterestRate`.

Procedures `Property Let` and `Property Set` differ in that `Property Let` is used for nonobject subtypes (e.g., integer, string, byte) and `Property Set` is used for object subtypes.

Attempting to call a method or access a property for a reference that does not refer to an object is an error. Attempting to assign a reference a value without using `Set` is an error.

Any `Property Get`, `Property Let` or `Property Set` method may contain the **Exit Property** statement that causes an immediate exit from a `Property` procedure.

Accessor methods can read or display data. Another common use for accessor methods is to test the truth or falsity of conditions—such methods are often called **predicate methods**.

An example of a predicate method would be an `IsEmpty` method for any container class—a class capable of holding multiple objects—such as a linked list or a stack. A program might test `IsEmpty` before attempting to remove another item from a container object. A program might test `IsFull` before attempting to insert another item into a container object. It would seem that providing *set* and *get* capabilities is essentially the same as making the instance variables `Public`. This is another subtlety of VBScript that makes the language desirable for software engineering. If an instance variable is `Public`, it may be read or written at will by any method in the program. If an instance variable is `Private`, a `Public get` method certainly seems to allow other methods to read the data at will, but the *get* method controls the formatting and display of the data. A `Public set` method can—and most likely will—carefully scrutinize attempts to modify the instance variable's value. This ensures that the new value is appropriate for the data item. For example, an attempt to *set* the day of the month to 37 would be rejected, an attempt to *set* a person's weight to a negative value would be rejected, and so on.

This example briefly introduces a VBScript feature for complex pattern matching called **regular expressions**. We use regular expressions to validate the format of the social security number. Client-side scripts often validate information before sending it to the server.

Example :

```
<!-- classes.html -->
<!-- VBScript Class -->

<html>
<head>
<title>Using a VBScript Class</title>

<script type = "text/vbscript">
<!--
Option Explicit
```

Class Person

Private name, yearsOld, ssn

Public Property Let FirstName( fn )

name = fn

End Property

Public Property Get FirstName()

FirstName = name

End Property

Public Property Let Age( a )

yearsOld = a

End Property

Public Property Get Age()

Age = yearsOld

End Property

Public Property Let SocialSecurityNumber( n )

If Validate( n ) Then

ssn = n

Else

ssn = "000-00-0000"

Call MsgBox( "Invalid Social Security Format" )

End If

End Property



```

Public Property Get SocialSecurityNumber()
SocialSecurityNumber = ssn
If regularExpression.Test( expression ) Then
    Validate = True
Else
    Validate = False
End If
End Property

Public Function ToString()
    ToString = name & Space( 3 ) & age & Space( 3 ) & ssn
End Function

End Class ' Person

Sub cmdButton_OnClick()
' Declare object reference
' Instantiate Person object

With p
    FirstName = Document.Forms(0).textBox1.Value
    Age = CInt( Document.Forms(0).textBox2.Value )
    SocialSecurityNumber = _
        Document.Forms(0).textBox3.Value
    Call MsgBox( .ToString() )
End With

End Sub

```

```

-->
</script>
</head>

<body>
<form action = "">Enter first name
<input type = "text" name = "txtBox1" size = "10" />
<p>Enter age
<input type = "text" name = "txtBox2" size = "5" /></p>
<p>Enter social security number
<input type = "text" name = "txtBox3" size = "10" />
</p><p>
<input type = "button" name = "cmdButton"
value = "Enter" /></p>
</form>
</body>
</html>

```

```

Private Function Validate( expression )
    Dim regularExpression
    Set regularExpression = New RegExp
    regularExpression.Pattern = "^\\d{3}-\\d{2}-\\d{4}$"
    Dim p
    Set p = New Person
End Function

```

---

## 14.11. Let us Sum UP

---

Procedures are the logical parts into which a program is divided. The code inside a procedure is run when the procedure is called. A procedure can be called with a Call statement in another procedure, or it can be triggered by an event such as a button click.

There are two types of procedures in VBScript: Sub procedures and Function procedures. Sub procedures are blocks of code that are wrapped in the Sub...End Sub keywords. A Sub can take arguments and process them within the Sub procedure. A Sub can call other procedures, but it can't return a value generated to the calling procedure directly.

A Function procedure works just like a Sub procedure. It can take arguments and call other procedures. Most importantly, Function procedures return a value to the calling procedure.

Arguments can be used to pass data to either Sub or Function procedures. When calling a procedure, you can simply use the procedure name followed by arguments, which are separated by commas:

When you create and use a function, the return value of the function is held by a variable with the name of the function

The message box is one of the most useful functions in VBScript. The two types of message boxes available to you are the message box and the input box.

### String Functions

Asc	Chr	InStr	Startpos
String1	String2	Type	LCase
Left	LTrim	Mid	Right
RTrim	Str()	StrComp	String
Trim	UCase	Val	

### Conversion Functions

CByte	CDbl	CInt	CLng
CStr	CVErr		

## Math Functions

Abs	Array	Atn	Exp
Hex	Int	Fix	Log
Oct	Rnd	Sgn	Sqr
Sin	Tan		

## Time and Date Functions

Date	DateSerial	DateValue	Day
Hour	Year	Weekday	Minute
Month	Now	Second	Time
TimeSerial	TimeValue		

## Classes and Objects

Objects **encapsulate** (i.e., wrap together) data (**attributes**) and methods (**behaviors**); the data and methods of an object are intimately related. Objects have the property of **information hiding**.

In VBScript, the unit of object-oriented programming is the **Class** from which objects are **instantiated** (i.e., created). **Methods** are VBScript procedures that are encapsulated with the data they process within the “walls” of classes. VBScript programmers can create their own **user-defined types** called **classes**. Classes are also referred to as **programmer defined types**. Each class contains data as well as the set of methods that manipulate the data. The data components of a class are called **instance variables**. Just as an instance of a Variant is called a **variable**, an instance of a class is called an **object**.

This kind of **software reusability** speeds the development of powerful, high-quality software. **Rapid applications development (RAD)** is of great interest today. Packaging software as classes out of which we make objects makes more significant portions of major software systems reusable. On the Windows platform, these classes have been packaged into class libraries, such as Microsoft’s **MFC (Microsoft Foundation Classes)**, that provide C++

programmers with reusable components for handling common programming tasks, such as the creating and manipulating of graphical user interfaces.

Classes often provide Public methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) Private instance variables. These methods are special methods in VBScript called **Property Let**, **Property Set** and **Property Get** (collectively these methods and the internal class data they manipulate are called **properties**).

Procedures Property Let and Property Set differ in that Property Let is used for nonobject subtypes (e.g., integer, string, byte) and Property Set is used for object subtypes.

---

#### **14.12. Lesson end Activities**

---

1. What is the different between function and procedure?
2. List the Built-in function in VBScript.

---

#### **14.13. Check your progress**

---

1. Write a programe to implement class in VBScript.
2. What are the string functions available in VBScript?

---

#### **14.14. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.microsoft.com](http://www.microsoft.com)



## **Active Server Pages – I**

---

### **Contents**

#### **15.0. Aim and Objective**

#### **15.1. Introduction**

#### **15.2. How ASP works?**

#### **15.3. Client Side Scripting Vs Server Side Scripting**

#### **15.4. Configuring your web server for ASP**

#### **15.5. Active Data Objects**

#### **15.6. Accessing Files on Server**

#### **15.7. Session and Cookies**

#### **15.8. Let us Sum Up**

#### **15.9. Lesson end Activities**

#### **15.10. Check your Progress**

#### **15.11. Reference**

---

### **15.0. Aim and Objects**

---

- **To learn ASP for web page development**
- **To understand client side and server side scripting**
- **To understand Active Data Objects**
- **To understand session and cookies**

---

### **15.1. Introduction**

---

Active Server Pages are browser independent. The browser only sees pure HTML pages; no vendor proprietary programs or extensions are needed for customers to use ASP applications. ASP is easy to learn. ASP hides the code from the customer (and the hacker). ASP gives an efficient link to the many databases that comply with the Open Database Connection (ODBC) standard.

---

## 15.2. How ASP works?

---

The Internet Information Server (IIS) versions 3.0, IIS version 4.0 and the Personal Web Server (PWS) can process active server pages. The server knows that ASP code is in the file from the ASP extension. ASP is included in Microsoft's Internet Information Server (IIS). ASP runs as an add-on dll to Microsoft's Personal Web Server.

ASP Delimiters are much like HTML:

```
<% Code %>
```

By default the code used to write ASPs is VBScript but Javascript, PERL or C++ can be used. The default language is set in the server's NT operating system registry by the server management software.

For Javascript [If statements](#):

```
<SCRIPT LANGUAGE=Javascript RUNAT=Server>If (iPrice=2)  
Buy="Yes"</script>
```

or

```
<%@ LANGUAGE=Javascript %>  
<% If (iPrice=2) Buy="Yes" %>
```

or

```
<% If (iPrice=2) %>  
<b>You should buy.</b>  
<% Else %>  
<b>Don't buy that!</b>
```

VBScript is a "lightweight" subset of Visual Basic with limitations imposed for reasons of security, portability, and performance. VBScripts can't read from or write to local drives or make system calls. Visual Basic and VB Script are Microsoft languages and work only with Microsoft products. Microsoft's guide to VBScript is at <http://www.microsoft.com/vbscript>

---

## 15.3. Client Side Scripting VS Server Side Scripting

---

One can add smarts to web pages using either client side or server side processes. The client side is the customer's browser such as Netscape Navigator or Microsoft Internet Explorer. The server side processes are performed on the



hypertext or web server - be it a personal computer or a mainframe computer. [Common Gateway Interface](#) (CGI) and perl scripts are server side processes. The variables are collected from the web page and passed via the CGI to the server where it is processed by [perl scripts](#). [Javascript](#) (not to be confused with Java) and VBScript are computer languages that typically execute on the client side (i.e. in the browser). Unlike CGI, these scripts are on the same page as HTML. A block of ASP code, some HTML, some more code, etc.

Active Server Pages (ASP) is an object-based server-side scripting environment. Both VBScript and JavaScript can be used for server-side scripting under ASP. However, the referred language for server-side scripting is usually VBScript. Client-side and server-side scripting code can be used within the same page.

The web server must be setup to allow *script* or *execute* permissions on the virtual directory where the ASP code will reside. The file containing the ASP code must have .asp extension otherwise the web server does not process the server-side script code.

The server-side script code uses <% and %> tags to identify the code that will get executed on the server. The scripting language in this case is the default language set up on the server.

It is a good practice to identify the server-side scripting language for each page by including a statement at the top of the page as shown below:

```
<% @ LANGUAGE=VBScript %>
```

The server-side scripting language can be changed for functions/subs within a page by using the SCRIPT tag along with RUNAT=SERVER identification.

```
<SCRIPT LANGUAGE=JavaScript RUNAT=SERVER>
```

```
.....
```

```
.....
```

```
</SCRIPT>
```

Example:

```
<!-- Serversc1.asp -->

<% @ LANGUAGE=VBScript%>

<HTML>

  <HEAD>

    <TITLE> Server side scripting - Reporting the Date and Time on the
Server </TITLE>

  </HEAD>

  <BODY>

    <H2> <CENTER> Server Side Scripting </CENTER> </H2>

    <H3> <CENTER> Date and Time on the server =

<%

  dim t1

  t1 = now

  Response.write t1

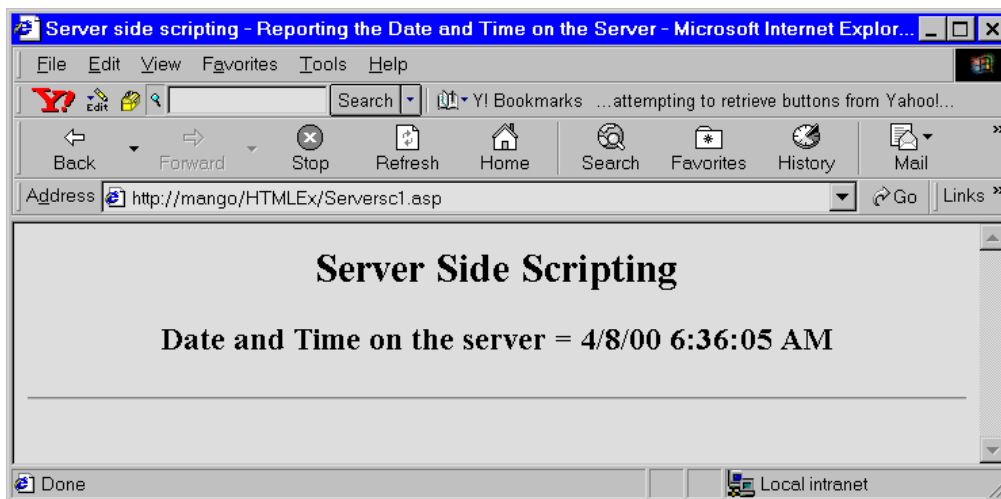
%>

  </CENTER></H3>

  <HR>

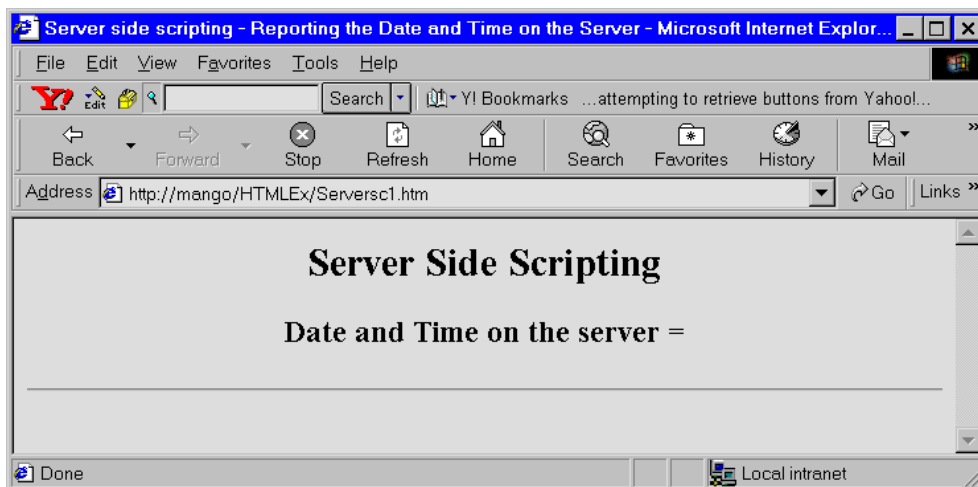
</BODY>

</HTML>
```



Note that if you save the *Serversc1.asp* file in the previous example as *Serversc1.htm*, and retrieve it from your browser, the web server does not execute the server-side script code and passes it as is to the Client browser.

```
<!-- Serversc1.htm -->
<% @ LANGUAGE=VBScript%>
<HTML>
  <HEAD>
    <TITLE> Server side scripting - Reporting the Date and Time on the
Server </TITLE>
  </HEAD>
  <BODY>
    <H2> <CENTER> Server Side Scripting </CENTER> </H2>
    <H3> <CENTER> Date and Time on the server =
<%
  dim t1
  t1 = now
  Response.write t1
%>
    </CENTER></H3>
    <HR>
  </BODY>
</HTML>
```



A file containing server-side script code must have the extension .asp.

Change the file name back to *Serversc1.asp* and examine it in the browser. Then try to view the source from the browser, you will note that the server-side script code is not visible to the browser, it only gets the HTML statements.

Result of *View -> Source* from the browser when the *Serversc1.asp* file is being viewed.

```
<!-- Serversc1.asp -->
<HTML>
  <HEAD>
    <TITLE> Server side scripting - Reporting the Date and Time on the
Server </TITLE>
  </HEAD>
  <BODY>
    <H2> <CENTER> Server Side Scripting </CENTER> </H2>
    <H3> <CENTER> Date and Time on the server =
4/8/00 6:49:57 AM
    </CENTER></H3>
    <HR>
  </BODY>
</HTML>
```

---

## 15.4. Configuring your Web Server for Active Server Pages

---

### Personal Web Server

In order to run Active Server Page code, you must have one of Microsoft's Web Server products installed. Microsoft's primary web server product is "Internet Information Server" (IIS) which is a rather large application designed to run on Windows NT or Windows 2000 based Internet web servers. It is normally not run in a desktop PC where you would typically do the development of your web pages. To make the task of developing and testing web pages easier, a scaled down version of IIS called Personal Web Server (PWS) is available from

Microsoft that can be run on any Windows 95, 98, NT or 2000 workstation. PWS does not require any connection to the Internet therefore you can do all your development and testing off-line.

Installation:

To Install PWS on Windows 98:

1. insert your Windows 98 operating system CD into your CD-ROM drive.
2. Click on the Start button and choose Run.
3. In the Run Dialog box type X:\add-ons\pws\setup.exe (where X is the drive letter of your CD-ROM drive) and click OK.
4. Follow the instructions on the screen to install PWS.

### **Internet Information Server**

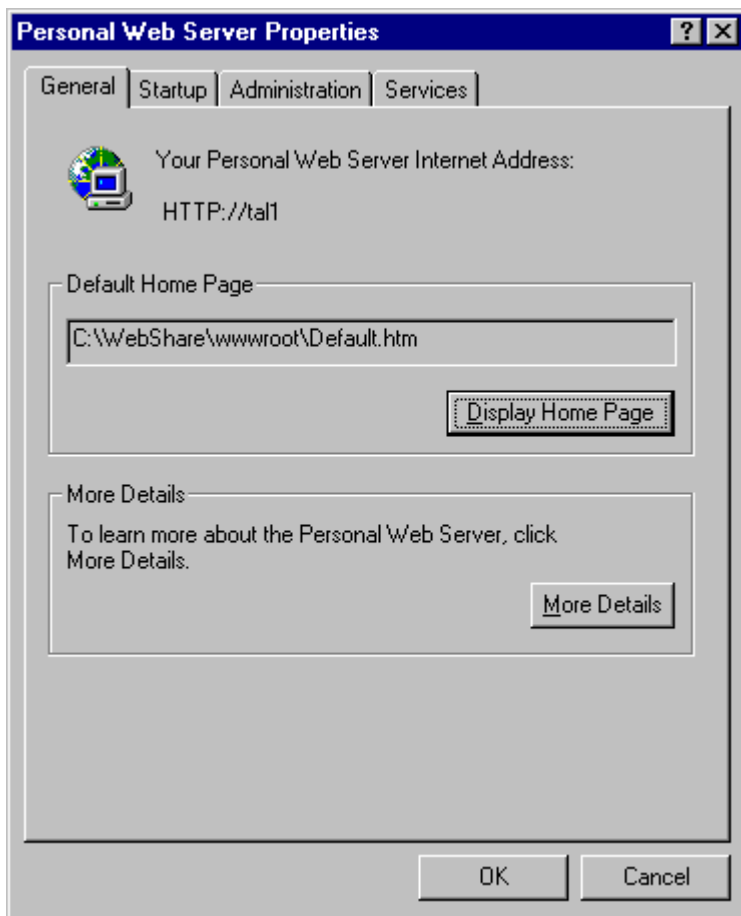
Microsoft Internet Information Server (IIS) is a web server that integrates into the Windows NT Server. IIS allows you to publish information on the World Wide Web and to run multiple business applications using ASP.

Installation

Make sure you have Service Pack 3 (or higher) and Microsoft Internet Explorer 4.01 (or higher - recommended) installed on your Windows NT Server before you install the Windows NT 4.0 Option Pack. It is recommended that you apply Service Pack 4 for Windows NT after installing IIS. Note: You must reapply Service Pack 4 to your computer when you install a new component of the Windows NT operating system or the Windows NT 4.0 Option Pack.

### **Getting Started**

Before jumping straight into Server Side Scripting for the TALtech ActiveX Plus, we recommend testing your web server to make sure that it is correctly configured to work with Active Server Pages (ASP). Some older versions of PWS, such as the one installed with Microsoft FrontPage 98 (AKA "FrontPage Web Server") do not install support for ASP by default. If your Personal Web Server looks like this:



Then you may wish to upgrade it. Below is a simple ASP file that can be used to test the server.

Copy and Paste the code below into Notepad. Save the file as Test1.asp into the home directory of the default Web site that was installed with PWS or IIS (Usually C:\Webshare\wwwroot or C:\InetPub\wwwroot by default.

```
<HTML><BODY>
Output:<BR>
<%
intS = 60*60
%>
There are
<%
Response.Write intS
%>
seconds in an hour.
</BODY></HTML>
```

You can access the ASP page Test1.asp by typing the following URL in your Web Browser: <http://localhost/Test1.asp>

You should see:

Output:

There are 3600 seconds in an hour.

---

## 15.5. Active Data Objects

---

Objects are the building blocks of Java, Javascript, Visual Basic and other object oriented languages. Objects have:

- Properties or attributes - are the characteristics of the object
- Methods - are the tricks the object can do
- Events - are when the object does its tricks

The server uses the Active Data Object (ADO), an Active-X object that handles all the data from server to browser. The ADO has various drivers for different databases and can work with any ODBC database.

There are six built-in ASP objects that simplify web development. These are:

Application

Session

ObjectContext

Request

Response

Server

ASP has several built in Objects. The five Objects we will learn about here are:

1. **Application objects** - for managing information for a web application created with ASP. Applications objects are for the application as a whole and start when the web server starts. Application objects are defined in the [global.asa](#) file.

2. **Session objects** - for managing information concerning the user's current Web session. Session objects last for the time a user is on the site. Session objects are created when individuals enter the application and continue until the timeout occurs. Session objects can be defined in any file including the [global.asa](#) file

3. **Response objects** - for sending information to the client or user
4. **Request objects** - for receiving information from the user
5. **Server objects** - for providing information about the server.

In addition to these **Server-side Objects**, ASP has a number of **Server-side Components**. Components are objects that one must explicitly add to the ASP application. The one we will focusing on is the Database Access Objects (DAO) and ActiveX Data Objects (ADO).

Response and Request Objects

**The Response Object has the following Methods:**

```
Response.Write("Hello World!")  
' This is a comment  
<%= "Bam" %>  
Response.Redirect("new_page.asp")  
Response.Cookies("Name") = "1234, etc."  
Response.Cookies("Name").Expires = Now()+7 (or date)
```

**Request Methods:**

```
request.form("txtName")  
request.cookies("Name")  
Request.ServerVariables("SERVER_NAME")
```

**Server Method:**

```
Server.CreateObject("VBOject.Class")
```

The list we will use the most are Response.Write and Request.form.

You can embed HTML tags inside the ASP Response.write method to format the output e.g., if you wanted the date and time on the server to appear on the next line, you would change the Response.write statement as:

```
Response.write "<BR>" & t1
```

**<%= variable or expression %>** replaces the value of variable or expression and sends it to the browser.

Example:

```
<!-- Serversc3.asp -->  
<% @ LANGUAGE=VBScript%>
```



```

<HTML>

<HEAD>

  <TITLE> Server side scripting - Reporting the Date and Time on the
Server </TITLE>

</HEAD>

<BODY>

  <H2> <CENTER> Server Side Scripting </CENTER> </H2>

  <H3> <CENTER> Date and Time on the server = <%= now %>

  </CENTER></H3>

  <HR>

</BODY>

</HTML>

```

### **Response.expires**

By placing `<% Response.expires = 0 %>`, you can indicate to the browser not to cache the page. This way the time will be obtained from the server each time the user comes to this page.

```

<% @ LANGUAGE=VBScript%>
<% Response.expires=0 %>
  <!-- Serversc5.asp -->

<HTML>

<HEAD>

  <TITLE> Server side scripting - Reporting the Date and Time on the
Server </TITLE>

</HEAD>

<BODY>

  <H2> <CENTER> Server Side Scripting </CENTER> </H2>

<%
  dim t1
  t1 = now
%>

  <H3> <CENTER> Date and Time on the server = <%= t1 %>

```

```

</CENTER></H3>

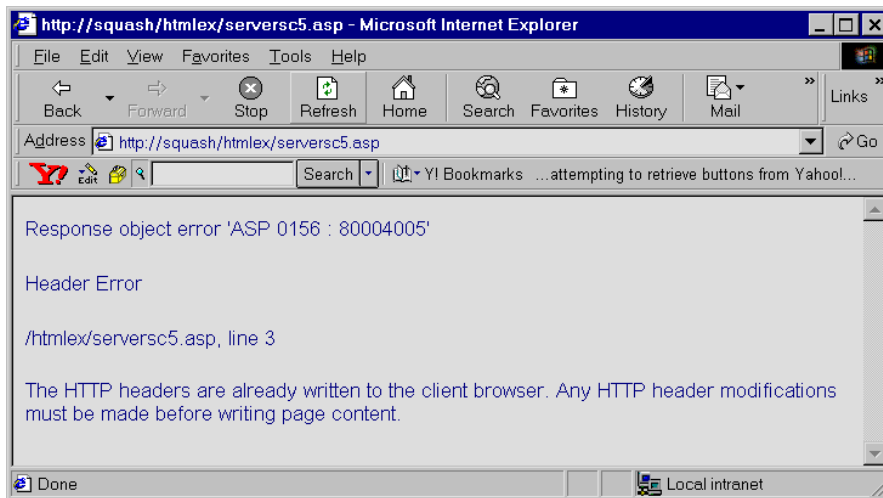
<HR>

</BODY>

</HTML>

```

If you try viewing the above page in your browser, you will get an error message:



The reason for the error is that the *Response.expires=* should be specified before any HTML content is sent to the page i.e., this needs to be in the header section in the response. So the HTML comment line which appears two lines before `<% Response.expires=0 %>` is the source of the problem. Modify the first few lines of the page as shown below:

```

<% @ LANGUAGE=VBScript%>
<% Response.expires=0 %>
<!-- Serversc5.asp -->

```

Now try viewing this page and retyping part of the *url* to see if the time is correctly updated.

You can also specify a relative or absolute time for the page to expire in the cache of the browser, e.g.,

```
Response.expires=60           page expires in 60 minutes
```

```
Response.ExpiresAbsolute=#6/1/2000 06:30:00#
```

Or

```
Response.ExpiresAbsolute=#June 1, 2000 06:30:00#
```

<SCRIPT RUNAT=SERVER> tag is used to define a function or sub that will be executed on the web server. This tag is particularly useful in mixing different scripting languages on the server.

Example: Change the Serversc5.asp file as shown below.

```
<% @ LANGUAGE=VBScript%>
<% Response.expires=0 %>
<!-- Serversc6.asp -->
<HTML>
  <HEAD>
    <TITLE> Server side scripting - Reporting the Date and Time on the
    Server </TITLE>
  </HEAD>
  <BODY>
    <H2> <CENTER> Server Side Scripting </CENTER> </H2>
    <SCRIPT LANGUAGE=VBSCRIPT RUNAT=SERVER>
      dim t1
      t1 = now
    </SCRIPT>
    <H3> <CENTER> Date and Time on the server = <%= t1 %>
    </CENTER></H3>
    <HR>
  </BODY>
</HTML>
```

Try viewing the above file in your browser and you will see that it does not show any date and time. Now modify the above file to create a function that will return date and time as shown below:

```
<% @ LANGUAGE=VBScript%>
<% Response.expires=0 %>
<!-- Serversc7.asp -->
<HTML>
```

```

<HEAD>

  <TITLE> Server side scripting - Reporting the Date and Time on the
Server </TITLE>

</HEAD>

<BODY>

  <H2> <CENTER> Server Side Scripting </CENTER> </H2>
<SCRIPT LANGUAGE=VBSCRIPT RUNAT=SERVER>

```

```
Function ServerDateTime()
```

```
  dim t1
```

```
  t1 = now
```

```
  ServerDateTime = t1
```

```
End Function
```

```
</SCRIPT>
```

```
  <H3> <CENTER> Date and Time on the server = <%= ServerDateTime
%>
```

```
  </CENTER></H3>
```

```
  <HR>
```

```
</BODY>
```

```
</HTML>
```

<SCRIPT LANGUAGE=VBSCRIPT RUNAT=SERVER> should be used to identify server-side functions and subs.

Example of mixing Server-side scripting languages:

```
<% @ LANGUAGE=VBScript %>
```

```
<% Response.expires=0 %>
```

```
<!-- Serversc8.asp -->
```

```
<HTML>
```

```
  <HEAD>
```

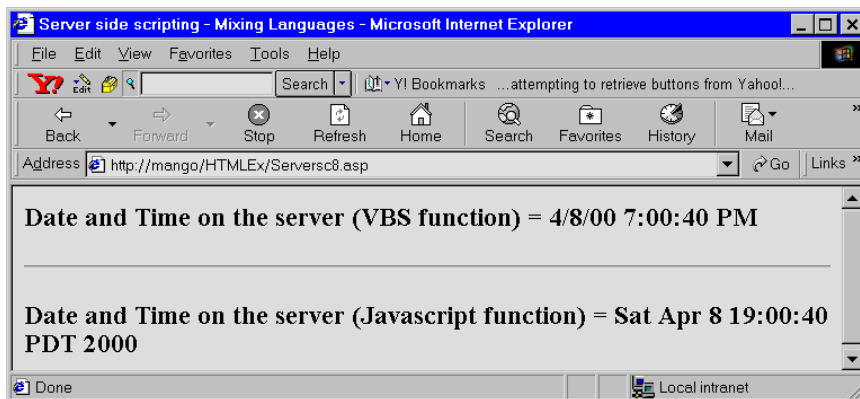
```
    <TITLE> Server side scripting - Mixing Languages </TITLE>
```

```
  </HEAD>
```

```

<BODY>
<SCRIPT LANGUAGE=VBSCRIPT RUNAT=SERVER>
Function ServerDateVBS()
    dim t1
    t1 = now
    ServerDateVBS = t1
End Function
</SCRIPT>
<SCRIPT LANGUAGE=JAVASCRIPT RUNAT=SERVER>
function ServerDateJS() {
    var t1;
    t1 = new Date();
    return t1;
}
</SCRIPT>
    <H3> Date and Time on the server (VBS function) = <%=
=ServerDateVBS %> </H3>
    <HR>
    <H3> Date and Time on the server (Javascript function) = <%=
ServerDateJS() %> </H3>
</BODY>
</HTML>

```



## Server-side Includes – reusable code blocks

When a web site involves quite a bit of script code, it is always a good idea to break the program into several files. Commonly used functions and subs can be placed in a file and this file can be included where ever these functions/subs are needed.

```
<!--#INCLUDE FILE=filename --> or <!--#INCLUDE VIRTUAL=filename -->
```

Virtual indicates the file relative to the virtual web server directory.

Example: Create a function called *greeting* and place it in the *greet.inc* file in the *include* subdirectory as shown below.

```
<!-- greet.inc.txt -->
<SCRIPT LANGUAGE=VBSCRIPT RUNAT=SERVER>
Function greeting()
    dim t1, strMsg
    t1 = now
    If Hour(t1) < 12 Then
        strMsg="Good Morning"
    ElseIf Hour(t1) < 18 Then
        strMsg="Good Afternoon"
    Else
        strMsg = "Good Evening"
    End If
    greeting = strMsg
End Function
</SCRIPT>
<% @ LANGUAGE=VBScript %>
<% Response.expires=0 %>
<!-- Serversc9.asp -->
<HTML>
<HEAD>
```

```

<TITLE> Server side scripting - Server-side Includes </TITLE>

</HEAD>

<BODY bgcolor="aqua">

<!-- #INCLUDE FILE="include/greet.inc.txt" -->

Greetings from the server:

<FONT FACE="Comic Sans MS" SIZE=6 COLOR="#FF00FF">

<%= greeting %>

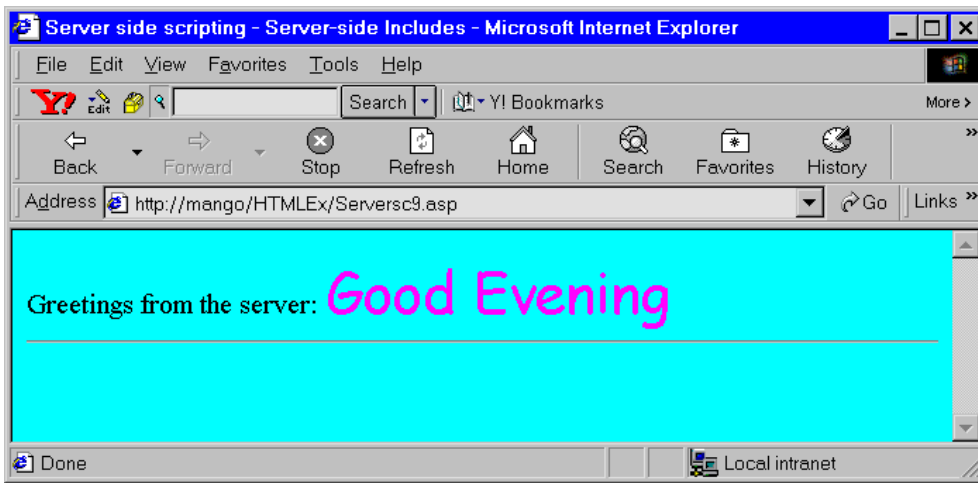
</FONT>

<HR>

</BODY>

</HTML>

```



**Security Concern:** Try viewing the include file directly in the browser i.e., type the url as:

```
http://localhost/MyWeb/include/greet.inc.txt
```

Even though you will not be able to view anything in the page, if you try to view the source (View->source from the browser menu), you will be able to see the ASP function code. In some practical situations, we may not want the client to be able to take a look at our ASP code, hence any extension other than an *.asp* for the INCLUDE files does not protect your ASP code from the client.

All INCLUDE files should have the extension *.asp*.

Rename the greet.inc.txt file to greet.asp and also make the corresponding change in the Serversc9.asp file as:

```
<!-- #INCLUDE FILE="include/greet.asp" -->
```

Now the client browser cannot see the code in the greet.asp file even if this file is viewed in the browser directly.

### **Response.expires=0 revisited**

Response.expires=0 causes the browser to not to cache the web page. This may be important for periodically changing data in the page such as server time, or stock quotes etc.. However, from performance point of view, setting expires=0 also causes a refetch of the page from the server. If the page involves a little dynamic data but quite a bit of images that do not change over time, then the page loading could become slow.

It is possible to break the page into a few different asp files some having a setting of Response.expires=0 and some with a greater expiration time. The asp files are not included by an #INCLUDE statement but rather by a client-side JavaScript **SRC** statement.

Example:

```
<% @ LANGUAGE=VBScript%>
<% Response.expires=0 %>
<!-- Serversc10.asp -->
<HTML>
  <HEAD>
    <TITLE> Server side scripting - Date and Time on the Server </TITLE>
  </HEAD>
  <BODY>
    <H2> <CENTER> Server Side Scripting </CENTER> </H2>
  <%
    dim t1
    t1 = now
  %>
```



<H3> This part of the page is not cached </H3>

<H3> <CENTER> Date and Time on the server = <%= t1 %>

</CENTER></H3>

<HR>

This is an image file which will be cached for some time:

<SCRIPT LANGUAGE=JAVASCRIPT SRC="IMG.asp">

</SCRIPT>

</BODY>

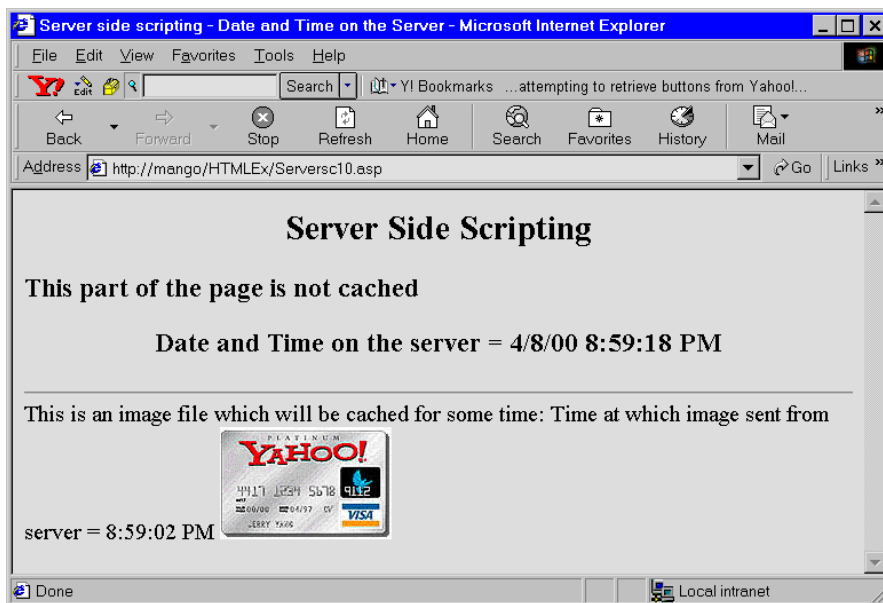
</HTML>

<% Response.expires=1 %>

<!-- IMG.asp -->

document.writeln(" Time at which image sent from server = <%=time%>");

document.writeln('<IMG SRC="yahooocard.gif" BORDER=0>');



Setting Page expiration relative to current time on the server.

Example:

```
<% @ LANGUAGE=VBScript%>
```

```
<%
```

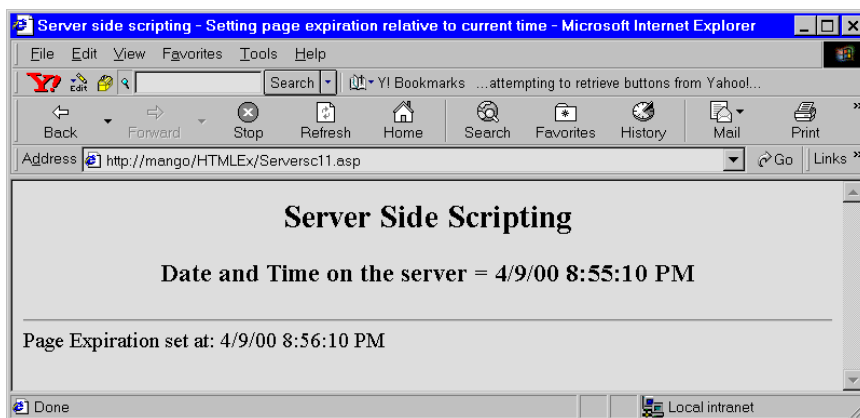
```
dim exp1
```

```
exp1 = DateAdd("n",1,now) 'n specifies minutes, m specifies month
```

```

Response.ExpiresAbsolute=exp1
%>
<!-- Serversc11.asp -->
<HTML>
  <HEAD>
    <TITLE> Server side scripting - Setting page expiration relative to
current time </TITLE>
  </HEAD>
  <BODY>
    <H2> <CENTER> Server Side Scripting </CENTER> </H2>
<SCRIPT LANGUAGE=VBSCRIPT RUNAT=SERVER>
Function ServerDateTime()
  dim t1
  t1 = now
  ServerDateTime = t1
End Function
</SCRIPT>
  <H3>  <CENTER>   Date   and   Time   on   the   server   =   <%
Response.write(ServerDateTime()) %>
  </CENTER></H3>
  <HR>
  <% Response.write("Page Expiration set at: " & exp1) %>
</BODY>
</HTML>

```



## Redirecting to another Page

We can use the `Response.redirect` method to redirect the user to another page. This is needed quite often when the web site location changes or after the user accesses a page that requires log-in, the user will be redirected to the log-in page.

The `Response.redirect` is part of the header and thus requires that no HTML output has been written to the page.

Example:

```
<% @ LANGUAGE=VBScript%>
<%
    Response.expires=20 '20 minute expiration
    If Hour(now) > 12 Then
        Response.redirect "serversc10.asp"
    else
        Response.redirect "http://www.amazon.com"
    end if
%>

<!-- Serversc12.asp -->
<HTML>
    <HEAD>
        <TITLE> Server side scripting - Response.redirect method </TITLE>
    </HEAD>
    <BODY>
        <H2> <CENTER> Server Side Scripting </CENTER> </H2>
        <H3>  <CENTER>   Date   and   Time   on   the   server   =   <%
Response.write(Now) %>
        </CENTER></H3>
        <HR>
    </BODY>
</HTML>
```

## Buffering Output

Both `Response.expires=0` and `Response.redirect` require that no HTML content is written before executing them. However, in some dynamic situations, we may want to change the expiration time or redirecting to a different site. This can be accomplished by buffering the output by setting **`Response.buffer=TRUE`**

If page buffering is on, then expiration can be changed any time later, even if some HTML content has been written to the buffer but not sent to the browser.

If the page is being buffered, then it can be sent from the server to the browser either by executing **`Response.flush`** or **`Response.end`** method.

Example:

```
<% @ LANGUAGE=VBScript%>
<%
    Response.buffer=True 'This is required if redirection is needed
                        'after some content has been written
    Response.expires=1 '1 minute expiration
%>
<!-- Serversc13.asp -->
<HTML>
<HEAD>
    <TITLE> Server sside scripting - Response.redirect method </TITLE>
</HEAD>
<BODY>
    <H2> <CENTER> Server Side Scripting - Response.redirect </CENTER>
</H2>
<%
    Response.write "We are redirecting you to a different page"
    If Hour(now) < 12 Then
        Response.redirect "serversc5.asp"
    else
```

```

        Response.redirect "http://www.amazon.com"
    end if
%>

    <H3> <CENTER> Date and Time on the server = <%
Response.write(Now) %>

    </CENTER></H3>

    <HR>

    </BODY>

</HTML>

```

Example: Use of flush, clear and end methods of Response object

```

<% @ LANGUAGE=VBScript%>

<%

    Response.buffer=True

    Response.expires=1 '1 minute expiration

%>

<!-- Serversc14.asp -->

<HTML>

    <HEAD>

        <TITLE> Server side scripting - Response.redirect method </TITLE>

    </HEAD>

    <BODY>

        <H2> <CENTER> Server Side Scripting - Response.redirect </CENTER>

    </H2>

    <%

        if Hour(Now) < 12 Then 'Try changing the "<" to ">"

            Response.write ("Server time = " & Now & "<BR>")

            Response.write "We are flushing the output so far and ending
response"

            Response.flush

```

```

Response.end 'no further output will be sent
          'actually Response.end flushes the output also
else
Response.clear 'all previous output is cancelled
Response.expires=0
Response.write("This content expires quickly")
end if
%>
<H3> <CENTER> Date and Time on the server = <%
Response.write(Now) %>
</CENTER></H3>
<HR>
</BODY>
</HTML>

```

### **Response.ContentType**

This identifies to the browser how the content should be displayed. For example, if `Response.ContentType="text/plain"` then the browser does not interpret HTML tags. However if the `Response.ContentType="text/html"` then the HTML tags are taken into account. If `Response.ContentType="application/msword"` then the internet explorer displays the page by opening MS WORD in the browser.

ContentType should be set before sending any content to the browser i.e., it is a header specification.

Example:

```

<% @LANGUAGE=VBSCRIPT %>
<% 'Serversc15.asp
Response.ContentType="text/plain"
'Change the content type to text/html and view the page
'to see how html tags are interpreted correctly
Response.expires=0
%>

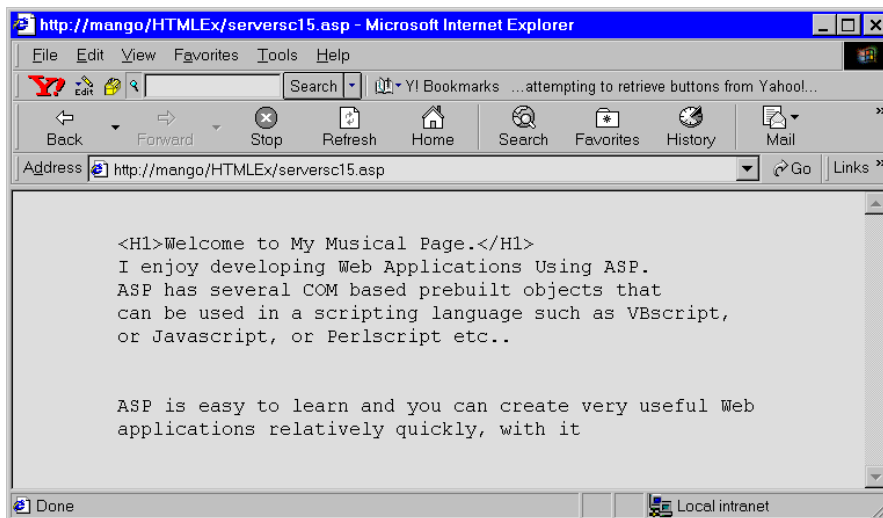
```

<H1>Welcome to My Musical Page.</H1>

I enjoy developing Web Applications Using ASP.

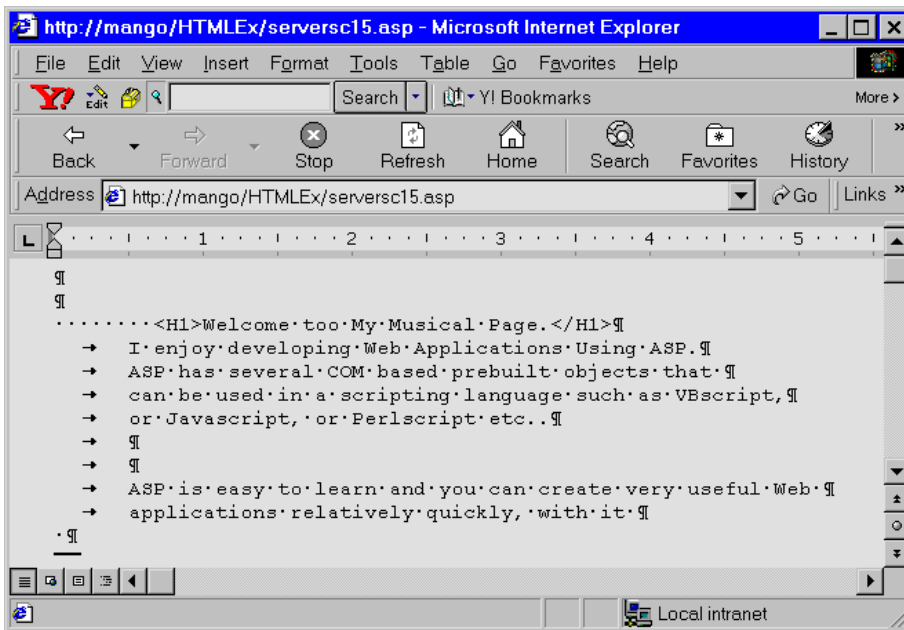
ASP has several COM based prebuilt objects that can be used in a scripting language such as VBscript, or Javascript, or Perlscript etc..

ASP is easy to learn and you can create very useful Web applications relatively quickly, with it



**Try changing the line** `Response.ContentType="text/plain"` to `Response.ContentType="text/html"` and then view it in the browser.

Try changing the `ContentType` to `"application/msword"` and view it in the browser.



## Request Object

One important collection of the request object is `ServerVariables` which provides information about the environment variables such as IP address of the client, length of the posted content, browser information etc..

```
strSelf = Request.ServerVariables("SCRIPT_NAME")
```

returns virtual path of the asp page itself

```
nLength = Request.ServerVariables("CONTENT_LENGTH")
```

returns length of the posted content (POST method)

`Request.ServerVariables("HTTP_headername")` returns the value of a particular HTTP header e.g.,

`Request. ServerVariables("HTTP_USER_AGENT")` returns the browser name and platform on which it is running.

`Request. ServerVariables("HTTP_REFERER")` returns the url of the web page that invoked this asp page.

`Request. ServerVariables("REMOTE_ADDR")` returns the client's IP address.

You can determine all HTTP headers sent from the browser by executing the following code:

```
<%= Replace(Request. ServerVariables("ALL_RAW"), vbCrLf,"<BR>") %>
```



Example:

```
<% @ LANGUAGE=VBSCRIPT %>

<% 'Serversc16.asp %>

<HTML>

<HEAD>

<TITLE>

    Test of HTTP headers determined from the Request Object

</TITLE>

<HEAD>

<BODY>

<H2> Some HTTP headers as determined from the Request object </H2>

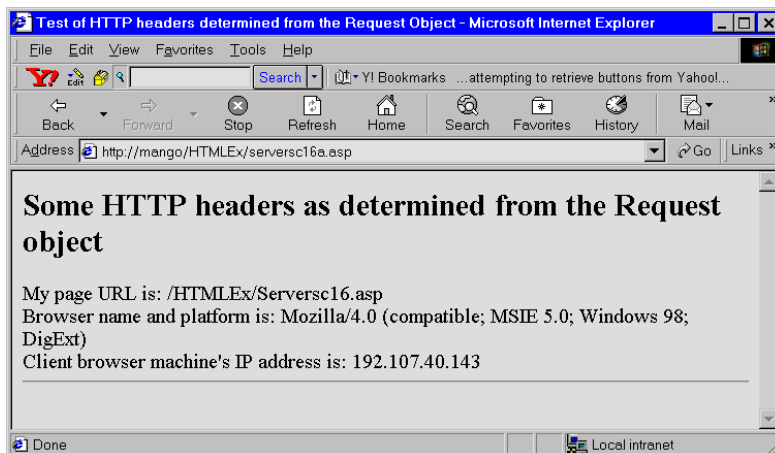
<%

    strSelf = Request.ServerVariables("SCRIPT_NAME")
    Response.write("My page URL is: " & strSelf)
    strBinfo = Request. ServerVariables("HTTP_USER_AGENT")
    Response.write("<BR>Browser name and platform is: " & strBinfo)
    strClientIP = Request. ServerVariables("REMOTE_ADDR")
    Response.write("<BR>Client browser machine's IP address is: " &
strClientIP) %>

<HR>

</BODY>

</HTML>
```



## GET and POST methods for Submitting Forms to the Server

### GET method

When GET method is used to submit a FORM to the server, the server script can use the Request.querystring collection to determine the values of different fields. In the GET method, querystring is appended to the URL when the form is submitted. Each element in the form is identified by its name=value. The different elements are separated by & e.g., an ID and password form when submitted using the GET method will have the following querystring:

**http://mango/HTMLEx/Serversc17a.asp?USERID=965&PASSWORD=45&cmdLogin>Login**

The value of querystring can be determined by:

```
Request.ServerVariables("QUERY_STRING")
```

Example:

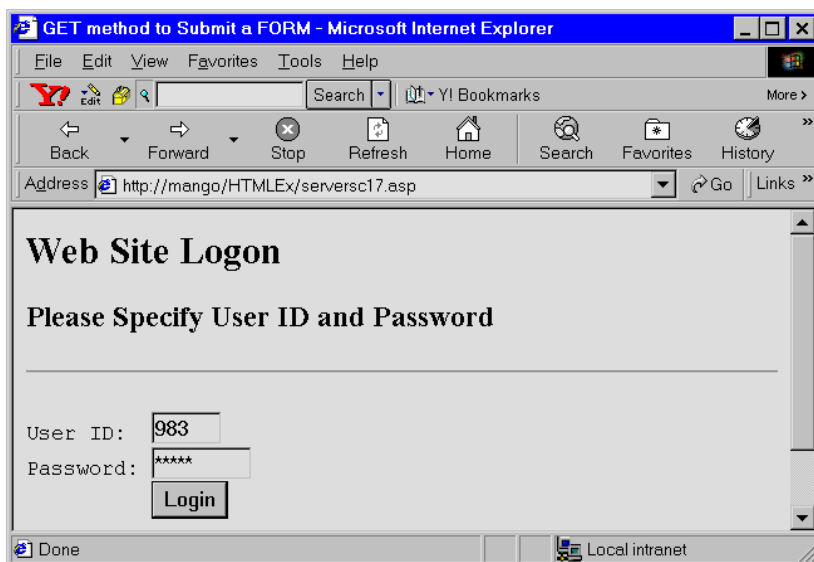
```
<% @LANGUAGE=VBSCRIPT %>
<% 'Serversc17.asp %>
<HTML>
  <HEAD>
    <TITLE> GET method to Submit a FORM</TITLE>
  </HEAD>
  <BODY>
    <H2> Web Site Logon </H2>
    <H3> Please Specify User ID and Password</H3>
    <HR>
      <FORM method=GET ACTION="Serversc17a.asp">
        User ID:   <INPUT NAME="USERID" SIZE="5" MAXLENGTH="5"
        VALUE="673">
        Password: <INPUT TYPE="password" NAME="PASSWORD" SIZE="8"
        MAXLENGTH="8" VALUE="">
          <INPUT TYPE=SUBMIT VALUE="Login" NAME=cmdLogin>
        <HR>
      </FORM>
    </BODY>
```

```

</HTML>

<% @LANGUAGE=VBSCRIPT %>
<% 'Serversc17a.asp %>
<HTML>
  <HEAD>
    <TITLE> Reading the Query String</TITLE>
  </HEAD>
  <BODY>
    <H2> User ID and Password </H2>
    <HR>
    <%
      Response.write("UserID      submitted      =      "      &
Request.querystring("USERID"))
      Response.write("<BR>Password      submitted      =      "      &
Request.querystring("PASSWORD"))
    %>
  </BODY>
</HTML>

```





The target of a GET or POST method can be the page itself, e.g. the above program can be modified as:

```
<% @LANGUAGE=VBSCRIPT %>

<% 'Serversc18.asp %>

<HTML>

  <HEAD>

    <TITLE> GET method to Submit a FORM</TITLE>

  </HEAD>

  <BODY>

    <H2> Web Site Logon </H2>

    <HR>

    <% If Request.ServerVariables("QUERY_STRING") = "" Then %>

      <H3> Please Specify User ID and Password</H3>

      <FORM method=GET ACTION="Serversc18.asp">

        User ID: <INPUT NAME="USERID" SIZE="5" MAXLENGTH="5"
        VALUE="673">

        Password: <INPUT TYPE="password" NAME="PASSWORD" SIZE="8"
        MAXLENGTH="8" VALUE="">

        <INPUT TYPE=SUBMIT VALUE="Login" NAME=cmdLogin>

        <HR>

      </FORM>

    <% Else
```

```

        Response.write("UserID submitted = " &
Request.querystring("USERID"))

        Response.write("<BR>Password submitted = " &
Request.querystring("PASSWORD"))

    End If

%>

</BODY>

</HTML>

```

It is a good practice to not to hard code the asp page name in the FORM's ACTION attribute. Instead, you should use the Request.ServerVariables("SCRIPT\_NAME").

Change the following line in the above program:

```

<FORM method=GET ACTION="Serversc18.asp"> to

<FORM method=GET ACTION=
"<%=Request.ServerVariables("SCRIPT_NAME")%>">

```

## **POST method**

GET method allows only 2KB of data to be appended to the querystring. If data submitted from a form is larger, then use the POST method. In the POST method, use Request.ServerVariables("CONTENT\_LENGTH") to determine if the form has been filled or not. Also use Request.Form(*element name*) to obtain the value of an HTML form element.

Example:

```

<% @LANGUAGE=VBSCRIPT %>

<% 'Serversc19.asp %>

<HTML>

<HEAD>

    <TITLE> POST method to Submit a FORM</TITLE>

</HEAD>

<BODY>

    <H2> Web Site Feedback </H2>

```

```

<HR>

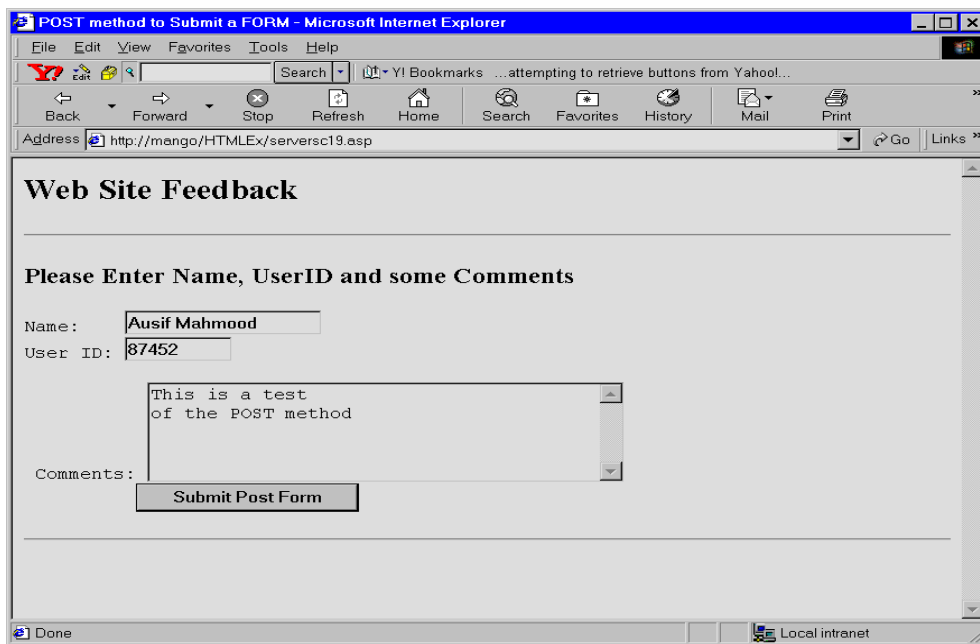
<% If Request.ServerVariables("CONTENT_LENGTH") = 0 Then %>
    <H3> Please Enter Name, UserID and some Comments </H3>
    <FORM method=POST ACTION=
"<%=Request.ServerVariables("SCRIPT_NAME")%>">
        <PRE>
Name:  <INPUT NAME="txtName" SIZE="20" MAXLENGTH="20"
VALUE="">
User ID: <INPUT NAME="txtID" SIZE="10" MAXLENGTH="10" VALUE="">

Comments: <TEXTAREA NAME=txaComments ROWS=5
COLS=40></TEXTAREA>
        <INPUT TYPE=SUBMIT VALUE="Submit Post Form"
NAME=cmdSubmit>
        <HR>
        </PRE>
    </FORM>

<% Else
    Response.write("Name submitted = " & Request.Form("txtName"))
    Response.write("<BR>User ID submitted = " & Request.Form("txtID"))
    strComm = Replace(Request.Form("txaComments"),vbCrLf,"<BR>")
    Response.write("<BR>" & strComm)
    End If
%>

</BODY>
</HTML>

```



Data validation can be done on the client side before submitting the form.

```
<% @LANGUAGE=VBSCRIPT %>

<% 'Serversc20.asp %>

<HTML>

<HEAD>

<TITLE> POST method to Submit a FORM</TITLE>

</HEAD>

<SCRIPT LANGUAGE=VBSCRIPT>

<!--

Sub cmdSubmit_OnClick()

    If (Trim(frmFB.txtName.value) = "") OR (Trim(frmFB.txtID.Value) = "")
Then
        MsgBox("You must enter a Name and ID before submitting form")
        window.event.returnValue=False
    End If
End Sub

-->

</SCRIPT>
```

```

<BODY>

  <H2> Web Site Feedback </H2>

  <HR>

  <% If Request.ServerVariables("CONTENT_LENGTH") = 0 Then %>

    <H3> Please Enter Name, UserID and some Comments </H3>

  <FORM NAME=frmFB method=POST ACTION=
  <%=Request.ServerVariables("SCRIPT_NAME")%>">

    <PRE>

    Name:  <INPUT NAME="txtName" SIZE="20" MAXLENGTH="20"
    VALUE="">

    User ID: <INPUT NAME="txtID" SIZE="10" MAXLENGTH="10" VALUE="">

    Comments: <TEXTAREA NAME=txaComments ROWS=5
    COLS=40></TEXTAREA>

    <INPUT TYPE=SUBMIT VALUE="Submit Post Form"
    NAME="cmdSubmit">

    <HR>

    </PRE>

  </FORM>

  <% Else

    Response.write("Name submitted = " & Request.Form("txtName"))

    Response.write("<BR>User ID submitted = " & Request.Form("txtID"))

    strComm = Replace(Request.Form("txaComments"),vbCrLf,"<BR>")

    Response.write("<BR>" & strComm)

    End If %>

  </BODY>

</HTML>

```



---

## 15.6. Accessing Files on the Server

---

ASP provides an ActiveX component called “Scripting.FileSystemObject” for accessing files, folders and drives on the server.

Scripting.FileSystemObject provides a few important methods to create files, open existing files for reading or writing e.g., CreateTextFile and OpenTextFile methods. These methods end up creating an object called TextStream object which has several methods for reading or writing data to a file e.g., Read, ReadLine, ReadAll, Write, WriteLn, Close.

TextStream object provides two important properties to determine the end of line or end of file when reading data from a file (AtEndOfLine, AtEndOfStream).

The following table shows the format of the file object.

FileSystemObject.OpenTextFile(fname,mode,create,format)

Parameter	Description
fname	Required. The name of the file to open
mode	Optional. How to open the file 1=ForReading - Open a file for reading. You cannot write to this file. 2=ForWriting - Open a file for writing. 8=ForAppending - Open a file and write to the end of the file.
create	Optional. Sets whether a new file can be created if the filename does not exist. True indicates that a new file can be created, and False indicates that a new file will not be created. False is default
Format	Optional. The format of the file 0=TristateFalse - Open the file as ASCII. This is default. -1=TristateTrue - Open the file as Unicode. -2=TristateUseDefault - Open the file using the system default.

The following code creates a text file (c:\test.txt) and then writes some text to the file:

Change the file name from c:\test.txt to your virtual path in a shared hosting environment; use this code as your file path instead,  
server.mappath("/test.txt").

```
<%  
dim fs,fname  
set fs=Server.CreateObject("Scripting.FileSystemObject")  
set fname=fs.CreateTextFile("c:\test.txt",true)    'change to virtual path  
in shared hosting environment  
fname.WriteLine("Hello World!")  
fname.Close  
set fname=nothing  
set fs=nothing  
%>
```

The following code is to check if a file exists.

```
<%  
Set fs=Server.CreateObject("Scripting.FileSystemObject")  
If (fs.FileExists(server.mappath("/test.txt")))=true Then  
    Response.Write("File exists.")  
Else  
    Response.Write("File does not exist.")  
End If  
set fs=nothing  
%>
```

The following code is to get a files extension.

```
<%  
Set fs=Server.CreateObject("Scripting.FileSystemObject")  
Response.Write("The file extension of the file is: ")  
Response.Write(fs.GetExtensionName(server.mappath("/test.txt")))  
set fs=nothing  
%>
```

The following code is to read from a file.

```
<%  
' create the fso object  
set fso = Server.Createobject("Scripting.FileSystemObject")  
path = server.mappath("/test.txt")  
' open the file  
set file = fso.opentextfile(path, 1) <-- For reading  
do until file.AtEndOfStream  
Response.write("Name: " & file.ReadLine & " ")  
Response.write("Home Page: " & file.ReadLine & " ")  
Response.write("Email: " & file.ReadLine & "<p>")  
loop  
' close and clean up  
file.close  
set file = nothing  
set fso = nothing  
>%
```

---

## 15.7. Session and Cookies

---

You can store information in Session object only if the browser supports Cookies or if the support for cookies has not been turned off. When a browser is started, a unique Session ID is created and stored in the browser as a session cookie. This session ID is submitted automatically to the web server on each request. The session values stored in the session object are unique for each user and cannot be shared between different users (clients). If sharing of information is needed between different users, then use the Application object to create truly global variables. The application level variables should be protected by Lock and Unlock methods when modifications are needed.

### ASP Session

The Session Object in ASP is a great tool for the modern web site. It allows you to keep information specific to each of your site's visitors. Information like username, shopping cart, and location can be stored for the life

of the session so you don't have to worry about passing information page to page.

In old web page designs you might have to try to pass information this information through HTML Forms or other methods.

### **ASP Session Object**

Contained within the Session Object are several important features that we will talk about in this lesson. The most important thing to know about ASP's Session Object is that it is only created when you store information into the Session Contents collection. We will now look into creating and storing information in an ASP Session.

### **ASP Session Variables**

To store a Session Variable you must put it into the Contents collection, which is very easy to do. Here we are saving the Time when someone visited this page into the Session Contents collection and then displaying it .

ASP Code:

```
<%  
'Start the session and store information  
Session("TimeVisited") = Time()  
Response.Write("You visited this site at: " & Session("TimeVisited"))  
%>
```

Display:

You visited this site at: 11:35:30 AM

Here we are creating two things actually: a key and a value. Above we created the key "TimeVisited" which we assigned the value returned by the Time() function. Whenever you create a Session Variable to be stored in the Session Contents collection you will need to make this Key / Value pair.

### **ASP Session ID**

The ASP Session ID is the unique identifier that is automatically created when a Session starts for a given visitor. The Session ID is a property of the Session Object and is rightly called the SessionID property. Below we store the user's SessionID into a variable.

ASP Code:

```
<%  
Dim mySessionID  
mySessionID = Session.SessionID  
%>
```

### **ASP Session Timeout**

A Session will not last forever, so eventually the data stored within the Session will be lost. There are many reasons for a Session being destroyed. The user could close their browser or they could leave their computer for an extended amount of time and the Session would time out. You can set how long it takes, in minutes, for a session to time out with the Timeout property.

Below we set our session to timeout after 240 minutes, which should be more than enough time for most web sites.

ASP Code:

```
<%  
Session.Timeout = 240  
Response.Write("The timeout is: " & Session.Timeout)  
%>
```

Display:

The timeout is: 240

Note: Timeout is defined in terms of minutes

### **ASP Cookies**

Like ASP Sessions, ASP Cookies are used to store information specific to a visitor of your website. This cookie is stored to the user's computer for an extended amount of time. If you set the expiration date of the cookie for some day in the future it will remain their until that day unless manually deleted by the user.

If you have read through the Sessions lesson you will notice that ASP Cookies code has several similarities with ASP Sessions.

## ASP Create Cookies

Creating an ASP cookie is exactly the same process as creating an ASP Session. Once again, you must create a key/value pair where the key will be the name of our "created cookie". The created cookie will store the value which contains the actual data.

In this example we will create a cookie named brownies that stores how many brownies we ate during the day.

ASP Code:

```
<%  
  
'create the cookie  
  
Response.Cookies("brownies") = 13  
  
%>
```

## ASP Retrieving Cookies

To get the information we have stored in the cookie we must use the ASP Request Object that provides a nice method for retrieving cookies we have stored on the user's computer. Below we retrieve our cookie and print out its value.

ASP Code:

```
<%  
  
Dim myBrownie  
  
'get the cookie  
  
myBrownie = Request.Cookies("brownies")  
  
Response.Write("You ate " & myBrownie & " brownies")  
  
%>
```

Display:

You ate 13 brownies

Note: Be sure you see that when you create a cookie you use Response.Cookies, but when you retrieve a cookie you use Request.Cookies.

## ASP Cookie Expiration Date

Unlike real life cookies, in ASP you can set how long you want your cookies to stay fresh and reside on the user's computer. A cookie's expiration can hold a date; this date will specify when the cookie will be destroyed.

In our example below we create a cookie that will be good for 10 days by first taking the current date then adding 10 to it.

ASP Code:

```
<%  
  
'create a 10-day cookie  
  
Response.Cookies("brownies") = 13  
  
Response.Cookies("brownies").Expires = Date() + 10  
  
'create a static date cookie  
  
Response.Cookies("name") = "Suzy Q."  
  
Response.Cookies("name").Expires = #January 1,2009#  
  
%>
```

## ASP Cookie Arrays or Collections

Up until now we have only been able to store one variable into a cookie, which is quite limiting if you wanted to store a bunch of information. However, if we make this one variable into a collection it can store a great deal more. Below we make a brownies collection that stores all sorts of information.

ASP Code:

```
<%  
  
'create a big cookie  
  
Response.Cookies("brownies")("numberEaten") = 13  
  
Response.Cookies("brownies")("eater") = "George"  
  
Response.Cookies("brownies")("weight") = 400  
  
%>
```

## ASP Retrieving Cookie Values From a Collection

Now to iterate through the brownies collection we will use a for each loop. See our for loop tutorial for more information.

ASP Code:

```
<%  
For Each key In Request.Cookies("Brownies")  
    Response.Write("<br />" & key & " = " & _  
    Request.Cookies("Brownies")(key))  
Next  
Response.Cookies("brownies")("numberEaten") = 13  
Response.Cookies("brownies")("eater") = "George"  
Response.Cookies("brownies")("weight") = 400  
%>
```

Display:

```
numberEaten = 13  
eater = George  
weight = 400
```

---

## 15.8. Let us Sum Up

---

Active Server Pages are browser independent. The browser only sees pure HTML pages; no vendor proprietary programs or extensions are needed for customers to use ASP applications. ASP is easy to learn. ASP hides the code from the customer (and the hacker). ASP gives an efficient link to the many databases that comply with the Open Database Connection (ODBC) standard.

The Internet Information Server (IIS) versions 3.0, IIS version 4.0 and the Personal Web Server (PWS) can process active server pages. The server knows that ASP code is in the file from the ASP extension. ASP is included in Microsoft's Internet Information Server (IIS). ASP runs as an add-on dll to Microsoft's Personal Web Server.

ASP Delimiters are much like HTML:

```
<% Code %>
```

By default the code used to write ASPs is VBScript but Javascript, PERL or C++ can be used. The default language is set in the server's NT operating system registry by the server management software.



## **Client Side Scripting VS Server Side Scripting**

One can add smarts to web pages using either client side or server side processes. The client side is the customer's browser such as Netscape Navigator or Microsoft Internet Explorer. The server side processes are performed on the hypertext or web server - be it a personal computer or a mainframe computer. [Common Gateway Interface](#) (CGI) and perl scripts are server side processes. The variables are collected from the web page and passed via the CGI to the server where it is processed by [perl scripts](#). [Javascript](#) (not to be confused with Java) and VBScript are computer languages that typically execute on the client side (i.e. in the browser). Unlike CGI, these scripts are on the same page as HTML. A block of ASP code, some HTML, some more code, etc.

## **Configuring your Web Server for Active Server Pages**

### **Personal Web Server**

Installation:

To Install PWS on Windows 98:

1. insert your Windows 98 operating system CD into your CD-ROM drive.
2. Click on the Start button and choose Run.
3. In the Run Dialog box type X:\add-ons\pws\setup.exe (where X is the drive letter of your CD-ROM drive) and click OK.
4. Follow the instructions on the screen to install PWS.

### **Internet Information Server**

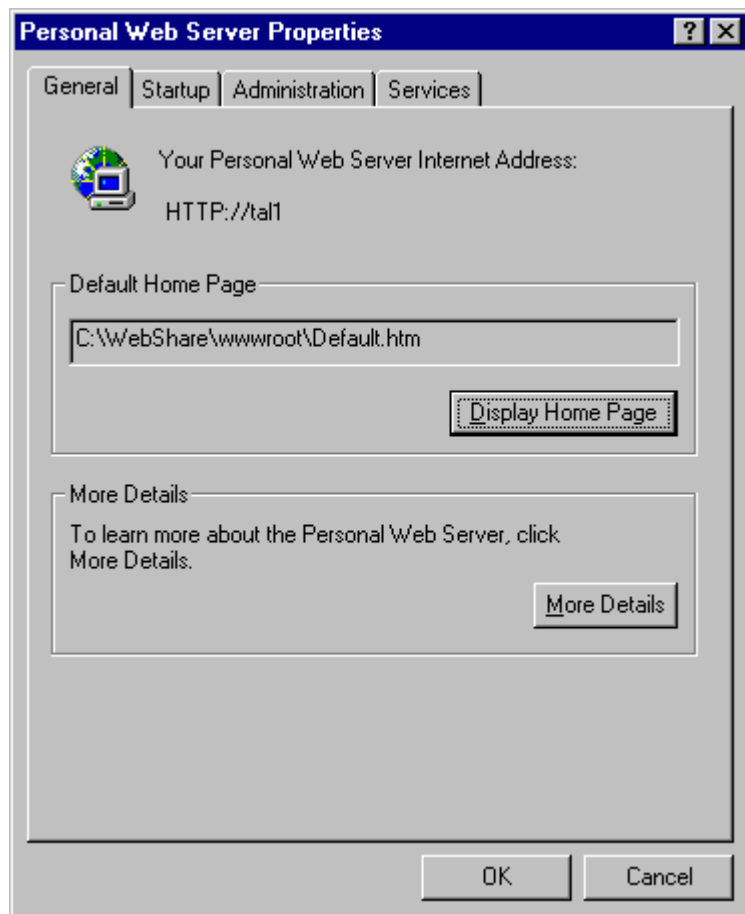
Microsoft Internet Information Server (IIS) is a web server that integrates into the Windows NT Server. IIS allows you to publish information on the World Wide Web and to run multiple business applications using ASP.

Installation

Make sure you have Service Pack 3 (or higher) and Microsoft Internet Explorer 4.01 (or higher - recommended) installed on your Windows NT Server before you install the Windows NT 4.0 Option Pack. It is recommended that you apply Service Pack 4 for Windows NT after installing IIS. Note: You must reapply Service Pack 4 to your computer when you install a new component of the Windows NT operating system or the Windows NT 4.0 Option Pack.

## Getting Started

Before jumping straight into Server Side Scripting for the TALtech ActiveX Plus, we recommend testing your web server to make sure that it is correctly configured to work with Active Server Pages (ASP). Some older versions of PWS, such as the one installed with Microsoft FrontPage 98 (AKA "FrontPage Web Server") do not install support for ASP by default. If your Personal Web Server looks like this:



Then you may wish to upgrade it. Below is a simple ASP file that can be used to test the server.

Copy and Paste the code below into Notepad. Save the file as Test1.asp into the home directory of the default Web site that was installed with PWS or IIS (Usually C:\Webshare\wwwroot or C:\InetPub\wwwroot by default).

```
<HTML><BODY>
Output:<BR>
<%
intS = 60*60
%>
There are
<%
Response.Write intS
%>
seconds in an hour.
</BODY></HTML>
```

You can access the ASP page Test1.asp by typing the following URL in your Web Browser: <http://localhost/Test1.asp>

You should see:

Output:

There are 3600 seconds in an hour.

### **Active Data Objects**

Objects are the building blocks of Java, Javascript, Visual Basic and other object oriented languages. Objects have:

- Properties or attributes - are the characteristics of the object
- Methods - are the tricks the object can do
- Events - are when the object does its tricks

The server uses the Active Data Object (ADO), an Active-X object that handles all the data from server to browser. The ADO has various drivers for different databases and can work with any ODBC database.

There are six built-in ASP objects that simplify web development. These are:

Application

Session

ObjectContext

Request

Response

### **Server**

ASP has several built in Objects. The five Objects we will learn about here are:

1. **Application objects** - for managing information for a web application created with ASP. Applications objects are for the application as a whole and start when the web server starts. Application objects are defined in the [global.asa](#) file.
2. **Session objects** - for managing information concerning the user's current Web session. Session objects last for the time a user is on the site. Session objects are created when individuals enter the application and continue until the timeout occurs. Session objects can be defined in any file including the [global.asa](#) file
3. **Response objects** - for sending information to the client or user
4. **Request objects** - for receiving information from the user
5. **Server objects** - for providing information about the server.

### **Accessing Files on the Server**

ASP provides an ActiveX component called “Scripting.FileSystemObject” for accessing files, folders and drives on the server.

Scripting.FileSystemObject provides a few important methods to create files, open existing files for reading or writing e.g., CreateTextFile and OpenTextFile methods. These methods end up creating an object called TextStream object which has several methods for reading or writing data to a file e.g., Read, ReadLine, ReadAll, Write, WriteLn, Close.

TextStream object provides two important properties to determine the end of line or end of file when reading data from a file (AtEndOfLine, AtEndOfStream).

The following table shows the format of the file object.

FileSystemObject.OpenTextFile(fname,mode,create,format)

Parameter	Description
fname	Required. The name of the file to open
mode	Optional. How to open the file 1=ForReading - Open a file for reading. You cannot write to this file. 2=ForWriting - Open a file for writing. 8=ForAppending - Open a file and write to the end of the file.
create	Optional. Sets whether a new file can be created if the filename does not exist. True indicates that a new file can be created, and False indicates that a new file will not be created. False is default
Format	Optional. The format of the file 0=TristateFalse - Open the file as ASCII. This is default. -1=TristateTrue - Open the file as Unicode. -2=TristateUseDefault - Open the file using the system default.

## Session and Cookies

You can store information in Session object only if the browser supports Cookies or if the support for cookies has not been turned off. When a browser is started, a unique Session ID is created and stored in the browser as a session cookie. This session ID is submitted automatically to the web server on each request. The session values stored in the session object are unique for each user and cannot be shared between different users (clients). If sharing of information is needed between different users, then use the Application object to create truly global variables. The application level variables should be protected by Lock and Unlock methods when modifications are needed.

## ASP Session

The Session Object in ASP is a great tool for the modern web site. It allows you to keep information specific to each of your site's visitors. Information like username, shopping cart, and location can be stored for the life of the session so you don't have to worry about passing information page to page.

In old web page designs you might have to try to pass information this information through HTML Forms or other methods.

### **ASP Session Object**

Contained within the Session Object are several important features that we will talk about in this lesson. The most important thing to know about ASP's Session Object is that it is only created when you store information into the Session Contents collection. We will now look into creating and storing information in an ASP Session.

### **ASP Session Variables**

To store a Session Variable you must put it into the Contents collection, which is very easy to do. Here we are saving the Time when someone visited this page into the Session Contents collection and then displaying it .

### **ASP Session ID**

The ASP Session ID is the unique identifier that is automatically created when a Session starts for a given visitor. The Session ID is a property of the Session Object and is rightly called the SessionID property. Below we store the user's SessionID into a variable.

### **ASP Session Timeout**

A Session will not last forever, so eventually the data stored within the Session will be lost. There are many reasons for a Session being destroyed. The user could close their browser or they could leave their computer for an extended amount of time and the Session would time out. You can set how long it takes, in minutes, for a session to time out with the Timeout property.

### **ASP Cookies**

Like ASP Sessions, ASP Cookies are used to store information specific to a visitor of your website. This cookie is stored to the user's computer for an extended amount of time. If you set the expiration date of the cookie for some day in the future it will remain their until that day unless manually deleted by the user.

If you have read through the Sessions lesson you will notice that ASP Cookies code has several similarities with ASP Sessions.

### **ASP Create Cookies**

Creating an ASP cookie is exactly the same process as creating an ASP Session. Once again, you must create a key/value pair where the key will be the name of our "created cookie". The created cookie will store the value which contains the actual data.

### **ASP Retrieving Cookies**

To get the information we have stored in the cookie we must use the ASP Request Object that provides a nice method for retrieving cookies we have stored on the user's computer. Below we retrieve our cookie and print out its value.

### **ASP Cookie Expiration Date**

Unlike real life cookies, in ASP you can set how long you want your cookies to stay fresh and reside on the user's computer. A cookie's expiration can hold a date; this date will specify when the cookie will be destroyed.

In our example below we create a cookie that will be good for 10 days by first taking the current date then adding 10 to it.

### **ASP Cookie Arrays or Collections**

Up until now we have only been able to store one variable into a cookie, which is quite limiting if you wanted to store a bunch of information. However, if we make this one variable into a collection it can store a great deal more. Below we make a brownies collection that stores all sorts of information.

### **ASP Retrieving Cookie Values From a Collection**

Now to iterate through the brownies collection we will use a for each loop. See our for loop tutorial for more information.

---

### **15.9. Lesson end Activities**

---

1. What is the different between client side scripting and server side scripting?
2. What is the purpose of session and cookies?

---

### **15.10. Check your progress**

---

1. Write a ASP program to find out the number of persons visited your web page.
2. Write a ASP program to copy a text file into different directory.

---

### **15.11. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.microsoft.com](http://www.microsoft.com)



## ASP – using ODBC

---

### Contents

#### 16.0. Aim and Objective

#### 16.1. Introduction

#### 16.2. ASP and the Open Database Connection

#### 16.3. Typical SQL Commands

#### 16.4. Active Data Objects

#### 16.5. Transactions: Adding, deleting and editing records

#### 16.6. Searching in the Database

#### 16.7. Let us Sum Up

#### 16.8. Lesson end Activities

#### 16.9. Check your Progress

#### 16.10. Reference

---

### 16.0. Aim and Objectives

---

- Connect to databases using the ODBC Data Source Administrator.
- Use the ADO Active-X object to access databases with Active Server Pages.
- Apply basic SQL and VBScript commands to build datasets to use in web pages.
- View, Add, Delete, Edit and Search Databases with Active Server Pages.

---

### 16.1. Introduction

---

ASP provide a convenient means to View, Add, Delete, Edit and Search Databases. Every database has these basic functions. To learn about how to use Active Server Pages to support a database application on the Web, we will develop a simple Access database of contact names and address. Most

databases are far more complex than this example, but the fundamental principles are the same.

---

## **16.2. ASP and the Open Database Connection**

---

ADO talks to the database using Structured Query Language (SQL), a standard language for communicating with relational (tabular) databases. To see web pages as they are processed, put the server's IP address and the directory and file name into the location window on the browser.

### **Setting Up an ODBC Connection using the System DSN**

Before we talk about SQL, let's tell ADO about our ODBC connection between the server and the database.

1. In Settings/Control Panel go to 32bit ODBC icon
2. Select the SYSTEM DSN tab and hit ADD
3. Select Microsoft Access Driver from the list and select Finish
4. The Data Source Name is Contacts
5. Select a database by hitting select and browsing to the database. Hit OK. The database should not be in the WEBROOT unless customers need to be able to download the database. The database can be on another machine as well. Note: the database cannot be open when setting this up. If the database is sensitive, put it on a machine only reachable by a non-TCP/IP networking protocol.
6. Hit OK and the connection is made.

### **Setting Up an ODBC Connection using the File DSN**

You can also make an ODBC connection with a file.dsn file and the appropriate commands in the ASP pages. The advantage is that one need not have access to the web server console to create the ODBC connection. Secondly, one can readily change the database location by simply changing the code. One can use the console ODBC program to create file.dsn files for a variety of ODBC compatible database. This example is for Microsoft Access database.

---

### 16.3. Typical SQL Commands

---

The SQL commands that we will be using are:

#### **SELECT**

SELECT tells ADO what fields to retrieve from which table. For example:

```
SELECT * FROM TableName
```

#### **WHERE**

WHERE limits what data is selected. An asterix means all elements (columns) in the database.

```
SELECT * FROM TableName WHERE FieldName = 'Value'
```

#### **ORDER BY**

ORDER BY sorts what is returned.

```
SELECT * FROM TableName WHERE FieldName > 50 or FieldName < 100 ORDER BY FieldName2, FieldName3
```

#### **INSERT INTO**

INSERT INTO adds a new record to the table.

```
INSERT INTO TableName (FieldName1, FieldName2) VALUES ('Value1', 'Value2')
```

#### **DELETE FROM**

DELETE FROM deletes records from the table.

```
DELETE FROM TableName WHERE FieldName = 'Value'
```

#### **UPDATE**

UPDATE changes the values of particular fields of the table

```
UPDATE TableName SET FieldName = 'New Value' WHERE FieldName2 = 'Value2'
```

```
UPDATE ComicCollection SET StreetValue = StreetValue + 10 WHERE Title = 'X-men'
```

#### **LIKE**

LIKE used to search through records

```
SELECT * FROM TableName Where FieldName LIKE '%%Value%%'
```

### ***Wild Card Characters!***

The % (percent sign) indicates there can be characters before or after the value.

---

## **16.4. Active Data Objects**

---

Thanks to the ODBC connection, the web server knows the database is there. Now we to get the data and do something useful with it. We employ the Active Data Objects (ADO) and SQL together to create a record set drawn from the database. The record set is then used to create the HTML page. A record set is like a temporary table that stores what is requested through the SQL command.

We will use Active Data Objects or ADO to create data objects the server can use. We use ADO to identify the database and the ODBC connection to use and what SQL command to use. We create the object, establish a connection to a database, define a SQL command and then execute the SQL command. By starting with the SQL Command first; the rest of the page will follow from it.

---

## **16.5. Transactions : Adding, Deleting and Editing Records**

---

The links for searching, adding, deleting and editing pages should be located at the bottom of the page that displays individual record information. Each of these processes is going to have a separate page. They will all be Active Sever Pages so we will name them add.html, search.html, edit.asp, and delete.asp. The URL to links to add.html and search.html are nothing unexpected. With edit and delete though there is one record that we want to delete or edit. One way to do it is to put the links on the bottom of the page labeled "Update this Contact" and "Delete this Contact". By doing this we can use the current record's ContactID to let the ASPs know which record to affect. The code at the foot of viewlist.asp would be:

```
<A href="add.html">Add a Contact</a><br>
<a href="search.html">Search Contacts</a><br>
<A href="edit.asp?ContactID=<%=rs("ContactID")%>">Update this
Contact</a><br>
```

```
<A href="delete.asp?ContactID=<%=rs("ContactID")%>">Delete this  
Contact</a><br>
```

## **Adding Records**

We will create the Add page with HTML forms to enter information through the web. First create a form tag in the body. The data is submitted to another page called addsubmit.asp with a method of Post. Next determine what fields to submit, in this case the text fields: Name, Address, City, State, Zip Code. Note we include hidden forms to capture information such as date submitted. We end the form with the submit button.

```
<body>  
<form action="addsubmit.asp" method="POST">  
Name: <input type="Text" name="txtName" size="20"><br>  
Address: <input type="Text" name="txtAddress" size="20"><br>  
City: <input type="Text" name="txtCity" size="20"><br>  
State: <input type="Text" name="txtState" size="20"><br>  
Zip Code: <input type="Text" name="txtZip" size="20"><br>  
<input type="Submit" name="" value="Add">  
</form>  
</body>
```

Note that there is no Javascript code in this file so save add.html with a file extension of .html instead of .asp. If we include a hidden field to capture the Date (<input type="hidden" name="txtDate" value="<%= now() %>">) then this would have to be an asp file. Do this as a class exercise.

## **Inserting the New Record into the Database**

Now, we create the page addsubmit.asp that puts the data we collected into a Database. First we define the SQL command. It doesn't really matter where the SQL command is defined as long as it is before the ADO tries to use the SQL command. However, the SQL statement is the key to the page so we suggest beginning with it. The SQL command we will use is the INSERT. It has the format:

INSERT INTO TableName (FieldName1, FieldName2) VALUES ('Value1', 'Value2') Specifically for this example:

INSERT INTO Contacts (Name, Address, City, State, Zip) VALUES ('Value1', 'Value2', 'Value3', 'Value4', 'Value5')

### Collecting the Data

We will use the request.form method to get the Values into our SQL command. The SQL command must be on one continuous string or line. Therefore, we will use the VB Concatenation string &\_

```
<%  
SQL="INSERT INTO Contacts (Name, Address, City, State, Zip) Values ("  
&  
"" & request.form("txtName") + ", " &  
"" & request.form("txtAddress") + ", " &  
"" & request.form("txtCity") + ", " &  
"" & request.form("txtState") + ", " &  
"" & request.form("txtZip") + ")"  
%>
```

The SQL command in addsubmit.asp is the same as the one created before, but now it is broken up with **request.form** to accept different values.

### Updating the Data

The Next three lines in addsubmit.asp are identical to those used in the previous example. An ADO created Object **DbConn.execute(SQL)** executes the SQL command.

```
<%  
SQL="INSERT INTO Contacts (Name, Address, City, State, Zip) Values ("  
&  
"" & request.form("txtName") + ", " &  
"" & request.form("txtAddress") + ", " &  
"" & request.form("txtCity") + ", " &  
"" & request.form("txtState") + ", " &  
"" & request.form("txtZip") + ")"  
DbConn = Server.CreateObject("ADODB.Connection")
```

```

DSName = Session("DSName")
DbConn.Open("DSName;")
RS = DbConn.execute(SQL)
%>

```

### Confirming the Success of the Transaction

If the code is correct, the data will be entered in the database. Next we let the user know that the update is done. One way to do this is to add a Success page after the ASP code. Once the ASP has executed, the HTML page will be shown. Another option is addsubmit.asp to end with a **response.redirect** tag to send users to another page. This would be done as so:

```

<%
SQL="INSERT INTO Contacts (Name, Address, City, State, Zip) Values ("
&_
"" & request.form("txtName") + ", " &_
"" & request.form("txtAddress") + ", " &_
"" & request.form("txtCity") + ", " &_
"" & request.form("txtState") + ", " &_
"" & request.form("txtZip") + ")"
DbConn = Server.CreateObject("ADODB.Connection")
DSName = Session("DSName")
DbConn.Open("DSName;")
RS = DbConn.execute(SQL)
response.redirect "list.asp"
%>

```

If you use the **response.redirect** you will need to set buffering to **true** before executing a **response.write** tag. You do this by placing the command `<% Response.Buffer = True %>` at the beginning of the ASP page (just after the ASP command `<%@ language=VBScript %>`). The **response.write** writes a partial HTML page while the **response.redirect** tries to move to a new page, causing an error if buffering is set to false. For IIS 4.0 and earlier, buffering defaults to FALSE while IIS 5.0 defaults buffering to true.

## Editing a Record

Editing a record combines the previous list processes. First, in edit.asp we display the information that has been entered (like in the first example). Then, we want to change the data with forms (like the second example). So first we write the ASP code edit.asp to retrieve the data. This is the same code used in list.asp for the first example.

```
<%  
DbConn = Server.CreateObject("ADODB.Connection")  
DSName = Session("DSName")  
DbConn.Open("DSName;")  
SQL = "SELECT * FROM Contacts Where ContactID=" +  
request("ContactID")  
RS = DbConn.execute(SQL)  
%>
```

Now we have a record set for the one record that we want to edit.

## Displaying the Record to Be Edited

Next we put the data into a form so that it can be changed. We copy the form from the second example but make a few changes. The form action points to a new asp file called editsubmit.asp. We also pass the ContactID so that the server knows which file to change. Also, we assign values to all the form fields that are the values of the corresponding Field in the Contacts Table.

```
<body>  
<form action="editsubmit.asp?ContactID=<%RS("ContactID")%>"  
method="POST">  
Name: <input type="Text" name="txtName"size="20" value="<%=  
rs("Name") %>">  
<br>  
Address: <input type="Text" name="txtAddress" size="20" value="<%=  
rs("Address") %>">>  
<br>  
City: <input type="Text" name="txtCity" size="20" value="<%= rs("City")  
%>">> <br>
```



```

State: <input type="Text" name="txtState" size="20" value="<%=
rs("State") %>">>
<br>
Zip Code: <input type="Text" name="txtZip" size="20" value="<%=
rs("Zip") %>">> <br>
<input type="Submit" name="" value="Add">
</form>
</body>

```

## Updating the Record

Now we will use the data we collected in the form. In editsubmit.asp we write over the each field, regardless if that field was changed, with the UPDATE SQL Command. The general format of the UPDATE command is:

UPDATE TableName SET FieldName = 'New Value' WHERE FieldName2 = 'Value2' For this exercise, the SQL Command is:

UPDATE Contacts Set Name = 'Value1', Address= 'Value2', City= 'Value3', State= 'Value4', Zip= 'Value5' WHERE ContactID=Value6

Value6 is not enclosed in single quotes because it is an integer, not a string.

## Replacing the Data

Once again we use in editsubmit.asp the request.form method to replace the data with the information submitted via form. This must be on one line, but for readability's sake we use concatenation. The ASP code is:

```

<%
SQL = "UPDATE Contacts SET " &_
"Name=" & request.form("txtName") & ", " &_
"Address=" & request.form("txtAddress") & ", " &_
"City=" & request.form("txtCity") & ", " &_
"State=" & request.form("txtState") & ", " &_
"Zip=" & request.form("txtZip") & " " &_
"WHERE ContactID=" & request("ContactID")
DbConn = Server.CreateObject("ADODB.Connection")
DSName = Session("DSName")
DbConn.Open("DSName;")

```

```
RS = DbConn.execute(SQL)
response.redirect "viewlist.asp?ContactID=" & request("ContactID")
%>
```

## Deleting Records

In delete.asp we delete records using the DELETE SQL command. First, we write out the SQL command. If we want to delete a particular record, use the WHERE command to specify which record. By using a record's unique ContactID we ensure that we delete only that record.

```
DELETE FROM Contacts WHERE ContactID = 'Value'
```

In this example:

```
<%
SQL = "DELETE FROM Contacts Where ContactID=" &
request("ContactID")
DbConn = Server.CreateObject("ADODB.Connection")
DSName = Session("DSName")
DbConn.Open("DSName;")
RS = DbConn.execute(SQL)
response.redirect "list.asp"
%>
```

The rest of the code in delete.asp is the same before, though we redirect this webpage to list.asp after execution.

Note that delete.asp deletes a record as soon as it executes. For student exercise, write a confirmation page to ask the user if she actually wants to go through with the delete.

---

## 16.6. Searching the Database

---

Searching the database is done with the LIKE SQL Command. Before we write the ASP to update the database, we need to identify the search criteria in a form. The form in search.html selects the field or fields to search and what string or strings to search for. In this example, we use a drop down form to determine what to search for and a textbox so users can input a search string.

The values of the Drop Down form are the Names of the Fields in the Database. In our example, the search page search.html will look something like this:

```
<form action="searchsubmit.asp" method="POST">
<select name="txtField">
<option value="Name">Name
<option value="Address">Address
<option value="City">City
<option value="State">State
<option value="Zip">Zip
</select>
<input type="Text" name="txtSearch" size="20"><p>
<input type="Submit" name="" value="Search">
</form>
```

### **Making the Search Form Work**

Once the search form is written, we write the ASP code that makes it work. The values are passed to the file searchsubmit.asp. First, we figure out what SQL Command to use. In this case we use the LIKE Command. LIKE is similar to WHERE but allows greater flexibility.

```
SELECT * FROM Contacts Where FieldName LIKE '%%Value%%'
```

In searchsubmit.asp we want the FieldName to be **request.form("txtField")** i.e. whatever is selected from the drop down box and the Value **request.form("txtSearch")** to be whatever is entered in the text box. The other three lines in searchsubmit.asp are like the same as the first example.

```
<%
SQL = "SELECT * FROM Contacts Where " &
request.form("txtField") & " LIKE '%" & request.form("txtSearch")
& "%'"
DbConn = Server.CreateObject("ADODB.Connection")
DSName = Session("DSName")
DbConn.Open(DSName;")
```

```
RS = DbConn.execute(SQL)
%>
```

## Displaying the Search Results

We have a record set ready to be output to HTML but we are not sure how many records are in the set. Therefore, we loop through the record set in searchsubmit.asp just like we did before, with list.asp. As before, we will link to viewlist.asp to display the detailed record:

```
<%
Do While NOT RS.EOF
%>
<li> <a href="viewlist.asp?ContactID=<%=RS("ContactID")%>">
<%=RS("Name")%></a><br>
<%
RS.MoveNext
loop
%>
```

This will output the Name of the Person **ContactID** who matches the search value for along with a link to that person's entire record. To show the field that was searched for, in searchsubmit.asp we add a few if then statements.

```
<%
Do While NOT RS.EOF
%>
<li> <a href="viewlist.asp?ContactID=<%=RS("ContactID")%>">
<%=RS("Name")%></a><br>
<% If request.form("txtField")="Address" then response.write
RS("Address") & "<br>"%>
<% If request.form("txtField")="City" or
request.form("txtField")="State" then %>
<% response.write RS("City")%>, <% response.write
RS("State")%><br>
<% end if %>
<% If request.form("txtField")="Zip" then response.write RS("Zip") &
```

```
"<br>"%>
<hr width="200" align="left"><br>
<%
RS.Movenext
loop
%>
```

If we try to submit a word with a single quote in it, the ASP will not work. One work around is to change all single quotes to another character.

Creating a system dsn using the ODBC Connection in Control Panel requires console and administrator access. Instead, create a file dsn and refer to the database and the file dsn with session variables.

IIS 4.0 looks for the global.asa in the root directory first, then in the directory where the application resides. If an application breaks, check the root directory for an incompatible global.asa file.

If you use the **response.redirect** you will need to set buffering to **true** before executing a **response.write** tag. You do this by placing the command `<% Response.Buffer = True %>` at the beginning of the ASP page (just after the ASP command `<%@ language=VBScript %>`). The **response.write** writes a partial HTML page while the **response.redirect** tries to move to a new page, causing an error if buffering is set to false. For IIS 4.0 and earlier, buffering defaults to FALSE while IIS 5.0 defaults buffering to true.

### Listing the Data

Look at list.asp to see the file being created. At the very top of ASP file (before the HTML tag), we create a server object:

```
<%
DbConn = Server.CreateObject("ADODB.Connection")
%>
```

Next we tell the Object DbConn which database to use. The ODBC connection must be created beforehand.

```
<%
DbConn = Server.CreateObject("ADODB.Connection")
```

```
DbConn.Open ("DSN=Contacts;")
```

```
%>
```

Write the SQL Command to be used:

```
<%
```

```
DbConn = Server.CreateObject("ADODB.Connection")
```

```
DbConn.Open ("DSN=Contacts;")
```

```
SQL = "SELECT * FROM Contacts ORDER BY Name"
```

```
%>
```

Finally, tell the Object DbConn to use that SQL command and create a Record Set:

```
<%
```

```
DbConn = Server.CreateObject("ADODB.Connection")
```

```
DSName = Session("DSName")
```

```
DbConn.Open("DSName;")
```

```
SQL = "SELECT * FROM Contacts ORDER BY Name"
```

```
RS = DbConn.execute(SQL)
```

```
%>
```

## Displaying the Data

Now the Record Set is created, we display the data with a loop in the body of the ASP file list.asp. The Do While loop moves record by record through the Record Set.

```
<body>
```

```
List of Stuff in the Database <p>
```

```
<ul>
```

```
<% Do While NOT RS.EOF %>
```

```
<li> Name <br>
```

```
<% RS.movenext
```

```
Loop %>
```

```
</body>
```

Use RS("fieldname") to access field data for each Record in the Record Set and write it to the HTML page:

```

<body>
Lets list more stuff from the database and include links <p>
<ul>
<% Do While NOT RS.EOF %>
<li><a href="viewlist.asp?ContactID=<%= rs
("ContactID")%>"><%= rs("Name")%></a><br>
<% RS.movenext
Loop %>
</ul>
</body>

```

### Retrieving a Particular Record

Next we want to create in list.asp a list of all the records in our table with links for each record. These links will invoke another ASP file viewlist.asp that will display the detailed information about each record. The linked list does basically the same task as our previous example, so we will use similar code.

The table and database are the same so the first list lines and last line do not change. However, we need to limit the record set to just one record in the table. Therefore, our SQL command will change.

Select only the fields for this one record from the Contacts table. One distinguishes between records by ContactID. (This is the database's key or index field.) We want to select all the fields from the table Contacts where the value of the field ContactID is the same as the value of the variable ContactID passed along with the URL. So the SQL command for the first record would be "SELECT \* FROM Contacts Where ContactID=1 ". (Do not use single quotes because ContactID is an integer, not a string). ContactID changes when we select a different record.

Instead of writing a new page for each record, we can write a single ASP page viewlist.asp that can be used for any of the records in this database.

```

<%
DbConn = Server.CreateObject("ADODB.Connection")
DSName = Session("DSName")
DbConn.Open("DSName;")

```

```

SQL = "SELECT * FROM Contacts where ContactID=" &
request("ContactID")
RS = DbConn.execute(SQL)
%>

```

When this file is called, the key field ContactID is passed along in the URL in the browser's location box. This requests the value of the variable ContactID. The record set is made of information from the record where the variable ContactID equals the contents of the database field ContactID.

Now we want to display the record set for ContactID with just the information we need. Because we display only one record, viewlist.asp does not need to loop through the record set. Before we write the data, we design the body of the web page viewlist.asp to look like we want it to look.

```

<body>
<font size="+3" face="arial">Name</font><hr> <p>
Name<br>
Address<br>
City, State Zip<p>
<hr><p>
List Contacts<br>
</body>

```

We put ASP code in viewlist.asp wherever we want data to appear:

```

<body>
<font size="+3" face="arial"><%=rs("Name")%></font><hr> <p>
<%=rs("Name")%><br>
<%=rs("Address")%><br>
<%=rs("City")%>, <%=rs("State") & " " & rs("Zip")%><p>
<hr><p>
<A href="list.asp">List Contacts</a><br>
</body>

```

This same file is viewlist.asp used for all the links. Just the SQL command will change, causing a unique Record Set for each ContactID .



---

## 16.7. Let us Sum Up

---

ASP provide a convenient means to View, Add, Delete, Edit and Search Databases. Every database has these basic functions. To learn about how to use Active Server Pages to support a database application on the Web, we will develop a simple Access database of contact names and address. Most databases are far more complex than this example, but the fundamental principles are the same.

ADO talks to the database using Structured Query Language (SQL), a standard language for communicating with relational (tabular) databases. To see web pages as they are processed, put the server's IP address and the directory and file name into the location window on the browser.

### **Setting Up an ODBC Connection using the System DSN**

Before we talk about SQL, let's tell ADO about our ODBC connection between the server and the database.

1. In Settings/Control Panel go to 32bit ODBC icon
2. Select the SYSTEM DSN tab and hit ADD
3. Select Microsoft Access Driver from the list and select Finish
4. The Data Source Name is Contacts
5. Select a database by hitting select and browsing to the database. Hit OK. The database should not be in the WEBROOT unless customers need to be able to download the database. The database can be on another machine as well. Note: the database cannot be open when setting this up. If the database is sensitive, put it on a machine only reachable by a non-TCP/IP networking protocol.
6. Hit OK and the connection is made.

### **Setting Up an ODBC Connection using the File DSN**

You can also make an ODBC connection with a file.dsn file and the appropriate commands in the ASP pages. The advantage is that one need not have access to the web server console to create the ODBC connection. Secondly, one can readily change the database location by simply changing the code. One

can use the console ODBC program to create file.dsn files for a variety of ODBC compatible database. This example is for Microsoft Access database.

### **Typical SQL Commands**

**SELECT**

**WHERE**

**ORDER BY**

**INSERT INTO**

**DELETE FROM**

**UPDATE**

**LIKE**

### **Active Data Objects**

We will use Active Data Objects or ADO to create data objects the server can use. We use ADO to identify the database and the ODBC connection to use and what SQL command to use. We create the object, establish a connection to a database, define a SQL command and then execute the SQL command. By starting with the SQL Command first; the rest of the page will follow from it.

### **Transactions**

**Adding**

**Deleting and**

**Editing**

---

### **16.8. Lesson end Activities**

---

1. What is the purpose of ODBC?
2. Explain about the two methods of ODBC connection
3. Write short notes on Active Data Objects.

---

### **16.9. Check your progress**

---

1. Write a ASP program to maintain the Telephone directory maintenance.
2. Write a ASP program to list the data.

---

**16.10. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.microsoft.com](http://www.microsoft.com)



## CGI and PERL

---

### Contents

- 17.0. Aim and Objectives**
- 17.1. Introduction**
- 17.2. Basics of CGI program**
- 17.3. Perl Variable and Data Types**
- 17.4. String Manipulation**
- 17.5. Regular Expression**
- 17.6. CGI Environment Variables**
- 17.7. Form Processing**
- 17.8. Sending Email**
- 17.9. Verifying user name and password**
- 17.10. Let us Sum Up**
- 17.11. Lesson End Activities**
- 17.12. Check your Progress**
- 17.13. Reference**

---

### 17.0. Aim and Objectives

---

- To understand CGI
- To learn PERL for web page developing

---

### 17.1. Introduction

---

CGI/Perl is very similar to HTML; it has a clearly defined syntax, and if you follow those syntax rules, you can write Perl as easily as you do HTML. The first line of your program should look like this:

```
#!/usr/bin/perl -wT
```

The first part, #!, indicates that this is a script.

The next part, `/usr/bin/perl`, is the location (or *path*) of the Perl interpreter.

The final part contains optional flags for the Perl interpreter.

Warnings are enabled by the `-w` flag.

Your program should follow the above line.

To print a line

```
#!/usr/bin/perl -wT
print("Welcome to Perl");
```

---

## 17.2. Basics of a CGI Program

---

A CGI is simply a program that is called by the webserver, in response to some action by a web user. This might be something simple like a page counter, or a complex form-handler.

CGI programs may be written in *any* programming language; we're just using Perl because it's fairly easy to learn. If you're writing a CGI that's going to generate an HTML page, you must include this statement somewhere in the program before you print out anything else:

```
print "Content-type: text/html\n\n";
```

This is a content-type header that tells the receiving web browser what sort of data it is about to receive — in this case, an HTML document. If you forget to include it, or if you print something else before printing this header, you'll get an "Internal Server Error" when you try to access the CGI program.

### First CGI Program

Create a file with the name "first.cgi". The first line (`#!/usr/bin/perl`) should start in column 1. The subsequent lines can start in any column.

### Example 1: first.cgi - Hello World Program

```
#!/usr/bin/perl -wT
print "Content-type: text/html\n\n";
print "Hello, world!\n";
```

Let's try for second example. If you want to display all html content then add print statement for every HTML tag:

## Example 2: second.cgi - Hello World Program 2

```
#!/usr/bin/perl -wT

print "Content-type: text/html\n\n";

print "<html><head><title>Hello World</title></head>\n";

print "<body>\n";

print "<h2>Hello, world!</h2>\n";

print "</body></html>\n";
```

Save this file, adjust the file permissions if necessary, and view it in your web browser. This time you should see "Hello, world!" displayed in a H2-size HTML header.

Now not only have you learned to write your first CGI program, you've also learned your first Perl statement, the print function:

```
print "somestring";
```

This function will write out any string, variable, or combinations thereof to the current output channel. In the case of your CGI program, the current output is being printed to the visitor's browser.

The `\n` you printed at the end of each string is the *newline* character. Newlines are not required, but they will make your program's output easier to read.

You can write multiple lines of text without using multiple print statements by using the here-document syntax:

```
print <<endmarker;

line1

line2

line3

etc.

endmarker
```

You can use any word or phrase for the end marker. Just be sure that the closing marker matches the opening marker exactly (it is case-sensitive),

and also that the closing marker is on a line by itself, with no spaces before or after the marker.

## The CGI.pm Module

Perl offers a powerful feature to programmers: add-on modules. These are collections of pre-written code that you can use to do all kinds of tasks. You can save yourself the time and trouble of reinventing the wheel by using these modules.

Some modules are included as part of the Perl distribution; these are called *standard library modules* and don't have to be installed. If you have Perl, you already have the standard library modules.

There are also many other modules available that are not part of the standard library. These are typically listed on the Comprehensive Perl Archive Network (CPAN), which you can search on the web at <http://search.cpan.org/>.

The CGI.pm module is part of the standard library, and has been since Perl version 5.004. CGI.pm has a number of useful functions and features for writing CGI programs, and its use is preferred by the Perl community

Let's see how to use a module in your CGI program. First you have to actually include the module via the use command. This goes after the `#!/usr/bin/perl` line and before any other code:

```
use CGI qw(:standard);
```

Note we're not doing `use CGI.pm` but rather `use CGI`. The `.pm` is implied in the use statement. The `qw(:standard)` part of this line indicates that we're importing the "standard" set of functions from CGI.pm.

Now you can call the various module functions by typing the function name followed by any arguments:

```
functionname(arguments)
```

If you aren't passing any arguments to the function, you can omit the parentheses.

A *function* is a piece of code that performs a specific task; it may also be called a *subroutine* or a *method*. Functions may accept optional *arguments* (also



called *parameters*), which are values (strings, numbers, and other variables) passed into the function for it to use.

The CGI.pm module has many functions; for now we'll start by using these three:

```
header;  
start_html;  
end_html;
```

The header function prints out the "Content-type" header. With no arguments, the type is assumed to be "text/html". start\_html prints out the <html>, <head>, <title> and <body> tags. It also accepts optional arguments. If you call start\_html with only a single string argument, it's assumed to be the page title. For example:

```
print start_html("Hello World");
```

will print out the following\*:

```
<html>  
<head>  
<title>Hello World</title>  
<head>  
<body>
```

You can also set the page colors and background image with start\_html:

```
print start_html(-title=>"Hello World",  
  -bgcolor=>"#cccccc", -text=>"#999999",  
  -background=>"bgimage.jpg");
```

Notice that with multiple arguments, you have to specify the name of each argument with -title=>, -bgcolor=>, etc. This example generates the same HTML as above, only the body tag indicates the page colors and background image:

```
<body bgcolor="#cccccc" text="#999999" background="bgimg.jpg">
```

The end\_html function prints out the closing HTML tags:

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

So, as you can see, using CGI.pm in your CGI programs will save you some typing.

Let's try using CGI.pm in an actual program now. Start a new file and enter these lines:

**Example 3: three.cgi - Hello World Program, using CGI.pm**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
print header;
print start_html("Hello World");
print "<h2>Hello, world!</h2>\n";
print end_html;
```

CGI.pm also has a number of functions that serve as HTML shortcuts. For instance:

```
print h2("Hello, world!");
```

Will print an H2-sized header tag. You can find a list of all the CGI.pm functions by typing **perldoc CGI** in the shell, or visiting <http://www.perldoc.com/> and entering "CGI.pm" in the search box.

---

### 17.3. Perl Variable and Data Types

---

Perl has built-in data types that represent different kinds of data. Each variable has a specific character preceding.

Data Type	Format	Description
Scalar	\$scalarname	Can be string, an integer number, a floating-point number or a reference
Array	@arrayname	An ordered list of scalar variables that can be accessed using integer indices
Hash	%hashname	An unordered set of scalar variables whose values are accessed using unique scalar value

## Documenting Your Programs

Documentation can be embedded in a program using comments. A comment in Perl is preceded by the # sign; anything appearing after the # is a comment:

### Example 4: four.cgi - Hello World Program, with Comments

```
#!/usr/bin/perl -wT
use CGI qw(:standard);

# This is a comment
# So is this
#
# Comments are useful for telling the reader
# what's happening. This is important if you
# write code that someone else will have to
# maintain later.

print header;           # here's a comment. print the header
print start_html("Hello World");
print "<h2>Hello, world!</h2>\n";
print end_html; # print the footer
# the end.
```

---

## 17.4. String Manipulation

---

Perl is very well known for ease in manipulation of strings. What might take other programming languages several lines of code can generally be done in Perl in less than 2 lines.

### The dot operator (.)

This allows you to concatenate two strings together into one string.

# Example using the Dot Operator

```
$sFirstName = "Bharathiar ";
```

```
$sLastName = "University";
```

```
$sFullName = $sFirstName.$sLastName;
```

This will take the value of `$sFirstName` and concatenate the value of `$sLastName`. Thus, this will give you the `$sFullName` "Bharathiar University".

### **The index function**

This allows you to find an occurrence of one string within another and return the position of that occurrence.

# Example using the `index()` function

```
$sString = "Computer Science and Engineering";
```

```
$sSubString = "Science";
```

```
$iIndex = index($sString, $sSubString);
```

`$sString` contains the string that you are searching. The `index()` function finds the position in the string where `$sSubString` begins. In the above example, `$iIndex` will get the value 10 since "Science" starts at position ten in the string.

### **The substring function**

This allows you to copy part of a string from another by specifying beginning position and (in one version) length of string.

# Example using the `substr()` function

```
# $sSubString = substr($sString, $iStart);
```

```
$sOldString = "Bharathiar University";
```

```
$sNewString = substr($sOldString, 12,3);
```

`$sNewString` will contain the new value. It looks at string `$sOldString` as an array of characters. It goes to the character at position 12 which would be 'Uni' and then it grabs it and the rest of the 3 character. This new string is now: "University".

### **The length function**

This function returns the length in characters of the string.

# Example using the `length()` function

```
$sString = "JavaCard";
```

```
$iLength = length ($sString);
```

`$iLength` will contain the length of the string `$sString` which is 8 characters long.

### **The split function**

This allows you to split up one string into multiple strings by looking for occurrences of a character or characters within it. The results are stored into an array.

```
# Examples using the split() function

# IP Address 163.185.20.182

$IpAddress = "163.185.20.182";

@asClassAddresses = split (/\. /, $IpAddress);
```

Notice the backslash in the split function. The `/.../` is a place where a regular expression can appear. A `"."` in a regular expression means match any character

Now the array contain

```
$asClassAddresses[0] = "163"
$asClassAddresses[1] = "185"
$asClassAddresses[2] = "20"
$asClassAddresses[3] = "182"
```

### **The join function**

This does the exact opposite of `split()`. Here you give an array of scalars and a delimiting character. Join will glue it all together.

```
# Examples using the join() function

@asIpAddress = ("163","185","20","182");

$sFullIpAddress = join(".", @asIpAddress);
```

Join will take the individual values "163", "185", "20", and "182" and glue them together using a dot. Your final result stored in string `$sFullIpAddress` will be "163.185.20.182".

---

## **17.5. Regular Expressions**

---

The major goal to design Perl is to easy the text processing.

## Simple Characters

In regular expressions, generally, a character matches itself. The only exceptions are regular expression special characters. To match one of these special characters, you must put a `\` before the character.

For example, the regular expression `abc` matches a set of strings that contain `abc` somewhere in them. Since `*` happens to be a regular expression special character, the regular expression `\*` matches any string that contains the `*` character.

### The `*` and `.` Special Character

The `*` is used to indicate that zero or more of the previous characters should be matched. Thus, the regular expression `a*` will match any string that contains zero or more `a`'s. Note that since `a*` will match any string with zero or more `a`'s, `a*` will match all strings, since all strings (including the empty string) contain at least zero `a`'s. So, `a*` is not a very useful regular expression.

A more useful regular expression might be `baa*`. This regular expression will match any string that has a `b`, followed by one or more `a`'s. Thus, the set of strings we are matching are those that contain `ba`, `baa`, `baaa`, etc. In other words, we are looking to see if there is any "sheep speech" hidden in our text.

The next special character we will consider is the `.` character. The `.` will match any valid character. As an example, consider the regular expression `a.c`. This regular expression will match any string that contains an `a` and a `c`, with any possible character in between. Thus, strings that contain `abc`, `acc`, `amc`, etc. are all in the class of strings that this regular expression matches.

### The `|` Character

The `|` special character is equivalent to an "or" in regular expressions. This character is used to give a choice. So, the regular expression `abc|def` will match any string that contains either `abc` or `def`.

### Grouping with `()`s

Sometimes, within regular expressions, we want to group things together. Doing this allows building of larger regular expressions based on smaller components. The `()`'s are used for grouping. For example, if we want to match any string that contains `abc` or `def`, zero or more times, surrounded by a

xx on either side, we could write the regular expression `xx(abc|def)*xx`. This applies the `*` character to everything that is in the parentheses. Thus we can match any strings such as `xxabcxx`, `xxabcdefxx`, etc.

## **The Anchor Characters**

Sometimes, we want to apply the regular expression from a defined point. In other words, we want to anchor the regular expression so it is not permitted to match anywhere in the string, just from a certain point.

The anchor operators allow us to do this. When we start a regular expression with a `^`, it anchors the regular expression to the beginning of the string. This means that whatever the regular expression starts with must be matched at the beginning of the string. For example, `^aa*` will not match strings that contain one or more a's; rather it matches strings that start with one or more a's.

We can also use the `$` at the end of the string to anchor the regular expression at the end of the string. If we applied this to our last regular expression, we have `^aa*$` which now matches only those strings that consist of one or more a's. This makes it clear that the regular expression cannot just look anywhere in the string, rather the regular expression must be able to match the entire string exactly, or it will not match at all.

In most cases, you will want to either anchor a regular expression to the start of the string, the end of the string, or both. Using a regular expression without some sort of anchor can also produce confusing and strange results. However, it is occasionally useful.

## **Pattern Matching**

Now that you are familiar with some of the basics of regular expressions, you probably want to know how to use them in Perl. Doing so is very easy. There is an operator, `=~`, that you can use to match a regular expression against scalar variables. Regular expressions in Perl are placed between two forward slashes (i.e., `//`). The whole `$scalar =~ //` expression will evaluate to 1 if a match occurs, and `undef` if it does not. Consider the following code sample:

```

use strict;

while ( defined($currentLine = <STDIN> ) ) {
    if ($currentLine =~ /^(J|R)MS speaks:/) {
        print $currentLine;
    }
}

```

This code will go through each line of the input, and print only those lines that start with `\JMS speaks:"` or `\RMS speaks:"`.

### Regular Expression Shortcuts

Writing out regular expressions can be problematic. For example, if we want to have a regular expression that matches all digits, we have to write:

```
(0|1|2|3|4|5|6|7|8|9)
```

It would be terribly annoying to have to write such things out. So, Perl gives an incredible number of shortcuts for writing regular expressions.

For example, for ranges of values, we can use the brackets, `[]`'s. So, for our digit expression above, we can write `[0-9]`. In fact, it is even easier in perl, because `\d` will match that very same thing.

---

## 17.6.CGI Environment Variables

---

Environment variables are a series of hidden values that the web server sends to every CGI program you run. Your program can parse them and use the data they send. Environment variables are stored in a hash named `%ENV`:

<code>DOCUMENT_ROOT</code>	The root directory of your server
<code>HTTP_COOKIE</code>	The visitor's cookie, if one is set
<code>HTTP_HOST</code>	The hostname of the page being attempted
<code>HTTP_REFERER</code>	The URL of the page that called your program
<code>HTTP_USER_AGENT</code>	The browser type of the visitor



HTTPS	"on" if the program is being called through a secure server
PATH	The system path your server is running under
QUERY_STRING	The query string (see GET, below)
REMOTE_ADDR	The IP address of the visitor
REMOTE_HOST	The hostname of the visitor (if your server has reverse-name-lookups on; otherwise this is the IP address again)
REMOTE_PORT	The port the visitor is connected to on the web server
REMOTE_USER	The visitor's username (for .htaccess-protected pages)
REQUEST_METHOD	GET or POST
REQUEST_URI	The interpreted pathname of the requested document or CGI (relative to the document root)
SCRIPT_FILENAME	The full pathname of the current CGI
SCRIPT_NAME	The interpreted pathname of the current CGI (relative to the document root)
SERVER_ADMIN	The email address for your server's webmaster
SERVER_NAME	Your server's fully qualified domain name (e.g. www.cgi101.com)
SERVER_PORT	The port number your server is listening on
SERVER_SOFTWARE	The server software you're using (e.g. Apache 1.3)

Some servers set other environment variables as well; check your server documentation for more information. Notice that some environment variables give information about your server, and will never change (such as SERVER\_NAME and SERVER\_ADMIN), while others give information about the visitor, and will be different every time someone accesses the program.

Not all environment variables get set. REMOTE\_USER is only set for pages in a directory or subdirectory that's password-protected via a .htaccess

file. And even then, REMOTE\_USER will be the username as it appears in the .htaccess file; it's not the person's email address. There is no reliable way to get a person's email address, short of asking them for it with a web form.

You can print the environment variables the same way you would any hash value:

```
print "Caller = $ENV{HTTP_REFERER}\n";
```

Let's try printing some environment variables. Start a new file named env.cgi:

#### **Example 5 : env.cgi - Print Environment Variables Program**

```
#!/usr/bin/perl -wT

use strict;

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

print header;

print start_html("Environment");

foreach my $key (sort(keys(%ENV))) {

    print "$key = $ENV{$key}<br>\n";

}

print end_html;
```

Let's look at several ways to use some of this data.

#### **Referring Page**

When you click on a hyperlink on a web page, you're being referred to another page. The web server for the receiving page keeps track of the referring page, and you can access the URL for that page via the HTTP\_REFERER environment variable. Here's an example:

#### **Example 6 : refer.cgi - HTTP Referer Program**

```
#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;
```

```
print start_html("Referring Page");  
print "Welcome, I see you've just come from  
$ENV{HTTP_REFERER}!<p>\n";  
print end_html;
```

Remember, HTTP\_REFERER only gets set when a visitor actually clicks on a link to your page. If they type the URL directly (or use a bookmarked URL), then HTTP\_REFERER is blank. To properly test your program, create an HTML page with a link to refer.cgi, then click on the link:

[Referring Page](#)

HTTP\_REFERER is not a foolproof method of determining what page is accessing your program. It can easily be forged.

### **Remote Host Name, and Hostname Lookups**

You've probably seen web pages that greet you with a message like "Hello, visitor from (yourhost)!", where (yourhost) is the hostname or IP address you're currently logged in with. This is a pretty easy thing to do because your IP address is stored in the %ENV hash.

If your web server is configured to do hostname lookups, then you can access the visitor's actual hostname from the \$ENV{REMOTE\_HOST} value. Servers often don't do hostname lookups automatically, though, because it slows down the server. Since \$ENV{REMOTE\_ADDR} contains the visitor's IP address, you can reverse-lookup the hostname from the IP address using the Socket module in Perl. As with CGI.pm, you have to use the Socket module:

```
use Socket;
```

(There is no need to add qw(:standard) for the Socket module.)

The Socket module offers numerous functions for socket programming (most of which are beyond the scope of this book). We're only interested in the reverse-IP lookup for now, though. Here's how to do the reverse lookup:

```
my $ip = "209.189.198.102";  
my $hostname = gethostbyaddr(inet_aton($ip), AF_INET);
```

There are actually two functions being called here: gethostbyaddr and inet\_aton. gethostbyaddr is a built-in Perl function that returns the hostname

for a particular IP address. However, it requires the IP address be passed to it in a packed 4-byte format. The `Socket` module's `inet_aton` function does this for you.

Let's try it in a CGI program. Start a new file called `rhost.cgi`, and enter the following code:

#### **Example 7: rhost.cgi - Remote Host Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
use Socket;
print header;
print start_html("Remote Host");
my $hostname = gethostbyaddr(inet_aton($ENV{REMOTE_ADDR}),
AF_INET);
print "Welcome, visitor from $hostname!<p>\n";
print end_html;
```

---

### **17.7. Form processing**

---

#### **A Simple Form Using GET**

There are two ways to send data from a web form to a CGI program: GET and POST. These *methods* determine how the form data is sent to the server.

With the GET method, the input values from the form are sent as part of the URL and saved in the `QUERY_STRING` environment variable. With the POST method, data is sent as an input stream to the program. You can set the `QUERY_STRING` value in a number of ways. For example, here are a number of direct links to the `env.cgi` program:

Try opening each of these in your web browser. Notice that the value for `QUERY_STRING` is set to whatever appears after the question mark in the URL itself.

You can also process simple forms using the GET method. Start a new HTML document called `envform.html`, and enter this form:

**Program 3-5: `envform.html` - Simple HTML Form Using GET**

```
<html><head><title>Test Form</title></head>

<body>

<form action="env.cgi" method="GET">

Enter some text here:

<input type="text" name="sample_text" size=30>

<input type="submit"><p>

</form>

</body></html>
```

Save the form and upload it to your website. Remember you may need to change the path to `env.cgi` depending on your server; if your CGI programs live in a "cgi-bin" directory then you should use `action="cgi-bin/env.cgi"`.

Bring up the form in your browser, then type something into the input field and hit return. You'll notice that the value for `QUERY_STRING` now looks like this:

```
sample_text=whatever+you+typed
```

The string to the left of the equals sign is the name of the form field. The string to the right is whatever you typed into the input box. Notice that any spaces in the string you typed have been replaced with a `+`. Similarly, various punctuation and other special non-alphanumeric characters have been replaced with a `%-code`. This is called *URL-encoding*, and it happens with data submitted through either GET or POST methods.

You can send multiple input data values with GET:

```
<form action="env.cgi" method="GET">

First Name: <input type="text" name="fname" size=30><p>

Last Name: <input type="text" name="lname" size=30><p>

<input type="submit">

</form>
```

This will be passed to the `env.cgi` program as follows:

```
$ENV{QUERY_STRING} = "fname=joe&lname=smith"
```

The two form values are separated by an ampersand (&). You can divide the query string with Perl's `split` function:

```
my @values = split(/&/,$ENV{QUERY_STRING});
```

`split` lets you break up a string into a list of strings, splitting on a specific character. In this case, we've split on the "&" character. This gives us an array named `@values` containing two elements: ("fname=ramesh", "lname=thanappan"). We can further split each string on the "=" character using a `foreach` loop:

```
foreach my $i (@values) {  
    my($fieldname, $data) = split(/=/, $i);  
    print "$fieldname = $data<br>\n";  
}
```

This prints out the field names and the data entered into each field in the form. It does not do URL-decoding, however. A better way to parse `QUERY_STRING` variables is with `CGI.pm`.

### **Using CGI.pm to Parse the Query String**

If you're sending more than one value in the query string, it's best to use `CGI.pm` to parse it. This requires that your query string be of the form:

```
fieldname1=value1
```

For multiple values, it should look like this:

```
fieldname1=value1&fieldname2=value2&fieldname3=value3
```

This will be the case if you are using a form, but if you're typing the URL directly then you need to be sure to use a `fieldname`, an equals sign, then the field value.

`CGI.pm` provides these values to you automatically with the `param` function:

```
param('fieldname');
```

This returns the value entered in the `fieldname` field. It also does the URL-decoding for you, so you get the exact string that was typed in the form field.

You can get a list of all the fieldnames used in the form by calling param with no arguments:

```
my @fieldnames = param();
```

### **param is NOT a Variable**

param is a function call.

```
print "$p = param($p)<br>\n";
```

If you want to print the value of param(\$p), you can print it by itself:

```
print param($p);
```

Or call param outside of the double-quoted strings:

```
print "$p = ", param($p), "<br>\n";
```

You won't be able to use param('fieldname') inside a here-document. You may find it easier to assign the form values to individual variables:

```
my $firstname = param('firstname');
```

```
my $lastname = param('lastname');
```

Another way would be to assign every form value to a hash:

```
my(%form);  
foreach my $p (param()) {  
    $form{$p} = param($p);  
}
```

You can achieve the same result by using CGI.pm's Vars function:

```
use CGI qw(:standard Vars);  
my %form = Vars();
```

The Vars function is not part of the "standard" set of CGI.pm functions, so it must be included specifically in the use statement.

Either way, after storing the field values in the %form hash, you can refer to the individual field names by using \$form{'fieldname'}.

Let's try it now. Create a new form called getform.html:

### **Example 8 : getform.html - Another HTML Form Using GET**

```
<html><head><title>Test Form</title></head>
```

```

<body>

<form action="get.cgi" method="GET">

First Name: <input type="text" name="firstname" size=30><br>
Last Name: <input type="text" name="lastname" size=30><br>
<input type="submit"><p>

</form>

</body></html>

```

Save and upload it to your webserver, then bring up the form in your web browser.

Now create the CGI program called get.cgi:

### **Example 9 : get.cgi Form Processing Program Using GET**

```

#!/usr/bin/perl -wT

use CGI qw(:standard);

use CGI::Carp qw(warningsToBrowser fatalsToBrowser);

use strict;

print header;

print start_html("Get Form");

my %form;

foreach my $p (param()) {
    $form{$p} = param($p);
    print "$p = $form{$p}<br>\n";
}

print end_html;

```

Take a look at the full URL of get.cgi after you press submit. You should see all of your form field names and the data you typed in as part of the URL. This is one reason why GET is not the best method for handling forms; it isn't secure.

### **Processing Forms using POST**

Most forms you create will send their data using the POST method. POST is more secure than GET, since the data isn't sent as part of the URL, and you



can send more data with POST. Also, your browser, web server, or proxy server may cache GET queries, but posted data is resent each time.

Your web browser, when sending form data, encodes the data being sent. Alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters — like tabs, quotes, etc. — are converted to "%HH" — a percent sign and two hexadecimal digits representing the ASCII code of the character. This is called URL encoding.

In order to do anything useful with the data, your program must decode these. Fortunately the CGI.pm module does this work for you. You access the decoded form values the same way you did with GET:

```
$value = param('fieldname');
```

So you already know how to process forms! You can try it now by changing your getform.html form to method="POST" (rather than method="GET"). You'll see that it works identically whether you use GET or POST. Even though the data is sent differently, CGI.pm handles it for you automatically.

## **Guestbook Form**

One of the first CGI programs you're likely to want to add to your website is a guestbook program, so let's start writing one. First create your HTML form. The actual fields can be up to you, but a bare minimum might look like this:

```
<form action="post.cgi" method="POST">
Your Name: <input type="text" name="name"><br>
Email Address: <input type="text" name="email"><br>
Comments:<br>
<textarea name="comments" rows="5"
  cols="60"></textarea><br>
<input type="submit" value="Send">
</form>
```

Now you need to create `post.cgi`. This is nearly identical to the `get.cgi` from last chapter, so you may just want to copy that program and make changes:

**Example 9 : `post.cgi` - Form Processing Program Using POST**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
print header;
print start_html("Thank You");
print h2("Thank You");
my %form;
foreach my $p (param()) {
    $form{$p} = param($p);
    print "$p = $form{$p}<br>\n";
}
print end_html;
```

Test your program by entering some data into the fields, and pressing "send" when finished. Notice that the data is not sent in the URL this time, as it was with the GET example.

Of course, this form doesn't actually DO anything with the data, which doesn't make it much of a guestbook. Let's see how to send the data in e-mail.

**Validating Form Data**

You should always *validate* data submitted on a form; that is, check to see that the form fields aren't blank, and that the data submitted is in the format you expected. This is typically done with `if/elsif` blocks.

Here are some examples. This condition checks to see if the "name" field isn't blank:

```
if (param('name') eq "") {
    &dienice("Please fill out the field for your name.");
}
```

```
}
```

You can also test multiple fields at the same time:

```
if (param('name') eq "" or param('email') eq "") {  
    &dienice("Please fill out the fields for your name  
    and email address.");  
}
```

The above code will return an error if either the name or email fields are left blank.

`param('fieldname')` always returns one of the following:

undef — or undefined	fieldname is not defined in the form itself, or it's a checkbox/radio button field that wasn't checked.
the empty string	fieldname exists in the form but the user didn't type anything into that field (for text fields)
one or more values	whatever the user typed into the field(s)

If your form has more than one field containing the same `fieldname`, then the values are stored sequentially in an array, accessed by `param('fieldname')`.

You should always validate all form data — even fields that are submitted as hidden fields in your form. Don't assume that your form is always the one calling your program. Any external site can send data to your CGI. Never trust form input data.

---

## 17.8. Sending Email

---

There are several ways to send mail. We'll be using the `sendmail` programme under Unix. Before you can write your form-to-mail CGI program, you'll need to figure out where the `sendmail` program is installed on your webserver.

Since we're using the `-T` flag for taint checking, the first thing you need to do before connecting to `sendmail` is set the `PATH` environment variable:

```
$ENV{PATH} = "/usr/sbin";
```

The path should be the directory where sendmail is located; if sendmail is in /usr/sbin/sendmail, then \$ENV{PATH} should be "/usr/sbin". If it's in /var/lib/sendmail, then \$ENV{PATH} should be "/var/lib".

Next you open a *pipe* to the sendmail program:

```
open (MAIL, "| /usr/sbin/sendmail -t -oi") or  
die "Can't fork for sendmail: $!\n";
```

The pipe (which is indicated by the | character) causes all of the output printed to that filehandle (MAIL) to be fed directly to the /usr/sbin/sendmail program as if it were standard input to that program. Several flags are also passed to sendmail:

- t    Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for recipient addresses
  - Ignore dots alone on lines by themselves in incoming messages.
- oi

The -t flag tells sendmail to look at the message headers to determine who the mail is being sent to. You'll have to print all of the message headers yourself:

```
my $recipient = 'recipient@b-u.ac.in;  
print MAIL "From: sender\@b-u.ac.in\n";  
print MAIL "To: $recipient\n";  
print MAIL "Subject: Guestbook Form\n\n";
```

Remember that you can safely put an @-sign inside a single-quoted string, like 'recipient@b-u.ac.in', or you can escape the @-sign in double-quoted strings by using a backslash ("sender\@b-u.ac.in").

The message headers are complete when you print a single blank line following the header lines. We've accomplished this by printing two newlines at the end of the subject header:

```
print MAIL "Subject: Guestbook Form\n\n";
```

After that, you can print the body of your message. Let's try it. Start a new file named `guestbook.cgi`, and edit it as follows. You don't need to include the comments in the following code; they are just there to show you what's happening.

**Example 9 : `guestbook.cgi` - Guestbook Program**

```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
use strict;
print header;
print start_html("Results");
# Set the PATH environment variable to the same path
# where sendmail is located:
$ENV{PATH} = "/usr/sbin";
# open the pipe to sendmail
open (MAIL, "| /usr/sbin/sendmail -oi -t") or
    &dienice("Can't fork for sendmail: $!\n");
# change this to your own e-mail address
my $recipient = 'nullbox@b-u.ac.in';
# Start printing the mail headers
# You must specify who it's to, or it won't be delivered:
print MAIL "To: $recipient\n";
# From should probably be the webserver.
print MAIL "From: nobody\@b-u.ac.in\n";
# print a subject line so you know it's from your form cgi.
print MAIL "Subject: Form Data\n\n";
# Now print the body of your mail message.
foreach my $p (param()) {
    print MAIL "$p = ", param($p), "\n";
}
# Be sure to close the MAIL input stream so that the
# message actually gets mailed.
```

```

close(MAIL);

# Now print a thank-you page

print <<EndHTML;

<h2>Thank You</h2>

<p>Thank you for writing!</p>

<p>Return to our <a href="index.html">home page</a>.</p>

EndHTML

print end_html;

```

Save the file, then modify your guestbook.html form so that the action points to guestbook.cgi:

```
<form action="guestbook.cgi" method="POST">
```

Try testing the form. If the program runs successfully, you'll get e-mail in a few moments with the results of your post.

---

## 17.9. Verifying user name and password

---

Verifying user name and password is a common security practice on the Internet. In normal circumstances, all the user names and their corresponding passwords are stored in a file on the server. Therefore, to verify user name and password for a particular user on the Internet, a CGI application is required. Since we don't have encryption at this moment, the implementation of password checking is relatively simple. Suppose you have password file called password.txt containing all the user names and passwords. To check a password for a particular user it is necessary to read this file and perform the comparison. To implement this, we have two programmes one for screen display and another for password verification.

Example : Display Screen

```

<html>

<head><title>Username and Password </title></head>

<style>

.butSt{background-color:#aaffaa;font-family:arial;font-weight:bold;

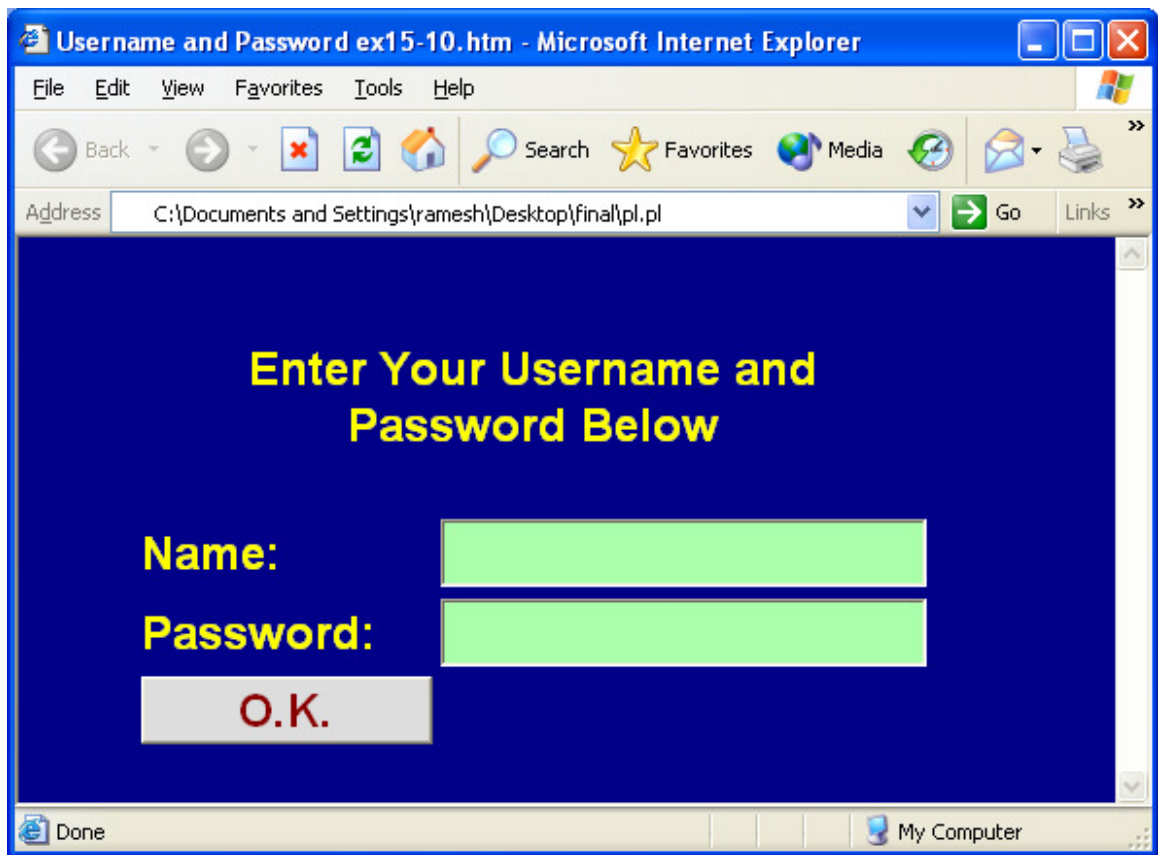
font-size:18pt;color:#880000;width:250px;height:35px}

```

```

.txtSt{font-family:arial;font-weight:bold; text-align:left;
font-size:18pt;color:#ffff00}</style>
<body style="background:#000088">
<form action="ex15-10.pl" method="post">
<table style="position:absolute;left:60px;top:50px" class="txtSt">
<tr><td colspan="2" style="text-align:center">
Enter Your Username and<br /> Password Below<br
/><br/></td></tr>
<tr><td>Name:</td><td><input type="text" name="userId" id="userId"
class="butSt" ></td></tr>
<tr><td>Password:</td><td><input type="password" name="passId"
id="passId" class="butSt"></td></tr>
<tr><td><input type="submit" class="butSt" value="O.K."
style="width:150px;background:#dddddd"></td></tr>
</table>
</form>
</body>
</html>

```



When the user name and password are filled and the O.K. button is clicked, the form is sent to the Perl script for processing. The program code of the script is listed as follows:

```
#!/usr/bin/perl
use CGI qw (:standard);
my $username = param(userId);
my $password = param(passId);
print "Content-type:text/html\n\n";
print << "mypage";
<?xml version="1.0" encoding="iso-8859-1"?>
<!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" xml:lang="en"
lang="en">
```



```

<head><title>Example: ex15-10.pl</title></head>
<style>.txtSt{font-size:18pt;color:#ffff00;font-family:arial}</style>
<body style="background:#000088;font-weight:bold" class="txtSt">
Mypage open(filehandle, "password.txt") or die "The File could not be
opened .. Error.";
while(my $st = <filehandle>)
{
    $st =~ s/\n//g;
    ($name, $pass) = split(/,/, $st);
    if($name eq "$username")
    {
        $userF = 1;
        if ($pass eq "$password")
        {
            $passwordF = 1;
        }
    }
}
close(filehandle);
if ($userF && $passwordF)
{
    print ("Thank you -- $username <br /> Access Granted !!
        <br />Enjoy Your Visit.");
}
elsif ($userF && !$passwordF)
{
    print ("Sorry, Wrong Password !!");
}
else

```

```
{  
    print ("Sorry, Access Denied !!");  
}  
  
print "</body></html>";
```

---

## 17.10. Let us Sum Up

---

CGI/Perl is very similar to HTML; it has a clearly defined syntax, and if you follow those syntax rules, you can write Perl as easily as you do HTML.

The first line of your program should look like this:

```
#!/usr/bin/perl -wT
```

The first part, `#!`, indicates that this is a script.

The next part, `/usr/bin/perl`, is the location (or *path*) of the Perl interpreter.

The final part contains optional flags for the Perl interpreter.

Warnings are enabled by the `-w` flag.

A CGI is simply a program that is called by the webserver, in response to some action by a web user. This might be something simple like a page counter, or a complex form-handler.

Perl offers a powerful feature to programmers: add-on modules. These are collections of pre-written code that you can use to do all kinds of tasks. You can save yourself the time and trouble of reinventing the wheel by using these modules.

Some modules are included as part of the Perl distribution; these are called *standard library modules* and don't have to be installed. If you have Perl, you already have the standard library modules.

There are also many other modules available that are not part of the standard library. These are typically listed on the Comprehensive Perl Archive Network (CPAN), which you can search on the web at <http://search.cpan.org/>.

Perl has built-in data types that represent different kinds of data. Each variable has a specific character preceding.

<b>Data Type</b>	<b>Format</b>	<b>Description</b>
Scalar	\$scalarname	Can be string, an integer number, a floating-point number or a reference
Array	@arrayname	An ordered list of scalar variables that can be accessed using integer indices
Hash	%hashname	An unordered set of scalar variables whose values are accessed using unique scalar value

## **Documenting Your Programs**

Documentation can be embedded in a program using comments. A comment in Perl is preceded by the # sign; anything appearing after the # is a comment:

## **String Manipulation**

Perl is very well known for ease in manipulation of strings. What might take other programming languages several lines of code can generally be done in Perl in less than 2 lines.

### **The dot operator (.)**

This allows you to concatenate two strings together into one string.

### **The index function**

This allows you to find an occurrence of one string within another and return the position of that occurrence.

### **The substring function**

This allows you to copy part of a string from another by specifying beginning position and (in one version) length of string.

### **The length function**

This function returns the length in characters of the string.

### **The split function**

This allows you to split up one string into multiple strings by looking for occurrences of a character or characters within it. The results are stored into an array.

## **The join function**

This does the exact opposite of `split()`. Here you give an array of scalars and a delimiting character. Join will glue it all together.

## **Regular Expressions**

In regular expressions, generally, a character matches itself. The only exceptions are regular expression special characters. To match one of these special characters, you must put a `\` before the character.

### **The \* and . Special Character**

The `*` is used to indicate that zero or more of the previous characters should be matched. Thus, the regular expression `a*` will match any string that contains zero or more a's. Note that since `a*` will match any string with zero or more a's, `a*` will match all strings, since all strings (including the empty string) contain at least zero a's. So, `a*` is not a very useful regular expression.

The next special character we will consider is the `.` character. The `.` will match any valid character. As an example, consider the regular expression `a.c`. This regular expression will match any string that contains an a and a c, with any possible character in between. Thus, strings that contain `abc`, `acc`, `amc`, etc. are all in the class of strings that this regular expression matches.

### **The | Character**

The `|` special character is equivalent to an “or” in regular expressions. This character is used to give a choice. So, the regular expression `abc|def` will match any string that contains either `abc` or `def`.

### **Grouping with ()s**

Sometimes, within regular expressions, we want to group things together. Doing this allows building of larger regular expressions based on smaller components. The `()`'s are used for grouping

### **The Anchor Characters**

The anchor operators allow us to do this. When we start a regular expression with a `^`, it anchors the regular expression to the beginning of the string. This means that whatever the regular expression starts with must be matched at the beginning of the string. For example, `^aa*` will not match strings

that contain one or more a's; rather it matches strings that start with one or more a's.

We can also use the \$ at the end of the string to anchor the regular expression at the end of the string.

## **Pattern Matching**

Now that you are familiar with some of the basics of regular expressions, you probably want to know how to use them in Perl. Doing so is very easy. There is an operator, `=~`, that you can use to match a regular expression against scalar variables. Regular expressions in Perl are placed between two forward slashes (i.e., `//`). The whole `$scalar =~ //` expression will evaluate to 1 if a match occurs, and `undef` if it does not.

## **CGI Environment Variables**

Environment variables are a series of hidden values that the web server sends to every CGI program you run. Your program can parse them and use the data they send. Environment variables are stored in a hash named `%ENV`:

<b>Key</b>	<b>Value</b>
<code>DOCUMENT_ROOT</code>	The root directory of your server
<code>HTTP_COOKIE</code>	The visitor's cookie, if one is set
<code>HTTP_HOST</code>	The hostname of the page being attempted
<code>HTTP_REFERER</code>	The URL of the page that called your program
<code>HTTP_USER_AGENT</code>	The browser type of the visitor
<code>HTTPS</code>	"on" if the program is being called through a secure server
<code>PATH</code>	The system path your server is running under
<code>QUERY_STRING</code>	The query string (see GET, below)
<code>REMOTE_ADDR</code>	The IP address of the visitor
<code>REMOTE_HOST</code>	The hostname of the visitor (if your server has reverse-name-lookups on; otherwise this is the IP address again)

REMOTE_PORT	The port the visitor is connected to on the web server
REMOTE_USER	The visitor's username (for .htaccess-protected pages)
REQUEST_METHOD	GET or POST
REQUEST_URI	The interpreted pathname of the requested document or CGI (relative to the document root)
SCRIPT_FILENAME	The full pathname of the current CGI
SCRIPT_NAME	The interpreted pathname of the current CGI (relative to the document root)
SERVER_ADMIN	The email address for your server's webmaster
SERVER_NAME	Your server's fully qualified domain name (e.g. www.cgi101.com)
SERVER_PORT	The port number your server is listening on
SERVER_SOFTWARE	The server software you're using (e.g. Apache 1.3)

## **Form processing**

### **A Simple Form Using GET**

There are two ways to send data from a web form to a CGI program: GET and POST. These *methods* determine how the form data is sent to the server.

With the GET method, the input values from the form are sent as part of the URL and saved in the QUERY\_STRING environment variable. With the POST method, data is sent as an input stream to the program. You can set the QUERY\_STRING value in a number of ways. For example, here are a number of direct links to the env.cgi program:

Try opening each of these in your web browser. Notice that the value for QUERY\_STRING is set to whatever appears after the question mark in the URL itself.

You can also process simple forms using the GET method. Start a new HTML document called envform.html, and enter this form:

### **Processing Forms using POST**

Most forms you create will send their data using the POST method. POST is more secure than GET, since the data isn't sent as part of the URL, and you can send more data with POST. Also, your browser, web server, or proxy server may cache GET queries, but posted data is resent each time.

Your web browser, when sending form data, encodes the data being sent. Alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters — like tabs, quotes, etc. — are converted to "%HH" — a percent sign and two hexadecimal digits representing the ASCII code of the character. This is called URL encoding.

## **Sending Email**

There are several ways to send mail. We'll be using the sendmail program under Unix. Before you can write your form-to-mail CGI program, you'll need to figure out where the sendmail program is installed on your webserver.

Since we're using the -T flag for taint checking, the first thing you need to do before connecting to sendmail is set the PATH environment variable:

```
$ENV{PATH} = "/usr/sbin";
```

The path should be the directory where sendmail is located; if sendmail is in /usr/sbin/sendmail, then \$ENV{PATH} should be "/usr/sbin". If it's in /var/lib/sendmail, then \$ENV{PATH} should be "/var/lib".

Next you open a *pipe* to the sendmail program:

```
open (MAIL, "| /usr/sbin/sendmail -t -oi") or  
die "Can't fork for sendmail: $!\n";
```

The pipe (which is indicated by the | character) causes all of the output printed to that filehandle (MAIL) to be fed directly to the /usr/sbin/sendmail program as if it were standard input to that program. Several flags are also passed to sendmail:

-t     Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for recipient addresses

- Ignore dots alone on lines by themselves in incoming messages.  
oi

The `-t` flag tells `sendmail` to look at the message headers to determine who the mail is being sent to. You'll have to print all of the message headers yourself:

```
my $recipient = 'recipient@b-u.ac.in;  
print MAIL "From: sender\@b-u.ac.in\n";  
print MAIL "To: $recipient\n";  
print MAIL "Subject: Guestbook Form\n\n";
```

Remember that you can safely put an `@`-sign inside a single-quoted string, like `'recipient@b-u.ac.in'`, or you can escape the `@`-sign in double-quoted strings by using a backslash (`"sender\@b-u.ac.in"`).

The message headers are complete when you print a single blank line following the header lines. We've accomplished this by printing two newlines at the end of the subject header:

```
print MAIL "Subject: Guestbook Form\n\n";
```

After that, you can print the body of your message. Let's try it. Start a new file named `guestbook.cgi`, and edit it as follows. You don't need to include the comments in the following code; they are just there to show you what's happening.

### **Verifying user name and password**

Verifying user name and password is a common security practice on the Internet. In normal circumstances, all the user names and their corresponding passwords are stored in a file on the server. Therefore, to verify user name and password for a particular user on the Internet, a CGI application is required. Since we don't have encryption at this moment, the implementation of password checking is relatively simple. Suppose you have password file called `password.txt` containing all the user names and passwords. To check a password for a particular user it is necessary to read this file and perform the comparison.



---

**17.10. Lesson end Activities**

---

1. What is CGI module?
2. Explain Perl Regular expression.
3. What are the function available in Perl for string processing?

---

**17.11. Check your progress**

---

1. Write a perl program for validating user name and password.
2. How Form processing is implemented in Perl?

---

**17.12. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.perl.org](http://www.perl.org).



---

**PERL ODBC and Cookies**

---

**Contents****18.0. Aim and Objective****18.1. Introduction****18.2. SQL****18.3. Perl DBI****18.4. Cookies****18.5. Let us sum Up****18.6. Lesson end Activities****18.7. Check your Progress****18.8. Reference**

---

**18.0. Aim and Objective**

---

- **To understand PERL DBI**
- **To learn database connectivity**
- **To learn cookies**

---

**18.1. Introduction**

---

A relational database is a bunch of rectangular tables. Each row of a table is a record about one person or thing; the record contains several pieces of information called *fields*. Here is an example table:

LASTNAME	FIRSTNAME	ID	POSTAL_CODE	AGE	SEX
Gauss	Karl	119	19107	30	M
Smith	Mark	3	T2V 3V4	53	M
Noether	Emmy	118	19107	31	F
Smith	Jeff	28	K2G 5J9	19	M
Hamilton	William	247	10139	2	M

---

## 18.2. SQL

---

SQL stands for *Structured Query Language*. It was invented at IBM in the 1970's. It's a language for describing searches and modifications to a relational database. SQL was a huge success, probably because it's incredibly simple and anyone can pick it up in ten minutes. As a result, all the important database systems support it in some fashion or another.

Important SQL commands :

### **SELECT**

Find all the records that have a certain property

### **INSERT**

Add new records

### **DELETE**

Remove old records

### **UPDATE**

Modify records that are already there

Those are the four most important SQL commands, also called *queries*. Suppose that the example table above is named `people`. Here are examples of each of the four important kinds of queries:

```
SELECT firstname FROM people WHERE lastname = 'Smith'
```

(Locate the first names of all the Smiths.)

```
DELETE FROM people WHERE id = 3
```

(Delete Mark Smith from the table)

```
UPDATE people SET age = age+1 WHERE id = 247
```

(William Hamilton just had a birthday.)

```
INSERT INTO people VALUES ('Euler', 'Leonhard', 248, NULL, 58, 'M')
```

(Add Leonhard Euler to the table.)

---

### 18.3. Perl DBI

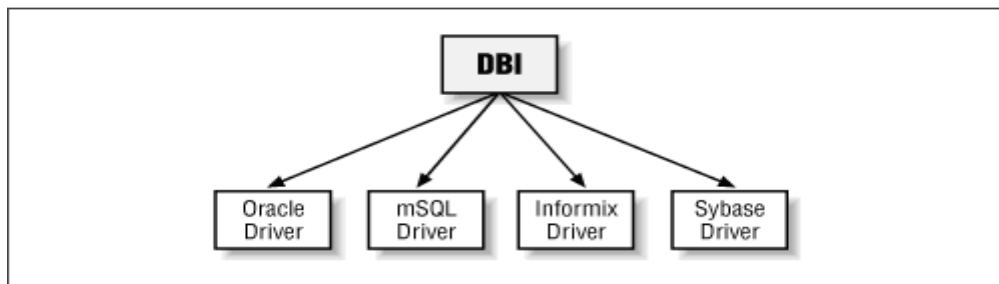
---

Perl's DBI ('Database Interface') module was written by Tim Bunce. DBI is designed to protect you from the details of the vendor libraries. It has a very simple interface for saying what SQL queries you want to make, and for getting the results back. DBI doesn't know how to talk to any particular database, but it does know how to locate and load in DBD ('Database Driver') modules. The DBD modules have the vendor libraries in them and know how to talk to the real databases; there is one DBD module for every different database.

When you ask DBI to make a query for you, it sends the query to the appropriate DBD module, which spins around three times or drinks out of its sneaker or whatever is necessary to communicate with the real database. When it gets the results back, it passes them to DBI. Then DBI gives you the results. Since your program only has to deal with DBI, and not with the real database, you don't have to worry about barking like a chicken.

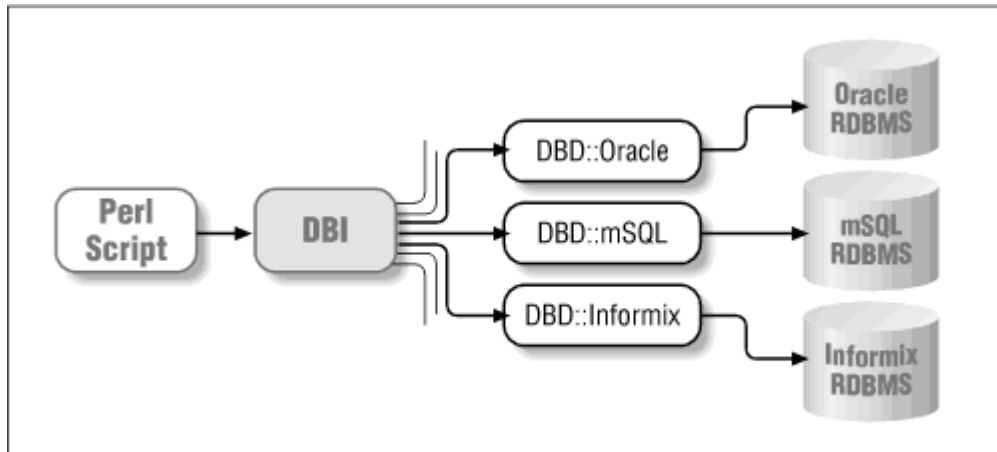
#### ***DBI Architecture***

The DBI architecture is split into two main groups of software: the DBI itself, and the *drivers*. The DBI defines the actual DBI programming interface, routes method calls to the appropriate drivers, and provides various support services to them. Specific drivers are implemented for each different type of database and actually perform the operations on the databases. The following figure illustrates this architecture.



**Figure.1 : The DBI Architecture**

The following figure shows the flow of data from a Perl script through to the database.



**Figure 2: Data flow through DBI**

Under this architecture, it is relatively straightforward to implement a driver for any type of database. All that is required is to implement the methods defined in the DBI specification, as supported by the DBI module, in a way that is meaningful for that database. The data returned from this module is passed back into the DBI module, and from there it is returned to the Perl program. All the information that passes between the DBI and its drivers is standard Perl datatypes, thereby preserving the isolation of the DBI module from any knowledge of databases.

The separation of the drivers from the DBI itself makes the DBI a powerful programming interface that can be extended to support almost any database available today. Drivers currently exist for many popular databases including Oracle, Informix, mSQL, MySQL, Ingres, Sybase, DB2, Empress, SearchServer, and PostgreSQL. There are even drivers for XBase and CSV files.

Drivers are also called database drivers, or DBDs, after the namespace in which they are declared. For example, Oracle uses DBD::Oracle, Informix uses DBD::Informix, and so on. A useful tip in remembering the DBI architecture is that DBI can stand for DataBase Independent and DBD can stand for DataBase Dependent.

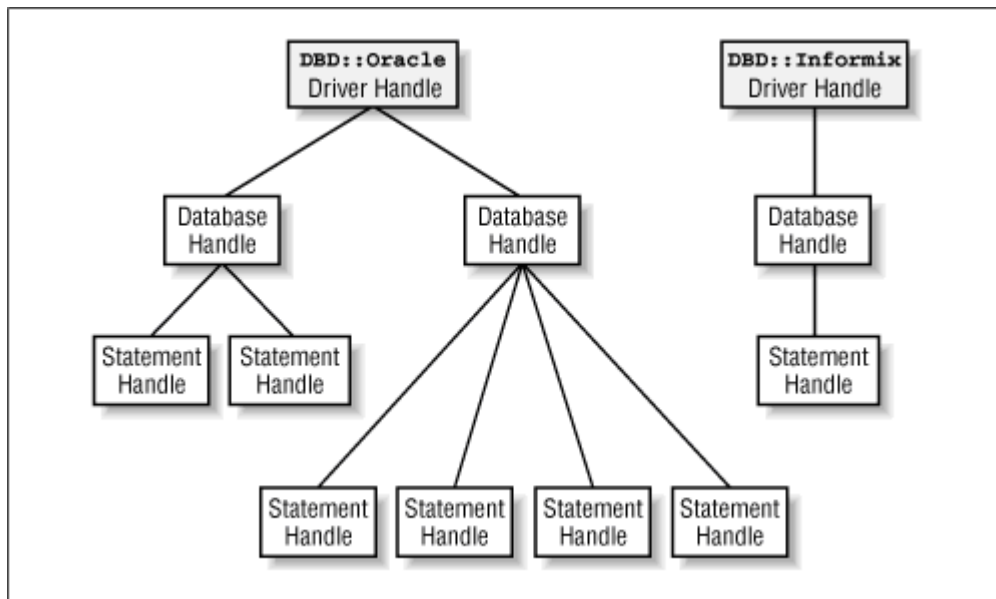
Because DBI uses Perl's object-orientation features, it is extremely simple to initialize DBI for use within your programs. This can be achieved by adding the line:

```
use DBI;
```

to the top of your programs. This line locates and loads the core DBI module. Individual database driver modules are loaded as required, and should generally not be explicitly loaded.

### ***Handles***

The DBI defines three main types of objects that you may use to interact with databases. These objects are known as *handles*. There are handles for drivers, which the DBI uses to create handles for database connections, which, in turn, can be used to create handles for individual database commands, known as statements. The following figure illustrates the overall structure of the way in which handles are related, and their meanings are described in the following sections.



**Figure 3. DBI handles**

### **Driver Handles**

*Driver handles* represent loaded drivers and are created when the driver is loaded and initialized by the DBI. There is exactly one driver handle per loaded driver. Initially, the driver handle is the only contact the DBI has with the driver, and at this stage, no contact has been made with any database through that driver.

The only two significant methods available through the driver handle are `data_sources()`, to enumerate what can be connected to, and `connect()`, to actually make a connection. These methods are more commonly invoked as DBI class methods, however, which we will discuss in more detail later in this chapter.

Since a driver handle completely encapsulates a driver, there's no reason why multiple drivers can't be simultaneously loaded. This is part of what makes the DBI such a powerful interface.

Within the DBI specification, a driver handle is usually referred to as `$drh`.

Driver handles should not normally be referenced within your programs. The actual instantiation of driver handles happens ``under the hood" of DBI, typically when `DBI->connect()` is called.

### **Database Handles**

Database handles are the first step towards actually doing work with the database, in that they encapsulate a single connection to a particular database. Prior to executing SQL statements within a database, we must actually *connect* to the database. This is usually achieved through the DBI's `connect()` method:

```
$dbh = DBI->connect( $data_source, ... );
```

Within the DBI specification and sample code, database handles are usually referred to as `$dbh`.

### **Statement Handles**

*Statement handles* are the final type of object that DBI defines for database interaction and manipulation. These handles actually encapsulate individual SQL statements to be executed within the database.

Statement handles are *children* of their corresponding database handle. Since statement handles are objects in their own right, data within one



statement is protected from tampering or modification by other statement handles.

Within the DBI specification and sample code, statement handles are generally referred to as `$sth`.

## **Data Source Names**

When connecting to a database via the DBI, you need to tell the DBI where to find the database to connect to. For example, the database driver might require a database name, or a physical machine name upon which the database resides. This information is termed a *data source name*, and of all the aspects of DBI, this is possibly the most difficult to standardize due to the sheer number and diversity of connection syntaxes.

The DBI requires the data source name to start with the characters `dbi:`, much like a URL begins with `http:`, and then the name of the driver, followed by another colon--for example, `dbi:Oracle:`. Any text that follows is passed to the driver's own `connect()` method to interpret as it sees fit. Most drivers expect either a simple database name or, more often, a set of one or more name/value pairs separated with semicolons. Some common examples are listed later in this section.

DBI offers two useful methods for querying which data sources are available to you for each driver you have installed on your system.

Firstly, you can get a list of all the available drivers installed on your machine by using the `DBI->available_drivers()` method. This returns a list with each element containing the data source prefix of an installed driver, such as `dbi:Informix:`.

Secondly, you can invoke the `DBI->data_sources()` method against one or more of the drivers returned by the `DBI->available_drivers()` method to enumerate which data sources are known to the driver. Calling the `data_sources()` method will actually load the specified driver and validate that it is completely and correctly installed. Because DBI dies if it can't load and initialize a driver, this method should be called inside an `eval{}`  block if you need to catch that error.

## **Connection**

In the case of simple databases, such as flat-file or Berkeley DB files, "connecting" is usually as simple as opening the files for reading or using the tie mechanism. However, in larger database systems, connecting may be considerably more complicated.

By looking at a broad spectrum of database systems, the information required to connect can be boiled down to:

1. The *data source name*, a string containing information specifying the driver to use, what database you wish to connect to, and possibly its whereabouts. This argument takes the format discussed in the previous section and is highly database-specific.
2. The *username* that you wish to connect to the database as. To elaborate on the concept of usernames a little further, some databases partition the database into separate areas, called schemas, in which different users may create tables and manipulate data. Users cannot affect tables and data created by other users. This setup is similar to accounts on a multiuser computer system, in that users may create their own files, which can be manipulated by them, but not necessarily by other users. In fact, users may decide to disallow all access to their files, or tables, from all other users, or allow access to a select group or all users.[\[6\]](#)

Most major database systems enforce a similar security policy, usually with an administrator having access to an account that allows them to read, modify, and delete any user's tables and data. All other users must connect as themselves. On these systems, your database username may be the same as your system login username, but it doesn't have to be.

More minimal database systems may not have any concept of username-based authentication, but you still need to supply the username and password arguments, typically as empty strings.

3. The *password* associated with the supplied username.

In light of these common arguments, the syntax for connecting to databases using DBI is to use the `connect()` call, defined as follows:

```
$dbh = DBI->connect( $data_source, $username, $password, \%attr );
```

The final argument, `\%attr`, is optional and may be omitted. `\%attr` is a reference to a hash that contains *handle attributes* to be applied to this connection. One of the most important items of the information supplied in this hash is whether or not automatic error handling should be supplied by DBI. We will discuss this in further detail in the following section, but the two common attributes are called `RaiseError` and `PrintError`, which cause the DBI to die or print a warning automatically when a database error is detected.

This method, when invoked, returns a *database handle* if the connection has been successfully made to the database. Upon failure, the value `undef` is returned.

To illustrate the `DBI->connect()` method, assume that we have an Oracle database called `archaeo`. To connect to this database, we might use the following code:

```
#!/usr/bin/perl -w

#
# ch04/connect/ex1: Connects to an Oracle database.
use DBI;          # Load the DBI module
### Perform the connection using the Oracle driver
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
    or die "Can't connect to Oracle database: $DBI::errstr\n";
exit;
```

## Disconnection

Explicit disconnection from the database is not strictly necessary if you are exiting from your program after you have performed all the work, but it is a good idea. We strongly recommend that you get into the habit of disconnecting explicitly.

DBI provides a method through which programmers may disconnect a given database handle from its database. This is good practice, especially in programs in which you have performed multiple connections or will be carrying out multiple sequential connections.

The method for performing disconnections is:

```
$rc = $dbh->disconnect();
```

According to this definition, `disconnect()` is invoked against a specific database handle. This preserves the notion that database handles are completely discrete. With multiple database handles active at any given time, each one must explicitly be disconnected.

An example of using `disconnect()` might look like:

```
#!/usr/bin/perl -w

#
# ch04/disconnect/ex1: Connects to an Oracle database
#
#           with auto-error-reporting disabled
#           then performs an explicit disconnection.
use DBI;      # Load the DBI module

### Perform the connection using the Oracle driver
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" ,
{
    PrintError => 0
} )
or die "Can't connect to Oracle database: $DBI::errstr\n";

### Now, disconnect from the database

$dbh->disconnect

or warn "Disconnection failed: $DBI::errstr\n";

exit;
```

Upon successful disconnection, the return value will be true. Otherwise, it will be false. In practice, failure to disconnect usually means that the connection has already been lost for some reason. After disconnecting the database handle can't be used for anything worthwhile.

The following program use a simple SQL statement

```
use strict;
```

```
use DBI;
```

```

my $dbh = DBI->connect( 'dbi:Oracle:orcl','ramesh','ramspassword', {
RaiseError => 1, AutoCommit => 0 }) || die "Database connection not
made :
$DBI::errstr";
my $sql = qq { CREATE TABLE employees ( id INTEGER NOT NULL,
name
VARCHAR2(128), title VARCHAR2(128), phone CHAR(8) ) };
$dbh->do ( $sql);
$dbh->disconnect();

```

---

## 18.4. Cookies

---

The Web is designed to be stateless. Each document sent from server to browser is a unique transaction. Having sent a document, the server forgets about it, which greatly simplifies programming a Web server. But it doesn't simplify writing Web applications. To implement shopping carts, user preferences, or detailed visitor tracking, you need to recognize a user from one page to the next. By the time the Web needed this level of sophistication, it was too late to turn around and make it support states. So Netscape invented cookies.

A cookie is like an ID card that a server issues to a browser. Every time the browser requests something from the server, it flashes the ID card, with all of the information left by the server, as part of the transaction. To track people on your Web site, you issue them a cookie with identifying information on their first visit to your server. Every subsequent request they make will include that cookie, from which you can extract their identity.

Cookies aren't foolproof. Browsers often store cookie data in readable files, so don't use them to hold information such as credit card numbers or passwords. And the browser, which must send cookies back to the server, may not have them enabled. It's informative to clear the cookies from your Web browser, disable cookie functionality, then surf for a while to see how many of your favorite sites no longer work.

Cookies are also pretty limited in what they can hold. Netscape's original specification set limits at 4K of cookie data, 20 cookies per domain, and 300 cookies per browser. Browsers past their limit may prematurely expire, that is delete, older cookies. Some browsers go beyond these limits, but you can't rely on it.

### **Cookie attributes**

Cookies contain a lot of information, the most useful being the name, a string that identifies the cookie, and the value, the data associated with that name.

If Web server `www.example.com` sends a cookie to your browser, your browser won't send it back to just any old server. The cookie has an associated domain, (such as `.example.com` or `.intranet.example.com`), and the browser sends the cookie only to servers that match that domain.

You can even limit which pages on the server receive the cookie with the path cookie option. A cookie with the path `/staff` will be sent to `/staff/contacts.html` or `/staff-images/kevin.jpg`, but not to `/~loser/cookie-grabber.cgi`.

In addition, you can specify that the cookie should be sent only for secure (HTTPS) requests. You might use this if you don't trust the network to be secure, but you do trust the client machine. However, remember that the Web browser may record the cookie in a file, so if that file can be read by a malicious hacker, it defeats the purpose of securing the network transmission.

And finally, you can set an expiration time for the cookie. A cookie with no expiration time disappears when the browser shuts down. You must specify expiration times precisely, such as `Sun, 10-10-00 15:02:19 GMT`, but dates such as these are inconvenient to work with. Fortunately, the [CGI.pm](#) module that we'll use to create and manipulate cookies has a simple notation for dates relative to the current date and time. The format for these relative dates and times is a positive or negative offset number, and a character representing the units (days, months, minutes, and so on). For example:

+30s	30 seconds from now
+5m	5 minutes from now
+10h	10 hours from now
-1d	1 day ago
+1M	1 month from now
+2y	2 years from now
now	Now (duh)

## Creating a cookie

Let's begin by writing a program that creates a cookie and sends it to the browser. This program has two parts: a form for the user to enter something we'll remember, and a page to display when the cookie is set.

First, the form:

```
#!/usr/bin/perl -w
# cookie-set.cgi - set a cookie
use CGI qw(:standard);
unless (param()) {
    # display form
    print
        header(),
        start_html("Cookie Baker"),
        h1("Cookie Baker"),
        start_form(),
        p("What's your name?", textfield("NAME")),
        submit(),
        end_form(),
        end_html();
# cookie-set.cgi will be continued ...
```

Setting a cookie is a two-part process: first, create it with the `cookie()` function, then pass it to the browser when you send the HTTP header. In the remaining code, `$to_set` holds the cookie we create, and the `-cookie` argument to the `header()` function passes it to the browser.

```
# cookie-set.cgi continues ...
} else {
    # process form and set cookie
    $name = param("NAME");
    $to_set = cookie(-name => "username",
        -value => $name,
        -expires => "+30s",
        -path => "/~gnat/zd",
    );
    print
        header(-cookie => $to_set),
        start_html("Thanks!"),
        h1("Thanks for using the Cookie Baker"),
        p("I set your name to ", b($name),
            " and I will remember this if you visit ",
            a({-href => "cookie-get.cgi"}, "here"),
            " within the next 30 seconds."),
        end_html();
}
```

If we were setting multiple cookies, we'd pass an array reference to the `header()` function:

```
header(-cookie => [$name_cookie, $age_cookie, $city_cookie])
```

## Fetch

Fetching cookies is even easier. Just call the `cookie()` function with the name of the cookie. The function returns the value of the cookie. The CGI.pm module cannot return the other parameters of a cookie, such as domain, path, and expiration time.



```
#!/usr/bin/perl -w
# cookie-get.cgi - fetch the value of a cookie
use CGI qw(:standard);
$name = cookie("username");
print
    header(),
    start_html("Hello $name"),
    h1("Hello " . $name . " | " | "Stranger");
if ($name) {
    print p("See, I remembered your name!");
} else {
    print p("The cookie must have expired.");
}
print end_html();
```

---

## 18.5. Let us sum Up

---

A relational database is a bunch of rectangular tables. Each row of a table is a record about one person or thing; the record contains several pieces of information called *fields*.

### SQL

SQL stands for *Structured Query Language*. It was invented at IBM in the 1970's. It's a language for describing searches and modifications to a relational database. SQL was a huge success, probably because it's incredibly simple and anyone can pick it up in ten minutes. As a result, all the important database systems support it in some fashion or another.

### Perl DBI

Perl's DBI ('Database Interface') module was written by Tim Bunce. DBI is designed to protect you from the details of the vendor libraries. It has a very simple interface for saying what SQL queries you want to make, and for getting the results back. DBI doesn't know how to talk to any particular database, but it does know how to locate and load in DBD ('Database Driver') modules. The

DBD modules have the vendor libraries in them and know how to talk to the real databases; there is one DBD module for every different database.

### ***DBI Architecture***

The DBI architecture is split into two main groups of software: the DBI itself, and the *drivers*. The DBI defines the actual DBI programming interface, routes method calls to the appropriate drivers, and provides various support services to them. Specific drivers are implemented for each different type of database and actually perform the operations on the databases.

Drivers are also called database drivers, or DBDs, after the namespace in which they are declared. For example, Oracle uses DBD::Oracle, Informix uses DBD::Informix, and so on. A useful tip in remembering the DBI architecture is that DBI can stand for DataBase Independent and DBD can stand for DataBase Dependent.

Because DBI uses Perl's object-orientation features, it is extremely simple to initialize DBI for use within your programs. This can be achieved by adding the line:

```
use DBI;
```

to the top of your programs. This line locates and loads the core DBI module. Individual database driver modules are loaded as required, and should generally not be explicitly loaded.

The DBI defines three main types of objects that you may use to interact with databases. These objects are known as *handles*. There are handles for drivers, which the DBI uses to create handles for database connections, which, in turn, can be used to create handles for individual database commands, known as statements.

*Driver handles* represent loaded drivers and are created when the driver is loaded and initialized by the DBI. There is exactly one driver handle per loaded driver. Initially, the driver handle is the only contact the DBI has with the driver, and at this stage, no contact has been made with any database through that driver.

Within the DBI specification, a driver handle is usually referred to as \$drh.

Driver handles should not normally be referenced within your programs. The actual instantiation of driver handles happens ``under the hood" of DBI, typically when `DBI->connect()` is called.

Database handles are the first step towards actually doing work with the database, in that they encapsulate a single connection to a particular database. Prior to executing SQL statements within a database, we must actually *connect* to the database. This is usually achieved through the DBI's `connect()` method:

```
$dbh = DBI->connect( $data_source, ... );
```

Within the DBI specification and sample code, database handles are usually referred to as `$dbh`.

*Statement handles* are the final type of object that DBI defines for database interaction and manipulation. These handles actually encapsulate individual SQL statements to be executed within the database.

Statement handles are *children* of their corresponding database handle. Since statement handles are objects in their own right, data within one statement is protected from tampering or modification by other statement handles.

Within the DBI specification and sample code, statement handles are generally referred to as `$sth`.

When connecting to a database via the DBI, you need to tell the DBI where to find the database to connect to. The DBI requires the data source name to start with the characters `dbi:`, much like a URL begins with `http:`, and then the name of the driver, followed by another colon--for example, `dbi:Oracle:`. Any text that follows is passed to the driver's own `connect()` method to interpret as it sees fit. Most drivers expect either a simple database name or, more often, a set of one or more name/value pairs separated with semicolons. Some common examples are listed later in this section.

DBI offers two useful methods for querying which data sources are available to you for each driver you have installed on your system.

Firstly, you can get a list of all the available drivers installed on your machine by using the `DBI->available_drivers()` method. This returns a list with each element containing the data source prefix of an installed driver, such as `dbi:Informix:`.

Secondly, you can invoke the `DBI->data_sources()` method against one or more of the drivers returned by the `DBI->available_drivers()` method to enumerate which data sources are known to the driver. Calling the `data_sources()` method will actually load the specified driver and validate that it is completely and correctly installed. Because DBI dies if it can't load and initialize a driver, this method should be called inside an `eval{}`  block if you need to catch that error.

In light of these common arguments, the syntax for connecting to databases using DBI is to use the `connect()` call, defined as follows:

```
$dbh = DBI->connect( $data_source, $username, $password, \%attr );
```

Explicit disconnection from the database is not strictly necessary if you are exiting from your program after you have performed all the work, but it is a good idea. We strongly recommend that you get into the habit of disconnecting explicitly.

DBI provides a method through which programmers may disconnect a given database handle from its database. This is good practice, especially in programs in which you have performed multiple connections or will be carrying out multiple sequential connections.

The method for performing disconnections is:

```
$rc = $dbh->disconnect();
```

## **Cookies**

The Web is designed to be stateless. Each document sent from server to browser is a unique transaction. Having sent a document, the server forgets about it, which greatly simplifies programming a Web server. But it doesn't simplify writing Web applications. To implement shopping carts, user preferences, or detailed visitor tracking, you need to recognize a user from one page to the next. By the time the Web needed this level of sophistication, it was

too late to turn around and make it support states. So Netscape invented cookies.

### **Cookie attributes**

Cookies contain a lot of information, the most useful being the name, a string that identifies the cookie, and the value, the data associated with that name.

Setting a cookie is a two-part process: first, create it with the `cookie()` function, then pass it to the browser when you send the HTTP header. In the remaining code, `$to_set` holds the cookie we create, and the `-cookie` argument to the `header()` function passes it to the browser.

Fetching cookies is even easier. Just call the `cookie()` function with the name of the cookie. The function returns the value of the cookie. The `CGI.pm` module cannot return the other parameters of a cookie, such as domain, path, and expiration time.

---

### **18.6. Lesson end Activities**

---

1. What is Perl DBI?
2. Why we need cookies?

---

### **18.7. Check your progress**

---

1. Write a perl program to list out from the database, employees whose salary is more than Rs. 20,000.
2. Write a perl program to set and destroy cookies.

---

### **18.8. Reference**

---

1. Internet & World Wide Web, H.M. Deitel, P.J.Deitel and A.B.Goldberg, Prentice Hall.
2. [www.perl.org](http://www.perl.org)



---

## Lesson 19

# XML

---

### Contents

#### 19.0. Aim and Objectives

#### 19.1. Introduction to XML

#### 19.2. XML Document Type Definition

#### 19.3. XML Parser

#### 19.4. XHTML

#### 19.5. Let us Sum Up

#### 19.6. Lesson end Activities

#### 19.7. Check your progress

#### 19.8. Reference

---

### 19.0 Aim and objectives

---

- To understand XML
- To learn XML to write simple program

---

### 19.1. Introduction to XML

---

XML was designed to transport and store data. HTML was designed to display data. XML stands for EXtensible Markup Language. XML is a markup language much like HTML. XML was designed to carry data, not to display data. XML tags are not predefined. You must define your own tags. XML is designed to be self-descriptive. XML is a W3C Recommendation

#### The Difference Between XML and HTML

XML is not a replacement for HTML. XML and HTML were designed with different goals. XML was designed to transport and store data, with focus on what data is.

HTML was designed to display data, with focus on how data looks. HTML is about displaying information, while XML is about carrying information.

## XML Does not DO Anything

Maybe it is a little hard to understand, but XML does not DO anything. XML was created to structure, store, and transport information.

The following example is a note to Tove from Jani, stored as XML:

```
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The note above is quite self descriptive. It has sender and receiver information, it also has a heading and a message body.

But still, this XML document does not DO anything. It is just pure information wrapped in tags. Someone must write a piece of software to send, receive or display it.

XML is nothing special. It is just plain text. Software that can handle plain text can also handle XML. However, XML-aware applications can handle the XML tags specially. The functional meaning of the tags depends on the nature of the application.

With XML You Invent Your Own Tags. The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document. That is because the XML language has no predefined tags. The tags used in HTML (and the structure of HTML) are predefined. HTML documents can only use tags defined in the HTML standard (like <p>, <h1>, etc.). XML allows the author to define his own tags and his own document structure.

XML is Not a Replacement for HTML. XML is a complement to HTML. It is important to understand that XML is not a replacement for HTML. In most web applications, XML is used to transport data, while HTML is used to format and display the data



---

## 19.2. Introduction to Document Type definition (DTD)

---

The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. A DTD can be declared inline in your XML document, or as an external reference. XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

### **XML – Structuring data**

#### **The building blocks of XML documents**

XML documents (and HTML documents) are made up by the following building blocks:

Elements, Tags, Attributes, Entities, PCDATA, and CDATA

#### **Elements**

Elements are the main building blocks of both XML and HTML documents.

Examples of HTML elements are "body" and "table". Examples of XML elements could be "note" and "message". Elements can contain text, other elements, or be empty. Examples of empty HTML elements are "hr", "br" and "img".

#### **Tags**

Tags are used to markup elements.

A starting tag like `<element_name>` mark up the beginning of an element, and an ending tag like `</element_name>` mark up the end of an element.

Examples:

A body element: `<body>body text in between</body>`.

A message element: `<message>some message in between</message>`

#### **Attributes**

Attributes provide extra information about elements.

Attributes are placed inside the start tag of an element. Attributes come in name/value pairs. The following "img" element has an additional information about a source file:

```

```

The name of the element is "img". The name of the attribute is "src". The value of the attribute is "computer.gif". Since the element itself is empty it is closed by a " /".

## **PCDATA**

PCDATA means parsed character data.

Think of character data as the text found between the start tag and the end tag of an XML element. PCDATA is text that will be parsed by a parser. Tags inside the text will be treated as markup and entities will be expanded.

## **CDATA**

CDATA also means character data.

CDATA is text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded.

## **Entities**

Entities as variables used to define common text. Entity references are references to entities. Most of you will know the HTML entity reference: "&nbsp;" that is used to insert an extra space in an HTML document. Entities are expanded when a document is parsed by an XML parser.

The following entities are predefined in XML:

<b>Entity References</b>	<b>Character</b>
&lt;	<
&gt;	>
&amp;	&

&quot;	"
&apos;	'

## **DTD - Elements**

### **Declaring an Element**

In the DTD, XML elements are declared with an element declaration. An element declaration has the following syntax:

```
<!ELEMENT element-name (element-content)>
```

### **Empty elements**

Empty elements are declared with the keyword EMPTY inside the parentheses:

```
<!ELEMENT element-name (EMPTY)>
```

example:

```
<!ELEMENT img (EMPTY)>
```

### **Elements with data**

Elements with data are declared with the data type inside parentheses:

```
<!ELEMENT element-name (#CDATA)>
```

or

```
<!ELEMENT element-name  
(#PCDATA)>
```

or

```
<!ELEMENT element-name (ANY)>
```

example:

```
<!ELEMENT note (#PCDATA)>
```

#CDATA means the element contains character data that is not supposed to be parsed by a parser.

#PCDATA means that the element contains data that IS going to be parsed by a parser.

The keyword ANY declares an element with any content.

If a #PCDATA section contains elements, these elements must also be declared.

### **Elements with children (sequences)**

Elements with one or more children are defined with the name of the children elements inside the parentheses:

```
<!ELEMENT element-name (child-element-name)>
```

or

```
<!ELEMENT element-name (child-element-name,child-element-  
name,.....)>
```

example:

```
<!ELEMENT note (to,from,heading,body)>
```

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the note document will be:

```
<!ELEMENT note  
(to,from,heading,body)>
```

```
<!ELEMENT to (#CDATA)>
```

```
<!ELEMENT from (#CDATA)>
```

```
<!ELEMENT heading (#CDATA)>
```

```
<!ELEMENT body (#CDATA)>
```

### **Wrapping**

If the DTD is to be included in your XML source file, it should be wrapped in a DOCTYPE definition with the following syntax:

```
<!DOCTYPE      root-element      [element-
declarations]>
```

example:

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to    (#CDATA)>
  <!ELEMENT from  (#CDATA)>
  <!ELEMENT heading (#CDATA)>
  <!ELEMENT body  (#CDATA)>
]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

### **Declaring only one occurrence of the same element**

```
<!ELEMENT  element-name  (child-
name)>
```

example

```
<!ELEMENT note (message)>
```

The example declaration above declares that the child element message can only occur one time inside the note element.

### **Declaring minimum one occurrence of the same element**

```
<!ELEMENT  element-name  (child-
name+)>
```

example

```
<!ELEMENT note (message+)>
```

The + sign in the example above declares that the child element message must occur one or more times inside the note element.

### Declaring zero or more occurrences of the same element

```
<!ELEMENT element-name (child-name*)>
```

example

```
<!ELEMENT note (message*)>
```

The \* sign in the example above declares that the child element message can occur zero or more times inside the note element.

### Declaring zero or one occurrences of the same element

```
<!ELEMENT element-name (child-name?)>
```

example

```
<!ELEMENT note (message?)>
```

The ? sign in the example above declares that the child element message can occur zero or one times inside the note element.

### Declaring mixed content

example

```
<!ELEMENT note (to+,from,header,message*,#PCDATA)>
```

The example above declares that the element note must contain at least one **to** child element, exactly one **from** child element, exactly one **header**, zero or more **message**, and some other parsed **character data** as well. Puh!

### DTD - Attributes

#### Declaring Attributes

In the DTD, XML element attributes are declared with an ATTLIST declaration. An attribute declaration has the following syntax:

```
<!ATTLIST element-name attribute-name attribute-type default-value>
```

As you can see from the syntax above, the ATTLIST declaration defines the element which can have the attribute, the name of the attribute, the type of the attribute, and the default attribute value.

The **attribute-type** can have the following values:

Value	Explanation
CDATA	The value is character data
(eval   eval   ..)	The value must be an enumerated value
ID	The value is an unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is predefined

The **attribute-default-value** can have the following values:

Value	Explanation
#DEFAULT value	The attribute has a default value
#REQUIRED	The attribute value must be included in the element
#IMPLIED	The attribute does not have to be included
#FIXED value	The attribute value is fixed

## Attribute declaration example

DTD example:

```
<!ELEMENT square EMPTY>  
<!ATTLIST square width CDATA "0">
```

XML example:

```
<square width="100"></square>
```

In the above example the element square is defined to be an empty element with the attributes width of type CDATA. The width attribute has a default value of 0.

## Default attribute value

Syntax:

```
<!ATTLIST element-name attribute-name CDATA "default-  
value">
```

DTD example:

```
<!ATTLIST payment type CDATA "check">
```

XML example:

```
<payment type="check">
```

Specifying a default value for an attribute, assures that the attribute will get a value even if the author of the XML document didn't include it.

## Implied attribute

Syntax:

```
<!ATTLIST element-name attribute-name attribute-type  
#IMPLIED>
```

DTD example:

```
<!ATTLIST contact fax CDATA #IMPLIED>
```



XML example:

```
<contact fax="555-667788">
```

Use an implied attribute if you don't want to force the author to include an attribute and you don't have an option for a default value either.

### **Required attribute**

Syntax:

```
<!ATTLIST element-name attribute_name attribute-type  
#REQUIRED>
```

DTD example:

```
<!ATTLIST person number CDATA #REQUIRED>
```

XML example:

```
<person number="5677">
```

Use a required attribute if you don't have an option for a default value, but still want to force the attribute to be present.

### **Fixed attribute value**

Syntax:

```
<!ATTLIST element-name attribute-name attribute-type #FIXED  
"value">
```

DTD example:

```
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```

XML example:

```
<sender company="Microsoft">
```

Use a fixed attribute value when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error.

## Enumerated attribute values

Syntax:

```
<!ATTLIST element-name attribute-name (eval|eval|..) default-value>
```

DTD example:

```
<!ATTLIST payment type (check|cash) "cash">
```

XML example:

```
<payment type="check">
```

or

```
<payment type="cash">
```

Use enumerated attribute values when you want the attribute values to be one of a fixed set of legal values.

## DTD - Entities

### Entities

Entities as variables used to define shortcuts to common text. Entity references are references to entities. Entities can be declared internal. Entities can be declared external .

### Internal Entity Declaration

Syntax:

```
<!ENTITY entity-name "entity-value">
```

DTD Example:

```
<!ENTITY writer "Jan Egil Refsnes.">
```

```
<!ENTITY      copyright      "Copyright  
XML101.">
```

XML example:

```
<author>&writer;&copyright;</author>
```

### External Entity Declaration

Syntax:

```
<!ENTITY entity-name SYSTEM "URI/URL">
```

DTD Example:

```
<!ENTITY          writer          SYSTEM  
"http://www.xml101.com/entities/entities.xml">
```

```
<!ENTITY          copyright        SYSTEM  
"http://www.xml101.com/entities/entities.dtd">
```

XML example:

```
<author>&writer;&copyright;</author>
```

---

### 19.3. XML Parser

---

Most browsers have a build-in XML parser to read and manipulate XML. The parser converts XML into a JavaScript accessible object.

#### Parsing XML

All modern browsers have a build-in XML parser that can be used to read and manipulate XML. The parser reads XML into memory and converts it into an XML DOM object that can be accessed with JavaScript.

There are some differences between Microsoft's XML parser and the parsers used in other browsers. The Microsoft parser supports loading of both XML files and XML strings (text), while other browsers use separate parsers. However, all parsers contain functions to traverse XML trees, access, insert, and delete nodes (elements) and their attributes.

#### Loading XML with Microsoft's XML Parser

Microsoft's XML parser is built into Internet Explorer 5 and higher. The following JavaScript fragment loads an XML document ("note.xml") into the parser:

```
var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
```

```
xmlDoc.async="false";  
xmlDoc.load("note.xml");
```

The first line of the script above creates an empty Microsoft XML document object. The second line turns off asynchronous loading, to make sure that the parser will not continue execution of the script before the document is fully loaded. The third line tells the parser to load an XML document called "note.xml".

The following JavaScript fragment loads a string called txt into the parser:

```
var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");  
xmlDoc.async="false";  
xmlDoc.loadXML(txt);
```

### **XML Parser in Firefox and Other Browsers**

The following JavaScript fragment loads an XML document ("note.xml") into the parser:

```
var xmlDoc=document.implementation.createDocument("", "", null);  
xmlDoc.async="false";  
xmlDoc.load("note.xml");
```

The first line of the script above creates an empty XML document object. The second line turns off asynchronous loading, to make sure that the parser will not continue execution of the script before the document is fully loaded. The third line tells the parser to load an XML document called "note.xml".

The following JavaScript fragment loads a string called txt into the parser:

```
var parser=new DOMParser();  
var doc=parser.parseFromString(txt,"text/xml");
```

The first line of the script above creates an empty XML document object. The second line tells the parser to load a string called txt.

---

## **19.4. XHTML**

---

XHTML is HTML "reformulated" to conform to the current Extensible Markup Language (XML) standard, version 1.0. Imagine taking the best parts from the HTML language and mixing them with all of the great aspects of XML... then you're coming close to imagining the power and flexibility of XHTML.

XHTML has much stricter language syntax than HTML, however. To create fully valid XHTML documents, they must adhere to these rules/guidelines:

### **All tags must be closed**

With normal HTML documents, some browsers will still render the contents of a `<table>` even if you don't close the table with a `</table>` tag. This allows developers to become lazy and forgetful. The tags within an XHTML document must always be nested correctly and closed properly.

If we have the following HTML 4.0 compliant table:

```
<table width="100%">
<tr>
<td>
<p><b>Welcome to my page
</td>
</tr>
</table>
<hr>
```

you can see straight away that the `<p>`, `<b>`, and `<hr>` tags aren't closed. This is a big no-no for XHTML documents and will raise a parser error, because all tags must be closed (yes, even the `<p>` tag).

The XHTML 1.0 compliant version of the table shown above looks like this:

```
<table width="100%">
<tr>
<td>
<p><b>Welcome to my page</b></p>
```

```
</td>
</tr>
</table>
<hr />
```

Notice how the `<p>`, `<b>`, and `<hr>` tags are now closed? To close tags like `<hr>`, we can simply add a space and forward-slash within the tag, like this: `<hr />`.

### **Attributes must contain quoted values**

All tag attributes, such as `<p align="center">` must be enclosed within double quotes. You no longer have the choice of either single or double quotes. Also, for attributes which have no value, or aren't quoted such as

```
<option checked>1</option>
```

you must assign a value to that attribute (even though it won't be used), and surround it in double quotes, for example:

```
<option checked="checked">1</option>
```

All element and attribute names must also be lower case.

### **Be careful with special characters**

Because of the way XHTML documents are validated and must conform to specific rules, HTML comments like this:

```
<!-- This is a comment -->
```

as well as inline style sheets and inline JavaScript should always be removed from your XHTML document. You should store them in separate `.css` and `.js` files respectively, and reference them like this:

```
<link rel="stylesheet" type="text/css" href="mystyle.css" />
```

for style sheets, or:

```
<script language="JavaScript" src="mystuff.js"></script>
```

for JavaScript files.

If you're using other HTML characters such as `<`, `>` and `&` in attribute values, for example, then they should be replaced with their corresponding HTML entity representations such as `"<"`, `">"` and `"&"` respectively.

Last but not least, **with the exception of** form input elements such as `<input>`, `<select>`, you should use the "id" attribute instead of "name" when attaching attributes to an element. In XHTML documents, the "name" attribute is rendered useless (again, apart from form elements), and belongs back with HTML 4.0.

### **Document Type Definition(DTD)**

XHTML documents have three parts: the DOCTYPE (which contains the DTD declaration), the head and the body. To create web pages that properly conform to the XHTML 1.0 standard, each page must include a DTD declaration; either strict, transitional, or frameset. Each of the three DTD's is described (with an example) below:

#### **Strict**

You should use the strict DTD when your XHTML pages will be marked up cleanly, free of presentational clutter. You use the strict DTD together with cascading style sheets, because it doesn't allow attributes like "bgcolor" to be set for the `<body>` tag, etc.

The strict DTD looks like this:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

#### **Transitional**

The transitional DTD should be used when you need to take advantage of the presentational features that are available through HTML. You should also use the transitional DTD when you want to support older browsers that don't have built-in support for cascading style sheets.

The transitional DTD looks like this:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

## Frameset

You should use the frameset DTD when your XHTML page will contain frames. The frameset DTD looks like this:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

It should be fairly obvious which DTD declaration to include in your XHTML file simply by just reading the features of each one. Now that we've got all of the formal details of XHTML out of the way, let's look at some examples of XHTML.

## XHTML Benefits

XHTML documents are backward compatible with older, non-XHTML compliant web browsers. Instead of sloppy HTML tags, your pages will now contain XML tags that are always properly closed and nested correctly, such as:

```
<p><b><u>This is some text<br /><br /></u></b></p>
```

Instead of:

```
<p><b><u>This is some text<br><br></b></u>
```

You can see that the XHTML version of the code above has a `<p>`, `<b>`, and then a `<u>` tag. These tags are closed in the reverse order that they were created in: `</u>`, `</b>`, and lastly `</p>`. All tags must be closed in this way for the XHTML document to be considered valid.

Also, notice the `<br />` tags? Because of the way non-XHTML browsers are designed, as long as you leave a space between the beginning of the tag and the `/>`, then they will just treat the tag normally, and ignore the forward-slash.

## XHTML is a standard

XHTML is an accepted standard (see <http://www.w3.org/TR/xhtml1/>), meaning that all of the newer browsers due for release in the future (such as newer version of IE, Netscape and Opera) will most definitely contain built-in support for XHTML.



The first document type in the XHTML family is XHTML 1.0. The W3C standard for XHTML takes three of the previous document types from HTML 4.0 and converts them to fully utilize XML wherever possible. This promotes a consistent, logical layout, while still keeping the actual content easy to follow.

XHTML documents must incorporate one of three document type definitions (DTD's). This makes sure that the XML data contained within an XHTML document is valid and conforms to a certain layout/logical style as defined in that DTD file.

You might be wondering what the benefits of migrating your current HTML documents over to XHTML are; you may also be wondering why you should bother learning more about XHTML; well, let me point out some of the benefits of XHTML (as listed at <http://www.w3.org/TR/xhtml1/>):

- XHTML documents are XML conforming. As such, they are readily reviewed, edited and validated with standard XML tools such as the MSXML parser.
- XHTML documents can be written to operate as well or better than they did before in existing HTML 4-conforming user agents as well as in new, XHTML 1.0 conforming user agents.
- XHTML documents can utilize applications (e.g. scripts and applets) that rely upon either the HTML Document Object Model or the XML Document Object Model (DOM)

### **XHTML Examples**

As you can probably guess by now, XHTML code looks very similar to plain old HTML code, with just a couple of syntactic differences. Three examples of valid XHTML documents are shown below. They were validated using the W3C's XHTML validation tool, located at <http://validator.w3.org/>.

#### **Example 1:**

This example used the strict DTD, meaning that every single tag must be closed properly, all attributes assigned values, etc:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE html
```

```

PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml11-strict.dtd">
<html      xmlns="http://www.w3.org/1999/xhtml"      xml:lang="en"
lang="en">
<head>
<title> Strict DTD XHTML Example </title>
</head>
<body>
<p>
Please Choose a Day:
<br /><br />
<select name="day">
<option selected="selected">Monday</option>
<option>Tuesday</option>
<option>Wednesday</option>
</select>
</p>
</body>
</html>

```

### **Example 2:**

This example uses the transitional DTD, which provides support for older browsers that don't recognize style sheets. You can see it uses several attributes within the <body> tag, which aren't allowed when using the strict DTD:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"DTD/xhtml11-transitional.dtd">
<html      xmlns="http://www.w3.org/1999/xhtml"      xml:lang="en"
lang="en">

```

```

<head>

<title> Transitional DTD XHTML Example </title>

</head>

<body bgcolor="#FFFFFF" link="#000000" text="red">

<p>This is a transitional XHTML example</p>

</body>

</html>

```

### **Example 3:**

This example uses the frameset DTD, which allows us to split one XHTML page into multiple frames, with each frame containing an XHTML page within it:

```

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE html

PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"

"DTD/xhtml11-frameset.dtd">

<html      xmlns="http://www.w3.org/1999/xhtml"      xml:lang="en"
lang="en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1" />

<title> Frameset DTD XHTML Example </title>

</head>

<frameset cols="100,*">

<frame src="toc.html" />

<frame src="intro.html" name="content" />

</frameset>

</html>

```

---

## **19.5. Let Us Sum UP**

---

XML was designed to transport and store data. HTML was designed to display data. XML stands for EXtensible Markup Language. XML is a markup

language much like HTML. XML was designed to carry data, not to display data. XML tags are not predefined. You must define your own tags. XML is designed to be self-descriptive. XML is a W3C Recommendation

### **Document Type definition (DTD)**

The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. A DTD can be declared inline in your XML document, or as an external reference. XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

XML documents (and HTML documents) are made up by the following building blocks:

Elements, Tags, Attributes, Entities, PCDATA, and CDATA

### **Elements**

Elements are the main building blocks of both XML and HTML documents.

### **Tags**

Tags are used to markup elements.

A starting tag like `<element_name>` mark up the beginning of an element, and an ending tag like `</element_name>` mark up the end of an element.

### **Attributes**

Attributes provide extra information about elements. Attributes are placed inside the start tag of an element. Attributes come in name/value pairs.

### **PCDATA**

PCDATA means parsed character data.

### **CDATA**

CDATA also means character data. CDATA is text that will NOT be parsed by a parser.

Tags inside the text will NOT be treated as markup and entities will not be expanded.

## **Entities**

Entities as variables used to define common text. Entity references are references to entities. Most of you will know the HTML entity reference: "&nbsp;" that is used to insert an extra space in an HTML document. Entities are expanded when a document is parsed by an XML parser.

## **Declaring an Element**

In the DTD, XML elements are declared with an element declaration.

## **Empty elements**

Empty elements are declared with the keyword EMPTY inside the parentheses:

## **Elements with data**

Elements with data are declared with the data type inside parentheses:

## **Elements with children (sequences)**

Elements with one or more children are defined with the name of the children elements inside the parentheses:

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children.

## **Wrapping**

If the DTD is to be included in your XML source file, it should be wrapped in a DOCTYPE definition.

## **DTD - Attributes**

### **Declaring Attributes**

In the DTD, XML element attributes are declared with an ATTLIST declaration. An attribute declaration has the following syntax:

## **DTD - Entities**

### **Entities**

Entities as variables used to define shortcuts to common text. Entity references are references to entities. Entities can be declared internal. Entities can be declared external .

### **XML Parser**

Most browsers have a build-in XML parser to read and manipulate XML. The parser converts XML into a JavaScript accessible object.

### **Parsing XML**

All modern browsers have a build-in XML parser that can be used to read and manipulate XML. The parser reads XML into memory and converts it into an XML DOM object that can be accessed with JavaScript.

There are some differences between Microsoft's XML parser and the parsers used in other browsers. The Microsoft parser supports loading of both XML files and XML strings (text), while other browsers use separate parsers. However, all parsers contain functions to traverse XML trees, access, insert, and delete nodes (elements) and their attributes.

### **XHTML**

XHTML is HTML "reformulated" to conform to the current Extensible Markup Language (XML) standard, version 1.0. Imagine taking the best parts from the HTML language and mixing them with all of the great aspects of XML... then you're coming close to imagining the power and flexibility of XHTML.

XHTML has much stricter language syntax than HTML, however. To create fully valid XHTML documents, they must adhere to these rules/guidelines:

All tags must be closed

Attributes must contain quoted values

## **Document Type Definition(DTD)**

XHTML documents have three parts: the DOCTYPE (which contains the DTD declaration), the head and the body. To create web pages that properly conform to the XHTML 1.0 standard, each page must include a DTD declaration; either strict, transitional, or frameset. Each of the three DTD's is described (with an example) below:

### **Strict**

You should use the strict DTD when your XHTML pages will be marked up cleanly, free of presentational clutter. You use the strict DTD together with cascading style sheets, because it doesn't allow attributes like "bgcolor" to be set for the <body> tag, etc.

### **Transitional**

The transitional DTD should be used when you need to take advantage of the presentational features that are available through HTML. You should also use the transitional DTD when you want to support older browsers that don't have built-in support for cascading style sheets.

### **Frameset**

You should use the frameset DTD when your XHTML page will contain frames. The frameset DTD looks like this:

### **XHTML Benefits**

XHTML documents are backward compatible with older, non-XHTML compliant web browsers. Instead of sloppy HTML tags, your pages will now contain XML tags that are always properly closed and nested correctly, such as:

You can see that the XHTML version of the code above has a <p>, <b>, and then a <u> tag. These tags are closed in the reverse order that they were created in: </u>, </b>, and lastly </p>. All tags must be closed in this way for the XHTML document to be considered valid.

Also, notice the <br /> tags? Because of the way non-XHTML browsers are designed, as long as you leave a space between the beginning of the tag and the ">", then they will just treat the tag normally, and ignore the forward-slash.

## **XHTML is a standard**

XHTML is an accepted standard (see <http://www.w3.org/TR/xhtml1/>), meaning that all of the newer browsers due for release in the future (such as newer version of IE, Netscape and Opera) will most definitely contain built-in support for XHTML.

The first document type in the XHTML family is XHTML 1.0. The W3C standard for XHTML takes three of the previous document types from HTML 4.0 and converts them to fully utilize XML wherever possible. This promotes a consistent, logical layout, while still keeping the actual content easy to follow.

XHTML documents must incorporate one of three document type definitions (DTD's). This makes sure that the XML data contained within an XHTML document is valid and conforms to a certain layout/logical style as defined in that DTD file.

---

### **19.6. Lesson end Activities**

---

1. What is need for XML?
2. What the elements of XML DTD?
3. Compare HTML and XHTML.

---

### **19.7. Check your Progress**

---

1. Describe XML DTD.
2. Write a XML program for you mark sheet where you pass or fail in a subject.
3. Explain about XHTML.

---

### **19.8. Reference**

---

1. Williamson, "The Complete reference XML", Tata McGraw Hill, 2005
2. Thomas A.Powell, " The Complete Reference HTML and XHTML", 4<sup>th</sup> Edition, Tata McGraw Hill