Web Technologies

UNIT-I

SYLLABUS

- Basic Syntax
- Standard HTML Document Structure
- Basic Text Markup
- Images
- Hypertext Links
- Lists
- Tables
- Forms
- HTML5 CSS:
- Levels of Style Sheets
- Style Specification Formats
- Selector Forms
- The Box Model
- Conflict Resolution

Introduction to HTML:-

HTML means Hypertext Markup Language. In 1960 Ted Nelson introduced Hypertext. HTML is a scripting language which is used to create web pages. If you are thinking of creating your own web pages, you need to know at least basic HTML. These HTML documents are plain text files; user can create these documents using text editor like Notepad or Edit.

HTML is a hypertext Language because it supports font styled text, pictures, graphics and animations and also it provides hyper links that used to browse the Internet easily. Text becomes hypertext with the addition of links that connects other hypertext documents. Hypertext is a text augmented with links-pointers to other pieces of text, possible elsewhere in the same document (internal linking) or in another document (external linking).

Rules to write HTML Code:-

- Every HTML document begins with start tag is <HTML> terminates with an ending tag is </HTML>
- HTML documents should be saved with the extension .html or .htm.
- A tag is made up of left operator(<), a right operator(>) and a tag name between these two operators.
- If you forget to mention the right operator(>) or if you give any space between left operator and tag name browser will not consider it as tag.
- At the same time if browser not understands the tag name it just ignores it, browser

won't generate any errors.

- HTML language is not case sensitive, hence user can write the code in either upper case or lower case. No difference between <HTML> and <html>

Syntax of a tag:

<Tag name [parameters=value]>

Ex: HR is a tag name that displays a horizontal ruler line.

<HR>----------(No parameters, no value)

<HR ALIGN=CENTER> ---------- (Tag with parameter and value for the parameter)

<HR WIDTH="30%" SIZE=100 ALIGN=RIGHT> ----- (Tag with more parameters with their values)

Different types of Tags:

1. Singleton tags
2. Paired tags

Singleton tag does not require an ending tag. (Ex: <HR>

Paired tag required an ending tag, which is similar to opening tag except backslash before the tag name (Ex: <HTML> is opening tag, then ending tag is </HTML>

Comments in HTML:

An HTML comment begins with "<!--" and ends with "-->".There should not be a space between angular bracket and exclamation mark.

Creating HTML Page:

The Following steps are needed to create a HTML page

Step 1: Open any text editor like Notepad, Edit, Word etc.

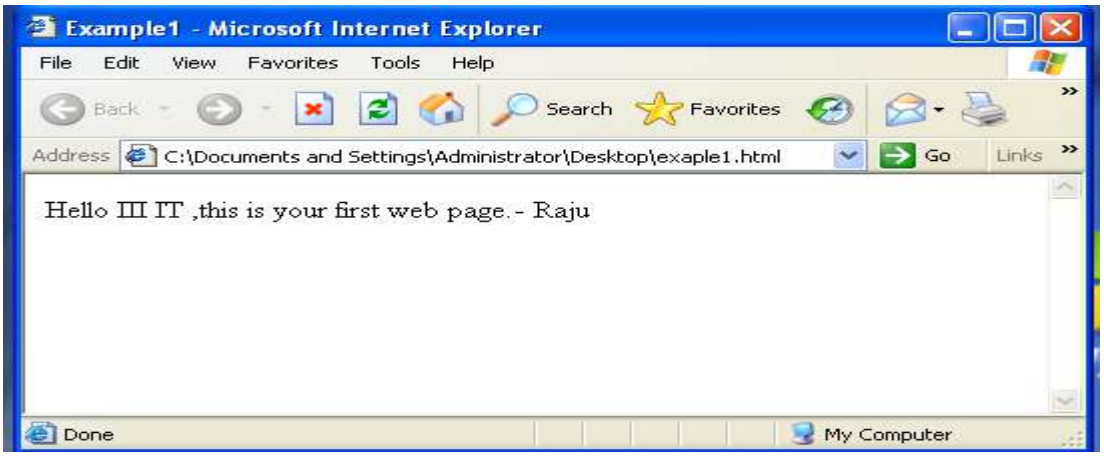
Step 2: Use the file menu to create a new document (File→ New) and type the following code

```
<HTML>
<HEAD>
<TITLE>Example1 </TITLE>
<BODY>
    Hello III-II IT-A SECTION , this is your first web page
</BODY>
</HTML>
```

Step 3: Go to the file menu and choose save as option (File->save as) and give the name of the file as "example1.html" under root directory(C:)(or any valid path)

Step 4: After saving, an internet explorer icon will be displayed as shown below

Step 5: Double click to execute it. The output displayed following



Basic HTML tags

1. Body tag:-

Body tag contain some attributes such as bgcolor, background etc. bgcolor is used for background color, which takes background color name or hexadecimal number and #FFFFFF and background attribute will take the path of the image which you can place as the background image in the browser.

<body bgcolor="#F2F3F4" background= "c:\amer\imag1.gif">

2. Paragraph tag:-

Most text is part of a paragraph of information. Each paragraph is aligned to the left, right or center of the page by using an attribute called as align.

<p align="left" | "right" | "center">

3. Heading tag:-

HTML is having six levels of heading that are commonly used. The largest heading tag is <h1>. The different levels of heading tag besides <h1> are <h2>,<h3>, <h4>, <h5> and <h6>. These heading tags also contain attribute called as align.

<h1 align="left" | "right" | "center">..... <h2>

4. hr tag:-

This tag places a horizontal line across the system. These lines are used to break the page. This tag also contains attribute i.e., width which draws the horizontal line with the screen size of the browser. This tag does not require an end tag.

<hr width="50%">.

5. base font:-

This specify format for the basic text but not the headings.

<basefont size="10">

6. font tag:-

This sets font size, color and relative values for a particular text.

<font size=”10” color=”#f1f2f3”>

7. bold tag:-
This tag is used for implement bold effect on the text

 .....

8. Italic tag:-
This implements italic effects on the text.

….....

9. strong tag:-
This tag is used to always emphasized the text

 .....

10. tt tag:-
This tag is used to give typewriting effect on the text
<tt>..... </tt>

11. sub and sup tag:-
These tags are used for subscript and superscript effects on the text.
_{.....}
^{.....}

12. Break tag:-
This tag is used to the break the line and start from the next line.

13. &amp &lt &gt   &quot
These are character escape sequence which are required if you want to display characters that HTML uses as control sequences.
Example: < can be represented as &lt.

14. Anchor tag:-
This tag is used to link two HTML pages, this is represented by <a>
 some text
href is an attribute which is used for giving the path of a file which you want to link.

Example 1: HTML code to implement common tags.
mypage.html

```
<html>
<head> <! -- This page implements common html tags -->
<title> My Home page </title>
</head>
<body >
<h1 align="center"> SIR C.R.REDDY COLLGE OF ENGINEERING ,ELURU</h1>
<h2 align="center">ELURU</h2>
<basefont size=4>
<p> This college runs under the <tt>management</tt> of <font size=5> <b><i>&quot
```

```
CRRE&amp;quot &amp;amp </i></b></font><br>
  it is affiliated to <strong> JNTUK</strong>
<hr size=5 width=80%>
<h3> <u>&amp;lt Some common tags &amp;gt</u> </h3><br>
</body>
</html>
```

Text Styles or Cosmetic tags:- HTML provides a numerous range of tags for formatting the text. If you want to format the text with different styles, just you include these tags one by one before text.

<B>……</B>	Bold Text
<U>……...</U>	Underline Text
<I>……..</I>	Displays as Italics
<EM>……</EM>	For Emphasis (New Standard for Italics)
<STRONG>……</STRONG>	Strong or Bold text (New Standard for Bold)
<S>…….</S> or <DEL>…….</DEL>	Strikes the text
<SAMP>……</SAMP>	Code sample text
<VAR>……..</VAR>	Small fonts, fixed width
<ADDRESS>……..</ADDRESS>	Like address model (Looks like italics)
<PRE>……..</PRE>	Considers spaces, new lines etc. As it is prints the information

Scrolling Text Tag:-

<marquee> .....</marquee> Displays scrolling text in a marquee style.

Marquee tag attributes:-

a) align: sets the alignment of the text relative to the marquee. Set to top(default), middle or bottom.
b) behaviour: Sets how the text in the marquee should move, It can be scroll(default), slide(text enters from one side and stops at the other end), or alternative(text seems to bounce from one side to other)
c) bgcolor: sets the background color for the marquee box
d) direction: sets the direction of the text for scrolling. It can be left(default), right, down or up.

Example:-

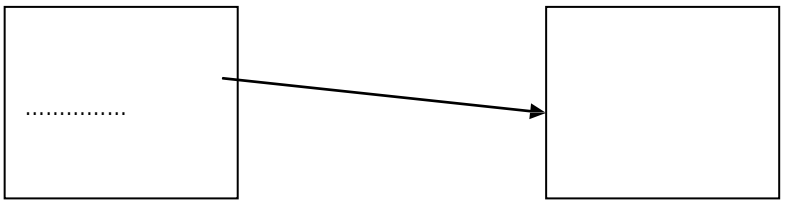
```
<marquee align=”middle” behavior=”slide” bgcolor=”red” direction=”down”>I.T</marquee>
```

Blinking text Tag:-

<blink>……</blank> displays enclosed text as blinking on and off approximately once a second.

Linking in HTML:-

Text becomes hypertext with the addition of links which connect separate locations with in a collection of hypertext documents.



Hyperlinks can be applied for either text or images. Links may connect several web pages of a web site. Links can connect web pages on the same or different servers.

Navigation between pages became easier because of links. Information in the same page also connected through links(internal links).

<A> ..... </A>

Anchor tag is used for creating links. Minimum it requires a parameter i.e., HREF, which indicates the destination document. Other parameters Name and target can be useful for identification for anchor tag and the location of a frame where target page is to be displayed respectively. Name and target tag are optional.

Syntax: <A HREF=”address” NAME=”id“ TARGET=”target window”>

…….

…….. id=Text that displays as link

</A>

HREF Parameter:-

If HREF is included, the text between the opening and closing anchor element that between <A> and </A> becomes hyper text. If users clicks on this text, they are moved to specified document.

Ex:-

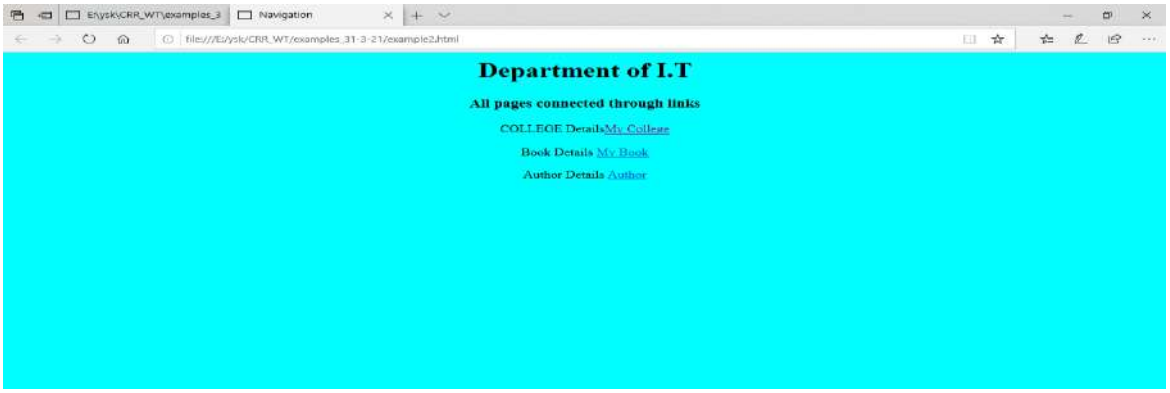
<A HREF=www.vaneshdoc.com>Publishers</A>

Result displayed as Publishers When the user click on this text, Publishers web site is displayed on the browser.

Example :

Create a HTML web page that connect web pages created through hyperlinks.

```
<html>
<head>
<title> Navigation </title>
</head>
<body bgcolor=cyan>
<h1 align=center>Overceas Publisher </h1>
<h3 align=center>All pages connected through links</h3>
<center>
<p>Company Details<A HREF=”1.html”>My Company</A>
<p>Book Details <A HREF=”2.html”>My Book</A>
<p> Author Details <A HREF=”3.html”> Author</A>
</center>
</body>
</html>
```



Color and Image:

Color can be used for background, elements and links. To change the color of links or of the page background hexadecimal values are placed in the <body> tag.

<body bgcolor = "#nnnnnn" text = "#nnnnnn" link= "#nnnnnn" vlink= "#nnnnnn" alink = "#nnnnnn">

The vlink attribute sets the color of links visited recently, alink the color of a currently active link. The six figure hexadecimal values must be enclosed in double quotes and preceded by a hash(#).

Images are one of the aspect of web pages. Loading of images is a slow process, and if too many images are used, then download time becomes intolerable. Browsers display a limited range of image types.

<body background = "URL">

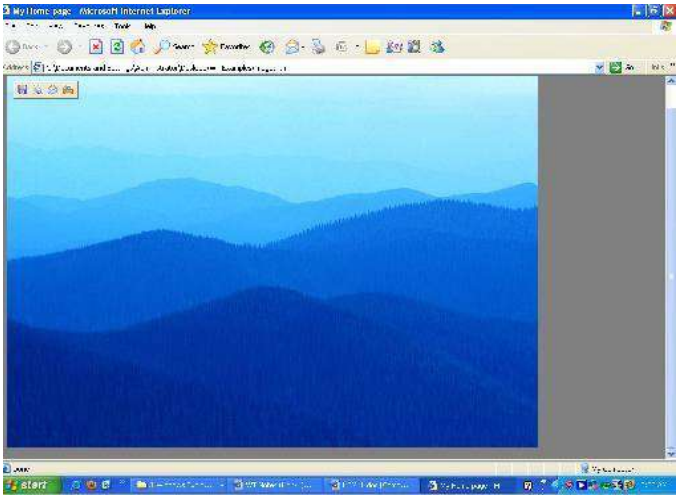
This tag will set a background image present in the URL.

Another tag that displays the image in the web page, which appears in the body of the text rather than on the whole page is given below

<img src="URL" height="n" width="n" align = "top" | "center" | "bottom" >

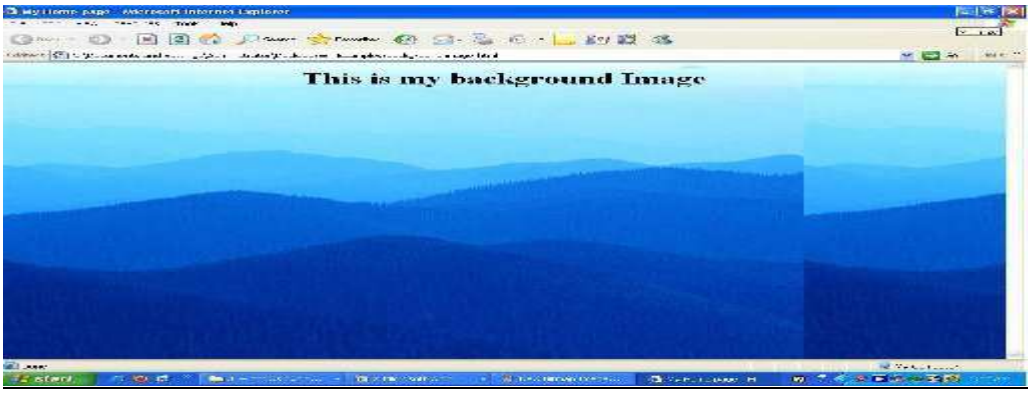
Example 4: HTML code that implements color and image

```
<html>
<head> <! -- This page implements color and image -->
<title> My Home page </title>
</head>
<body bgcolor="gray" text="magneta" vlink="yellow" alink="brown">
<img src= " C:\Documents and Settings\All Users\Documents\My
Pictures\Sample Pictures\Blue hills.jpg">
</body>
</html>
```



Example 5: HTML code that implements background image

```
<html>
<head> <! -- This page implements background image -->
<title> My Home page </title>
</head>
<body background="C:\Documents and Settings\All Users\Documents\My Pictures\Sample
Pictures\Blue hills.jpg">
<h1 align="center"> This is my background Image</h1>
</body>
</html>
```



Lists:

One of the most effective ways of structuring a web site is to use lists. Lists provides straight forward index in the web site. HTML provides three types of list i.e., bulleted list, numbered list and a definition list. Lists can be easily embedded easily in another list to provide a complex but readable structures. The different tags used in lists are explained below.

Unordered Lists:-

Unordered lists are also called unnumbered .lists. The Unordered list elements are used to represent a list of items, which are typically separated by white space and/or marked by bullets. Using <UL> tag does creation of unordered lists in HTML. Which is paired tag, so it requires ending tag that is </UL>. The list of items are included in between <UL>…..</UL>. The TYPE attribute can also be added to the <UL> tag that indicates the displayed bullet along with list of item is square, disc or circle. By default it is disc.

Syntax:-

```
<UL [TYPE={square/disc/circle}]>
    <LI>item name1
    <LI>item name2
    ……..
    - - - - - - - - - - - - - - -
    <LI>item namen
</UL>
```

Example:

Write a HTML program for displaying names of B.Tech Courses with default bullets and names of PG Courses with square bullets.

```
<html>
    <head>
        <title>Unordered Lists</title>
    </head>
<body bgcolor=”tan”>

    <h1>B.Tech Courses
        <h3>
<ul>
    <li>CSE
    <li>IT
    <li>ECE
    <li>EEE
    <li>MECH
```

```
</ul>
      </h3>
<h1>PG Courses
       <h3>
            <ul type=”square”>
                    <li>MCA
                    <li type=”circle”>MBA
                   <li>M.Tech
            </ul>
      </h3>
  </body>
</html>
```

Ordered Lists:-

Ordered lists are also called sequenced or **numbered lists**. In the ordered list the list of item have an order that is signified by numbers, hence it some times called as number lists. Elements used to present a list of items, which are typically separated by white space and/or marked by numbers or alphabets. An orders list should start with the <OL> element, which is immediately followed by a <LI> element which is same as <LI> in unordered list. End of ordered lists is specified with ending tag </OL>.

Different Ordered list types like roman numeral list, alphabet list etc. can be specified with TYPE tag. Another optional parameter with <OL> tag is START attribute, which indicates the starting number or alphabet of the ordered list. For example TYPE=”A” and START=5 will give list start with letter E. The TYPE attribute used in <LI>, changes the list type for particular item. To give more flexibility to list, we can use VALUE parameter with <LI>tag that helps us to change the count for the list item and subsequence items.

Syntax:-

```
<OL [type={“1” or “I” or “a” or”A” or “i”}] START=n>
<LI>item name1
<LI>item name2
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
<LI>item namen
</OL>
```

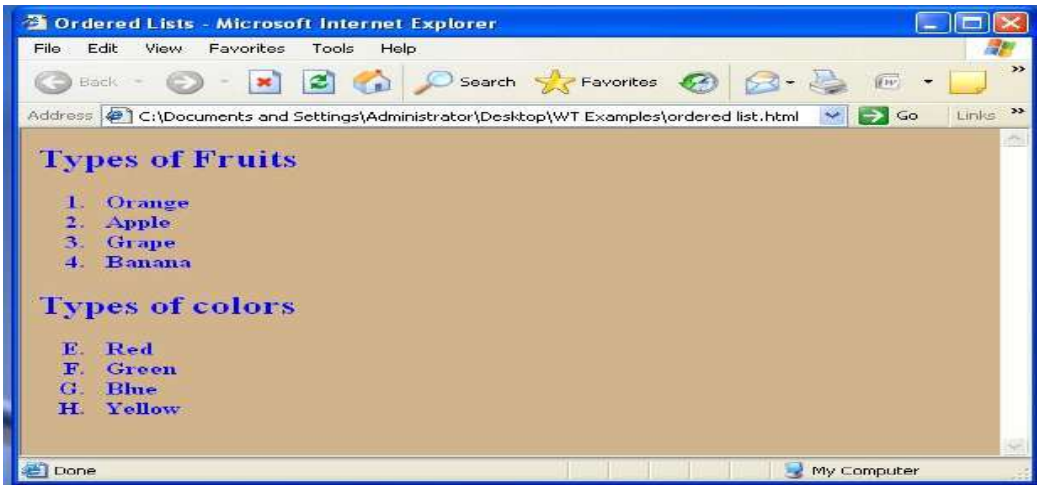
Different Ordered list types

Type=”1” (default) e.g.1,2,3,4…..

Type=”A” Capital letters e.g.A,B,C…
Type=”a” Small letters e.g. a,b,c……
Type=”I” Large roman letters e.g. I,
II, III,…

Example:-

```
<html>
<head>
<title>Ordered Lists</title>
</head>
<body bgcolor=”tan” text=”blue”>
<h2> Types of Fruits
<h4>
<OL type=”A” START=5>
<LI>Red
<LI>Green
<LI>Blue
<LI>Yellow
</OL>
</h4>
<h2>Types of colors
<h4>
        <OL type=”A” START=5>
                <LI>Red
                <LI>Green
                <LI>Blue
                <LI>Yellow
        </OL>
</h4>
</body>
</html>
```



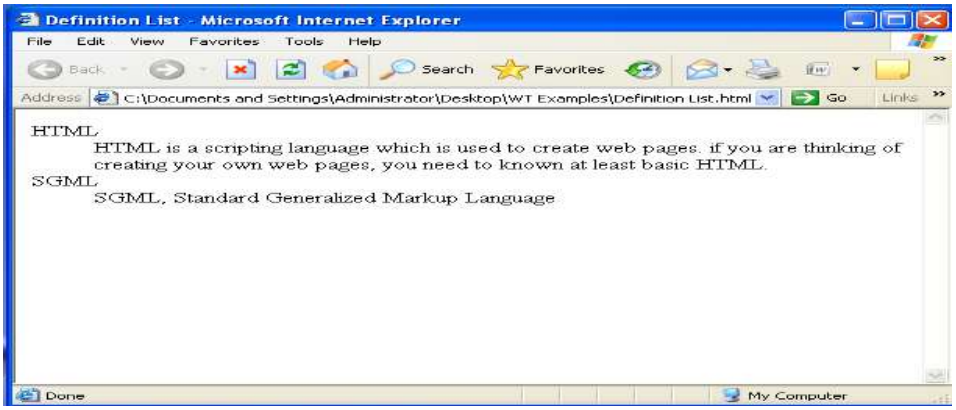
Other Lists:-

There are several lists in HTML, some of them are definition list and Directory List.
Definition List:- <DL>……</DL>A Definition list is a list of definition terms <DT> and corresponding Definition Description<DD> on a new line. To create a definition it must start with <DL> and immediately followed by the first definition term <DT>

Example:-

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

```
        <title>Definition List</title>
    <body>
        <DL>
        <DT>HTML
        <DD> HTML is a scripting language which is used to create web pages. if you
are thinking of creating your own web pages, you need to known at least basic HTML.
        <DT>SGML
        <DD>SGML, Standard Generalized Markup Language
        </DL>
    </body>
</html>
```



Directory List:-

A Directory list element is used to present a list of items containing up to 20 characters each. Items in a Directory List may be arranged in columns, typically 24 characters wide. A Directory List being with <DIR>element,which is immediately followed by a <LI>element. This tag is a deprecated tag, so it is not preferable to use. Hence, use <UL> instead of <DIR>

```
    Other information
        <DIR>
            <LI>Contacts-2043240
            <LI>Business-4123412
            <LI>Personal-3123122
        </DIR>
```

Nested Lists:- Lists can be nested that is Nested Lists is list with in another list.

Tables:

Table is one of the most useful HTML constructs. Tables are find all over the web application. The main use of table is that they are used to structure the pieces of information and to structure the whole web page. Below are some of the tags used in
table.

```
<table align="center" | "left" | "right"  border="n" width="n%" cellpadding="n"
    cellspacing="n">
    ……
    </table>
```

Every thing that we write between these two tags will be within a table. The attributes of the table will control in formatting of the table. Cell padding determines how much space there is between the contents of a cell and its border, cell spacing sets the amount of white space between cells. Width attribute sets the amount of screen that table will use.

```
    <tr> ….. </tr>
```

This is the sub tag of <table> tag, each row of the table has to be delimited by these tags.

<th>……</th>

This is again a sub tag of the <tr> tag. This tag is used to show the table heading .

<td>…..</td>

This tag is used to give the content of the table.

Example 3: HTML code showing the use of table tag

```
<html>
<head>
<title> table page</title>
</head>
<body>
<table align="center" cellpadding="2" cellspacing="2" border="2">
<caption> Time Table for III year IT </caption>
<tr><th> 1 period </th>
<th> 2 peiord </th>
<th> 3 peiord </th>
<th> 4 peiord </th>
</tr>
<tr>
<td> wt </td>
<td> uml</td>
<td> MT</td>
<td> DMDW</td>
</tr>
</table>
</body>
</html>
```

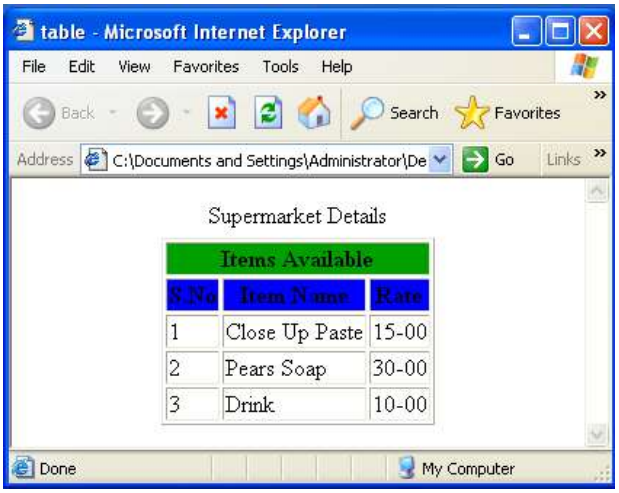
Complex HTML Tables and Formatting:-

You can add background color and background images by using bgcolor and background attributes respectively. Spanning of cells is possible that is you can merge some sequence of rows or columns with the help of ROWSPAN or COLSPAN attributes respectively. For example <th COLSPAN=”2”>widened to span two cells. VALIGN attribute is used for vertical alignment formats and it accepts the values “top”, “middle”, “bottom” and “baseline”.

Example:

```
<html>
    <head>
        <title> table</title>
    </head>
    <body>
```

```
                    <center>
                        <table border="2">
                        <caption>Supermarket Details</caption>
                        <tr>
                        <th colspan=3 bgcolor="tan" align="center">Items
Available</th>
        </tr>
        <tr><th>S.No<th>Item Name<th>Rate</tr>
        <tr><td>1<td>Close Up Paste<td>15-00</tr>
        <tr><td>2<td>Pears Soap<td>30-00</tr>
        <tr><td>3<td>Drink<td>10-00</tr>
        </table>
</center>
            </body>
        </html>
```

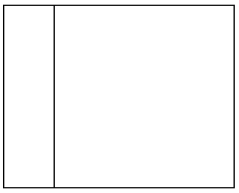


Frames:

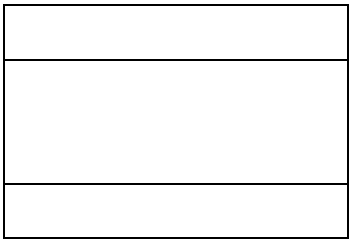
Frames provide a pleasing interface which makes your web site easy to navigate. When we talk about frames actually we are referring to frameset, which is a special type of web page.

Simply frameset is nothing but collection of frames. Web page that contains frame element is called framed page. Framed page begins with <frameset> tag and ends with </frameset>. Each individual frame is identified through <frame> tag. Creation of framed page is very simple. You can nest the framesets. First you decide how you want to divide your webpage and accordingly define frame elements.

Consider the following diagrams, first form divides into two columns and the second form divides into three rows.



Two columns frameset



Three rows frameset

In order to divide into two columns we can use the following syntax

<frameset cols="25%,75%>

<frame name=”disp” src=”1.html”>
<frame name=”res” src=”2.html”>
</frameset>

In the second diagram we have three rows so by using rows parameter of frameset, we can divide logically the window into three rows.

<frameset rows=”20%,*,10%>
<frame name=”first” src=”1.html”> <frame name=”second” src=”2.html”>
<frame name=”third” src=”3.html”>
</frameset>

According to above code ,first row occupies 20% of the window, third row occupies 10% of the window, second row * represents remaining area that is 70% of the window.

Nested Framesets:-

Sometimes it is required to divide your window into rows and columns, then there is requirement of nested framesets. Frameset with in another frameset is known as nested frameset.

The purpose of NAME parameter in frame tag in the above example is nothing but main importance is if we have some links in left side and you want to display respective pages in the right side frame, then name is essential. Using target parameter of Anchor(A) tag as follows users can specify name of frame.

Example:

Frame.html

```
<frameset rows=”20%,*”>
    <frame name=”fr1” src=”frame1.html”>
    <frameset cols=”25%,*”>
        <frame name=”fr2” src=”frame2.html”>
        <frame name=”fr3” src=”frame3.html”>
    </frameset>
</frameset>
```

Frame1.html

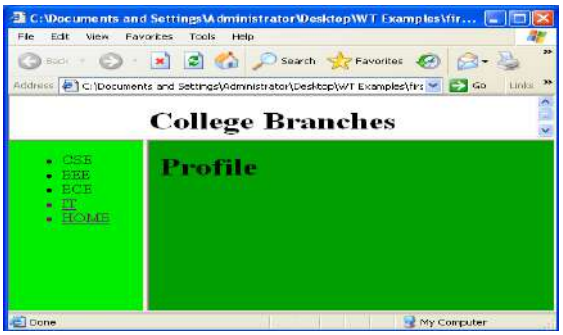
```
<html>
    <body>
        <center><h1> College branches</h1></center>
    </body>
</html>
```

Frame2.html

```
<html>
    <body bgcolor=”green”>
        <ul>
<li>CSE
<li>EEE
<li>ECE
<A href=”example2.html” target=”fr3”><li>IT</A>

        </ul>
    </body>
</html>
```

Frame3.html

```
<html>
    <body text=”white” bgcolor=”tan”>
    <h1>Profile</h1>
    </body>
</html>
```



Forms:

Forms are the best way of adding interactivity of element in a web page. They are usually used to let the user to send information back to the server but can also be used to simplify navigation on complex web sites. The tags that use to implement forms are as follows.

<forms action=”URL” method = “post” | “get”>…….</form>

When get is used, the data is included as part of the URL. The post method encodes the data within the body of the message. Post can be used to send large amount of data, and it is more secure than get. The tags used inside the form tag are:

<input type = “text” | “password” | “checkbox” | “radio” | “submit” name=”string” value=”string” size=”n”>

In the above tag, the attribute type is used to implement text, password, checkbox, radio and submit button.

Text: It is used to input the characters of the size n and if the value is given than it is used as a default value. It uses single line of text. Each component can be given a separate name using the name attribute.

Password: It works exactly as text, but the content is not displayed to the screen, instead an * is used.

Radio: This creates a radio button. They are always grouped together with a same name but different values.

Checkbox: It provides a simple checkbox, where all the values can be selected unlike radio button.

Submit: This creates a button which displays the value attribute as its text. It is used to send the data to the server.

<select name=”string”>…..</select>

This tag helps to have a list of item from which a user can choose. The name of the particular select tag and the name of the chosen option are returned.

<option value=”string” selected>……</option>

The select statement will have several options from which the user can choose. The values will be displayed as the user moves through the list and the chosen one returned to the server.

<textarea name=”string” rows=”n” cols=”n”>…….</textarea>

This creates a free format of plain text into which the user can enter anything they like. The

area will be sized at rows by cols but supports automatic scrolling.

Example 6: HTML code that implements forms

```
<html>
<head>
<title>form</title>
</head>
<body>
<p align="left">Name:<input type="text" maxlength=30 size=15>
<p align="left">Password:<input type="password" maxlenght=10 size=15>
<p align="left">Qualification: <br><input type="checkbox" name="q"
value="be">B.E
<input type="checkbox" name="q" value="me">M.E
<p align="left">Gender:<br> <input type="radio" name="g" value="m">Male
<input type="radio" name="g" value="f">Female
<p align="left">course:<select name="course" size=1>
<option value=cse selected>CSE
<option value=it>CSIT
</select>
<p align="left">Address:<br><textarea name="addr" rows=4 cols=5
scrolling=yes></textarea>
<p align="center"><input type="submit" name="s" value="Accept">
<p align="center"><input type="reset" name="c" value="Ignore">
</body>
</html>
```

HTML5

HTML5 is the latest evolution of the standard that defines HTML. The term represents two different concepts. It is a new version of the language HTML, with new elements, attributes, and behaviors, and a larger set of technologies that allows the building of more diverse and powerful Web sites and applications. This set is sometimes called HTML5 & friends and often shortened to just HTML5.

Designed to be usable by all Open Web developers, this reference page links to numerous resources about HTML5 technologies, classified into several groups based on their function.

Semantics: allowing you to describe more precisely what your content is.
Connectivity: allowing you to communicate with the server in new and innovative ways.

Offline and storage: allowing webpages to store data on the client-side locally and operate offline more efficiently.

Multimedia: making video and audio first-class citizens in the Open Web.
2D/3D graphics and effects: allowing a much more diverse range of presentation options.

Performance and integration: providing greater speed optimization and better usage of computer hardware.

Device access: allowing for the usage of various input and output devices.
Styling: letting authors write more sophisticated themes

Cascading Style Sheet:-

CSS stands for Cascading Style Sheets. It not only extends its features in controlling colors and sizes of fonts, but also controls spaces between various elements, the color and width of a given line etc.

Syntax: p {font-size:30pt;}
p =selection
Font-size =property
30pt =value

Types of Style Sheets:-

1. Inline Style sheet
2. Embedded Style Sheet
3. External Style Sheet

An inline sheet applies style to a particular element in a webpage
Embedding a style sheet is for defining styles to a single webpage. Linking and importing are two ways to attach external style sheet to html documents.
An external style sheet is a file created separately, saved with the .css extension and attached to an HTML document by means of the link element. Linking an external style sheet can apply styles to multiple web pages.

Inline Style Sheet:-

Inclusion of style in a tag is called inline styles. Operator colon(:) Is followed by style property. To separate multiple properties we have to use operator semicolon(;)

Example:-

```
<html>
    <head>
        <title>Inline style</title>
</head>
<body bgcolor=”tan”>
<h1 align=”center” style=”color:white;background-color:blue”>
Inline Style Sheet
</h1>
<hr>
<p>Normal Para graph</p>
<p style=”font-size:5pt”>Web technologies</p>
<p style=”font-size:10pt”>Web technologies</p>
<p style=”font-size:20pt”>Web technologies</p>
</body>
</html>
```

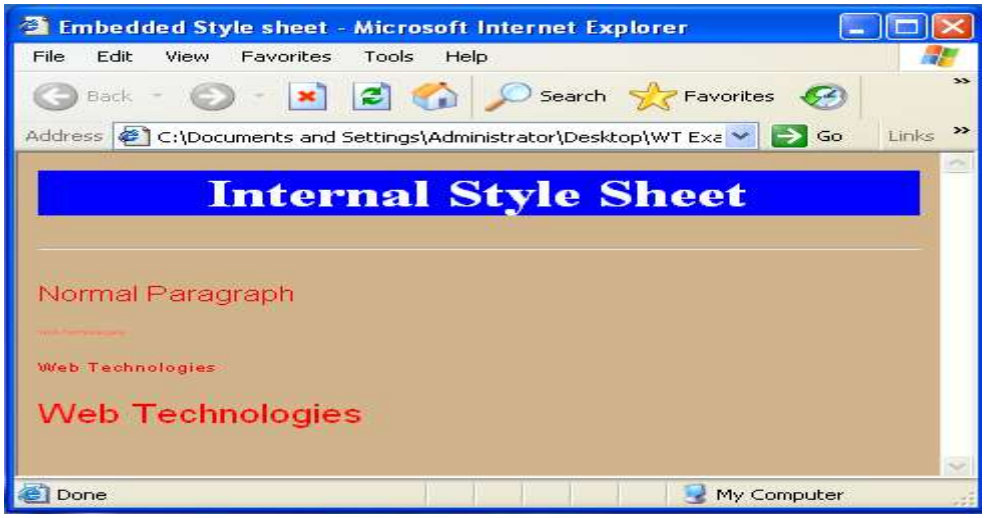
Embedded Style Sheet:-

If Style is used as tag, in Header Section then that style sheet is known as internal style sheet. If you include all the formatting parameters in between <style> and </style>, then this is called as internal style sheets or embedded style sheets. Advantage of Internal Style Sheet comparing with inline styles, at a time several tags can be formatted with internal style sheets, where as in inline styles only one tag at a time formatted.

```
<style>
        P       {color:red;font-family:arial}
        .s5     {font-size:5;}
        .s10    {font-size:10;}
        .s15    {font-size:20;}
        H1      {color:white;background-color:blue}
</style>
```

Example:-

```
<html>
<head>

    <title>Embedded Style sheet</title>
    <style>
     P      {color:red;font-family:arial}
    .s5     {font-size:5;}
    .s10    {font-size:10;}

    .s15    {font-size:20;}
  H1    {color:white;background-color:blue}

    </style>

    </head>
  <body bgcolor=tan>
            <h1 align=”center”> Internal Style Sheet</h1>
            <hr>
            <p>Normal Paragraph</p>
            <p class=”s5”>Web Technologies</p>
            <p class=”s10”>Web Technologies</p>
            <p class=”s5”>Web Technologies</p>

    </body>
    </html>
```



External Style Sheets:-

Third kind of style is external style sheet. In this total style elements are defined in a separate document, and this document is added to required web page. By using this we can use this style sheets in different web pages. As style sheet is separated from the document it gained the name External Style sheet.

Example:

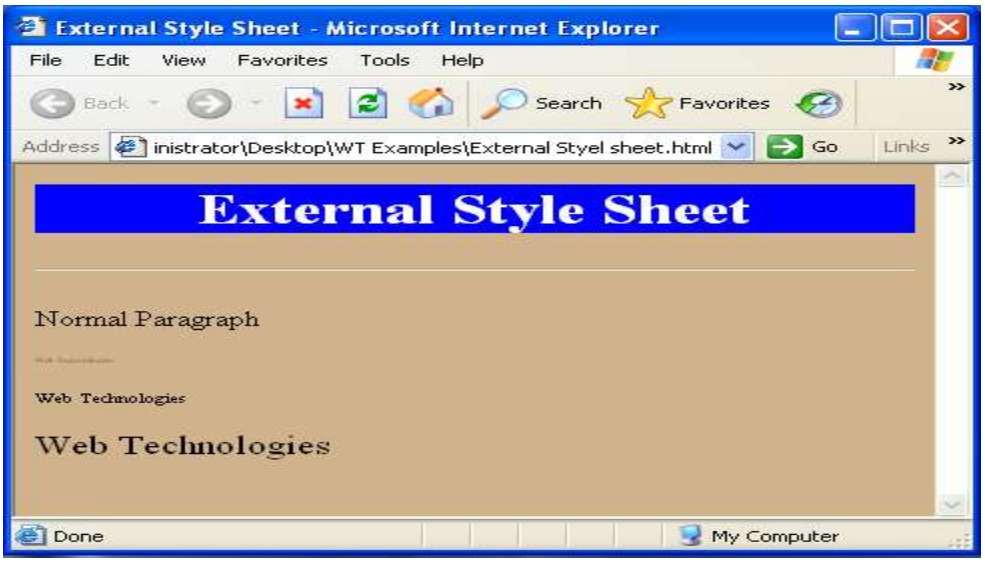
Write one html program that calls a style sheet file.

Ourstyles.css

```
<link rel=”stylesheet” type=”text/css” href=”ourstyles.css”>
        P       {color:red;font-family:arial}
        .s5     {font-size:5;}
        .s10    {font-size:10;}
        .s15    {font-size:20;}
        H1      {color:white;background-color:blue}
```

External style sheet.html

```
<html>
<head>
<title>External Style Sheet</title>
<link rel=”stylesheet” type=”text/css” href=”ourstyles.css”>
</head>
<body bgcolor=tan>
        <h1 align=”center”>External Style Sheet</h1>
        <hr>
        <p>Normal Paragraph</p>
        <p class=”s5”>Web Technologies</p>
        <p class=”s10”>Web Technologies</p>
        <p class=”s5”>Web Technologies</p>
</body>
</html>
```



Style Specification Formats

Format depends on the level of the style sheet

Inline:

Style sheet appears as the value of the style attribute

General form:

style = "property_1: value_1;
property_2: value_2;
...
property_n: value_n"

Choices of properties

Document-level:

Style sheet appears as a list of rules that are the content of a <style> tag

The <style> tag must include the type attribute, set to "text/css"

General form:

<style type = "text/css">
rule list
</style>

Form of the rules:

Rule

selector {list of property/values}

Each property/value pair has the form:

property: value

Pairs are separated by semicolons, just as in the value of a <style> tag

Comments in the rule list must have a different form - use C comments (/*...*/)

External style sheets

Form is a list of style rules, as in the content of a <style> tag for document-level style sheets

Selector forms:

CSS selectors are used *to select the content you want to style*. Selectors are the part of CSS rule set. There are several different types of selectors in CSS.

CSS Element Selector

CSS Id Selector

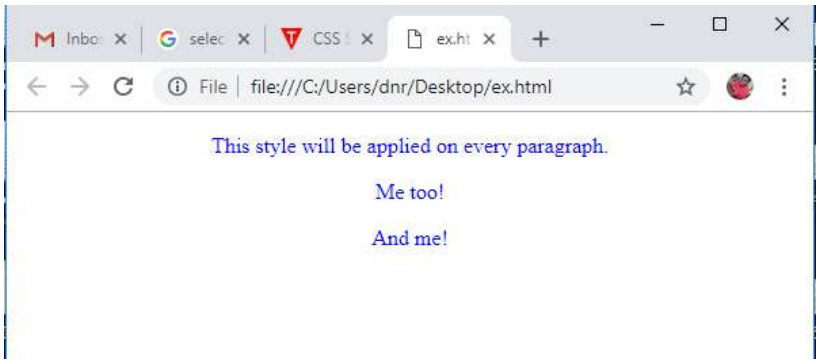
CSS Class Selector

1) Element Selector

The element selector selects the HTML element by name.

```
<!DOCTYPE html>
<html>
<head>
<style>
p{
   text-align: center;
   color: blue;
}
</style>
</head>
<body>
<p>This style will be applied on every paragraph. </p>
<p id="para1">Me too!</p>
<p>And me!</p>
</body>
</html>
```

Output:



2) **Id Selector**

The id selector selects the id attribute of an HTML element to select a specific element. An id is always unique within the page so it is chosen to select a single, unique element.

It is written with the hash character (#), followed by the id of the element.

Let?s take an example with the id "para1".

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

```
<head>
<style>
#para1 {
   text-align: center;
   color: blue;
}
</style>
</head>
<body>
<p id="para1">Hello wtlab.com</p>
<p>This paragraph will not be affected.</p>
</body>
</html>
```

Output

---

3) **Class Selector**:

The class selector selects HTML elements with a specific class attribute. It is used with a period character . (full stop symbol) followed by the class name.

Let's take an example with a class "center".

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
   text-align: center;
   color: blue;
}
</style>
</head>
<body>
<h1 class="center">This heading is blue and center-aligned.</h1>
<p class="center">This paragraph is blue and center-aligned.</p>
</body>
</html>
```

Output:

The CSS Box Model

All HTML elements can be considered as boxes. In CSS, the term "box model" is used when talking about design and layout.

The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content. The image below illustrates the box model:

Explanation of the different parts:

Content - The content of the box, where text and images appear
Padding - Clears an area around the content. The padding is transparent
Border - A border that goes around the padding and content
Margin - Clears an area outside the border. The margin is transparent
The box model allows us to add a border around elements, and to define space between elements.

Demonstration of box model

```
<html>
<head>
<style>
div {
  background-color: lightgrey;
  width: 300px;
  border: 15px solid green;
  padding: 50px;
  margin: 20px;
}
</style>
</head>
```

```
<body>

<h2>Demonstrating the Box Model</h2>

<p>SIR CRR COLLEGE OF ENGINEERING</p>

<div>BTECH
        MTECH
                MCA
.</div>

</body>
</html>
```

Output:

Resolving conflicts in CSS made simple

Two or more conflicting CSS rules are sometimes applied to the same element. What are the rules in CSS that resolve the question of which style rule will actually be used when a page is rendered by a browser? The answer is, “it’s complicated.” Several factors are involved. I’ll give you a brief explanation of each factor.

Inheritance
Some properties are passed from parent to child. For example, this rule in a style sheet would be inherited by all child elements of the body and make every font on the page display as Georgia.

```
body {font-family: Georgia;}
```

The Cascade
Within the cascade, more than one factor can figure into determining which one of several conflicting CSS rules will actually be applied. These factors are source order, specificity and importance. Location is part of the cascade, too.

Source order means the order in which rules appear in the style sheet. A rule that appears later in the source order will generally overrule an earlier rule. Consider this example.

```
body {font-family: Georgia;}
h1, h2, h3 {font-family: Arial;}
```

Because the h1, h2, and h3 selectors are in the source order after the body rule, the headings would

display in Arial, not in Georgia. There's also the fact that the selector is picking specific elements.

Specificity
Specificity is determined by a mathematical formula, but common sense can help you understand it.

```
p {font-family: Georgia;}
.feature  p {font-family: Arial;}
```

In this case, the selector .feature p is more specific than the selector p. For any paragraph assigned to the class 'feature' the font-family would be Arial. Common sense tells you that selecting a paragraph that belongs to a particular class is a more specific choice than selecting all paragraphs. The more specific selector overrules the less specific selector.

!important
There are rules that are declared !important. !important rules always overrule other rules, no matter what inheritance, source order or specificity might otherwise do. A user created stylesheet can use !important to overrule the author's CSS.

```
*{font-family: Arial !important;}
```

This rule would mean that everything (* selects everything) would be Arial no matter what other rules were used in the CSS.

Location
Style rules can exist in a number of locations in relation to the HTML page affected. The location of a rule also plays into determining which rule actually ends up being implemented. The locations are:

Browser style rules
External style rules
Internal style (in the document head) rules
Inline style rules
Individual user style rules
In the list, the browser style rules are the most "distant" from the HTML page, the individual user styles are the "closest." Within this cascade of style declarations, the closest rule wins. An inline style overrules an external style, which overrules a browser style.

PREVIOUS QUESTION PAPERS

1 a) Explain about the following terms with examples	
(i) Unordered Lists (ii) Ordered Lists.	[7]
b) Illustrate the concept of Conflict Resolution.	[7]
2. a) State the concept of Forms in HTML with examples.	[7]
b) Discuss about Box Model in CSS.	[7]
3. a) Give a detail note on Standard HTML Document Structure.	[7]
b) Explain the purpose of and tags with examples.	[7]
4. a) Summarize the concept of Margins and Padding in CSS.	[7]
b) Outline the purpose of rowspan and colspan attributes.	[7]

Script means small piece of code. Scripting languages are two kinds one is client-side other one is servers-side scripting. In general client-side scripting is used for verifying simple validation at client side; server-side scripting is used for database verifications. VBScript, java script and Jscript are examples for client-side scripting and ASP, JSP, Servlets, PHP etc. are examples of server-side scripting.

JavaScript (originally known as "Live Script") is a scripting language that runs inside the browser to manipulate and enhance the contents of Web pages. Java Script is designed to add interactivity to HTML pages. Web pages are two types

1.Static web page 2.Dynamic webpage

- Static web page where there is no specific interaction with the client
- Dynamic web page which is having interactions with client and as well as validations can be added.

Simple HTML script is called static web page, if you add script to HTML page it is called dynamic page. Netscape navigator developed java script. Microsoft's version of JavaScript is Jscript.

- Java script code as written between <script> -------- </script>tags
- All java script statements end with a semicolon
- Java script ignores white space
- Java script is case sensitive language
- Script program can save as either. Js or.html

Benefits of JavaScript

- It is widely supported by web browsers;
- It gives easy access to the document objects and can manipulate most of them.
- Java Script gives interesting animations with long download times associated with many multimedia data types;
- Web surfers don't need a special plug-in to use your scripts
- Java Script relatively secure - you can't get a virus infection directly from Java Script.
- JavaScript code resembles the code of C Language; the syntax of both the language is very close to each other. The set of tokens and constructs are same in both the language.

Problems with JavaScript

- Most scripts relay upon manipulating the elements of DOM;
- Your script does not work then your page is useless
- Because of the problems of broken scripts many web surfers disable java script support in their browsers
- Script can run slowly and complex scripts can take long time to start up

Similarities between java script and java:

1. Both java script and java having same kind of operators
2. Java script uses similar control structures of java
3. Nowadays both are used as languages for use on internet.
4. Labeled break and labeled continue both are similar
5. Both are case-sensitive languages

Difference between java script and java:

Java Script	Java
Java Script is scripting language	Java is a programming language
Java Script is object-based programming language.	Java is object –oriented programming language
Java script code is not compiled, only interpreted.	Java is compiled as well as interpreted language
Java Script is Weekly-typed language	Java is Strongly-typed language
JavaScript code is run on a browser only.	Java creates applications that run in a virtual machine or browser

The syntax of the script tag isas follows:

```
<script language=""scripting language name"">
----------------------------------------
</script>
```

The language attribute specifies the scripting language used in the script. Both Microsoft internet explorer and Netscape navigator use java script as the default scripting language. The script tag may be placed in either the head or the body or the body of an HTML document.

```
Ex: <script language=""javascript"">
        ------------
        ------------
        ------------
</script>
```

Comments in JavaScript:

Single line comment- //

Multi-line comment- <!-- comment -->

Operators in JavaScript:

- Arithmetic operators (+,-,*,/,%)
- Relational operators (<,>,!=,<=,>=)
- Logical operators (&&,||,!)
- Assignment operator(=)
- Increment decrement operators(++,--)
- Conditional/Ternary operator (?:)
- Bitwise operators (&,|,!)

Control structures:

- If statement
- Switch
- While
- Do-while
- For
- Break
- Continue

Control structures syntax and working as same as java language.

Variables

Variables are like storage units/place holders to hold values. A variable is a memory location to hold certain different types of data. In Javascript, A variable can store all kinds of data. It is important to know the proper syntax to which variables must conform:

- They must start with a letter or underscore ("_")
- Subsequent characters can also be digits (0-9) or letters (A-Z and/or a-z). Remember, JavaScript is case-sensitive. (That means that MyVar and myVar are two different names to JavaScript, because they have different capitalization.)
- You cannot use reserved words as variable names.
- You cannot use spaces in names.
- Names are case-sensitive.

Syntax:

var v_name = value;

Examples of legal variable names are fname, temp99, and _name.

When you declare a variable by assignment outside of a function, it is called global variable, because it is available everywhere in the document. When you declare a variable within a function, it is called local variable, because it is available only within the function. To assign a value to a variable, you use the following notation:

```
var num = 8; var
real= 4.5;
```

```
var myString = "Web Technologies";
```

Values of Variables(Data types)

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159
- Boolean values, either true or false
- Strings, such as "Howdy!"
- NULL, a special keyword which refers to nothing.

JavaScript Screen Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

Using innerHTML

To access an HTML element, JavaScript can use the document.getElementById(id) method.

The id attribute defines the HTML element. The innerHTML property defines the HTML content:

Example

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Web Page</h2>
<p>My First Paragraph.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

My First Web Page

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
document.write(8 + 6);
</script>
</body>
</html>
```

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<button type="button" onclick="document.write(5 + 6)">Try it</button>
</body>
</html>
```

Using window.alert()

You can use an alert box to display data:

Example

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

```
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
window.alert(5 + 6);
</script>
</body>
</html>
```

You can skip the window keyword.

In JavaScript, the window object is the global scope object, that means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional:

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
alert(5 + 6);
</script>
</body>
</html>
```

Using console.log()

For debugging purposes, you can call the console.log() method in the browser to display data.

Example

```
<!DOCTYPE html>
<html>
<body>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

JavaScript Print

JavaScript does not have any print object or print methods.

You cannot access output devices from JavaScript.

The only exception is that you can call the window.print() method in the browser to print the content of the current window.

Example

```
<!DOCTYPE html>
<html>
<body>
<button onclick="window.print()">Print this page</button>
</body>
</html>
```

OBJECTS IN JAVA SCRIPT

When you load a document in your Web browser, it creates a number of JavaScript objects with properties and capabilities based on the HTML in the document and other information. These objects exist in a hierarchy that reflects the structure of the HTML page itself. The pre-defined objects that are most commonly used are the window and document objects. Some of the useful Objects are:

1. Document
2. Window
3. Browser

4. Form
5. Math
6. Date

The Document Object

A document is a web page that is being either displayed or created. The document has a number of properties that can be accessed by JavaScript programs and used to manipulate the content of the page.

write or writeln

Html pages can be created on the fly using JavaScript. This is done by using the write or writeln methods of the document object.

Syntax:

document.write("String");
document.writeln("String");

In this document is object name and write () or writeln () are methods. Symbol period is used as connector between object name and method name. The difference between these two methods is carriage form feed character that is new line character automatically added into the document.

Example:
```
document.write("<body>");
document.write("<h1> Hello</h1>");
```

bgcolor and fgcolor

These are used to set background and foreground(text) color to webpage. The methods accept either hexadecimal values or common names for colors.

Syntax: document.bgcolor="#1f9de1"; document.fgcolor="silver";

anchors

The anchors property is an array of anchor names in the order in which they appear in the HTML Document. Anchors can be accessed like this:

Syntax: document.anchors[0]; document.anchors[n-1];

Links

Another array holding all links in the order in which they were appeared on the Webpage

Forms

Another array, this one contains all of the HTML forms. By combining this array with the individual form objects each form item can be accessed.

The Window Object

The window object is used to create a new window and to control the properties of window.

Methods:

1. *open("URL","name") :* This method opens a new window which contains the document specified by URL and the new window is identified by its name.
2. *close():* this shutdowns the current window.

Properties:

toolbar = [1|0] location= [1|0] menubar = [1|0] scrollbars = [1|0] status = [1|0] resizable = [1|0]

where as 1 means *on* and 0 means *off*

height=pixels, width=pixels : These properties can be used to set the window size.

The following code shows how to open a new window

```
newWin = open("first.html","newWin","status=0,toolbar=0,width=100,height=100");
```

Window object supports three types of message boxes.

1. Alert box
2. Confirm box
3. Prompt box

Alert box is used to display warning/error messages to user. It displays a text string with *OK* button.

Syntax: window. Alert ("Message");

Confirm Box is useful when submitting form data. This displays a window containing message with two buttons: *OK* and *Cancel.* Selecting *Cancel* will abort the any pending action, while *OK* will let the action proceed.

Syntax

window.confirm("String");

Prompt box used for accepting data from user through keyboard. This displays simple window that contains a prompt and a text field in which user can enter data. This method has two parameters: a text string to be used as a prompt and a string to use as the default value. If you don"t want to display a default then simply use an empty string. Syntax

Variable=window.prompt("string","default value");

The Form Object

Two aspects of the form can be manipulated through JavaScript. First, most commonly and probably most usefully, the data that is entered onto your form can be checked at submission. Second you can actually build forms through JavaScript.

Form object supports three events to validate the form

onClick = "method()"

This can be applied to all form elements. This event is triggered when the user clicks on the element.

onSubmit = "method()"

This event can only be triggered by form itself and occurs when a form is submitted.

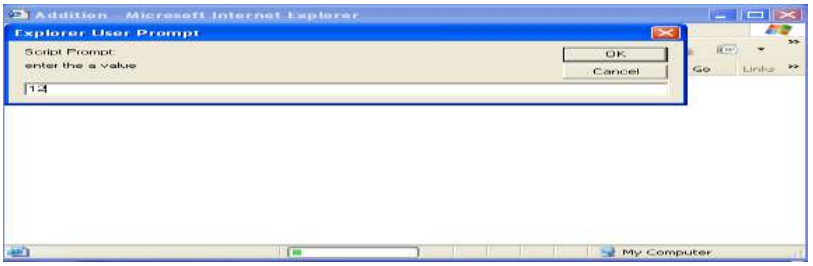
onReset = "method()"

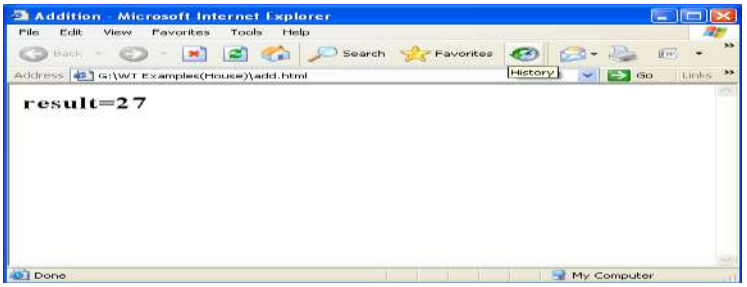
This event can only be triggered by form itself and occurs when a form is reset.

Example :

Write a java script addition of two numbers.

```
<html>
<head>
<title>addition</title>
<script language=”java script”> Var s1,s2,a,b,c;
S1=window.prompt(“enter the a value”,”0”);
S2=window.prompt(“enter the b value”,”0”); A=parseInt(s1);
b=parseInt(s2);
c=a+b;
document.write(“<h2>result=”+c+</h2>”);
</script>
</head>
</html>
```





Example: HTML program that applies a random background color when you click on button

```
<html>
<head>
<script language = "javascript">
        function change()
        {
            var clr = document.bgColor=parseInt(Math.random()*999999);
            document.f1.color.value=clr;
         }
  </script>
  </head>
<body>
<form
```

```
<form name="f1">
        <input type="text" name="color"/>
        <input type="button" value="Cick me" onclick="change()"/>
 </form> </body></html>
```

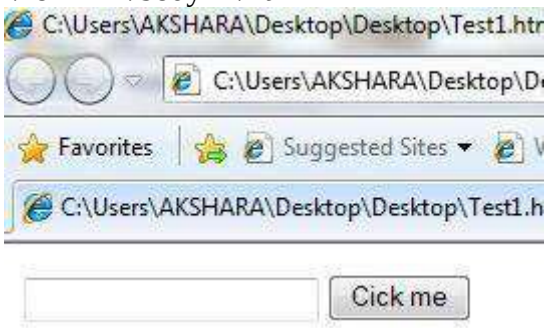


Fig.1: On first run backgroundis white

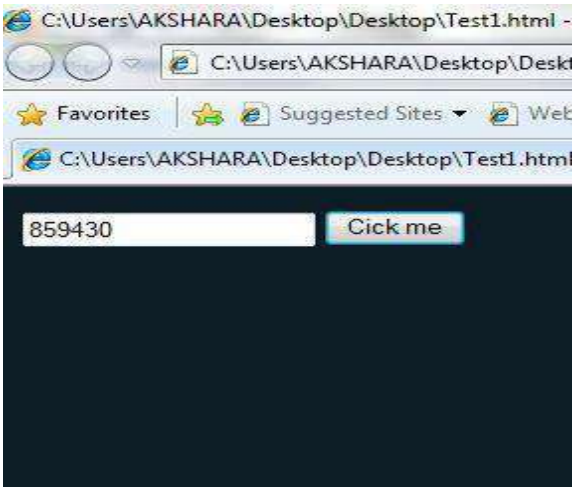
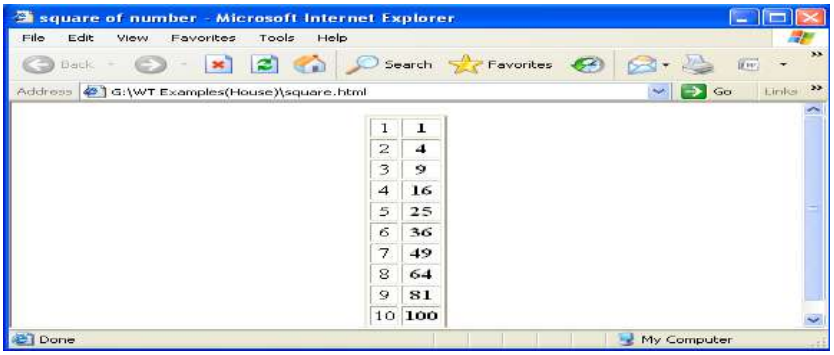


Fig.2: After clicking on button"Clickme": background changed to "black"

Example:
Write a program to display 1 to 10 and square of number.

```
<html>
<head>
<title>square of number</title>
<script language="java script">
Var i=1,n;
Document .writeln("<center><table border=1>");
While(i<=10)
{
Document.writeln("<tr><td>"+i+"<td>"+(i*i)+"</tr>"); i++;
}
Document.writeln("</table></center>");
</script>
</head>
</html>
```



Example :
Write a program to display table for given number

```
<html>
<head>
<title>nth table </title>
<script language="java script">
Var i=i,n;
```

```
n=parseInt(window.prompt(“enter n value”,”0”));
Document .writeln(“<center><table border=1>”);
For(i=1;i<=10;i++)
{
Document.writeln(“<tr><td>”+n+”<td>X<td>”+i+”<td>=”+(n*i)+”</tr>”);
}
Document.writeln(“</table></center>”);
</script>
</head></html>
```

The browser Object

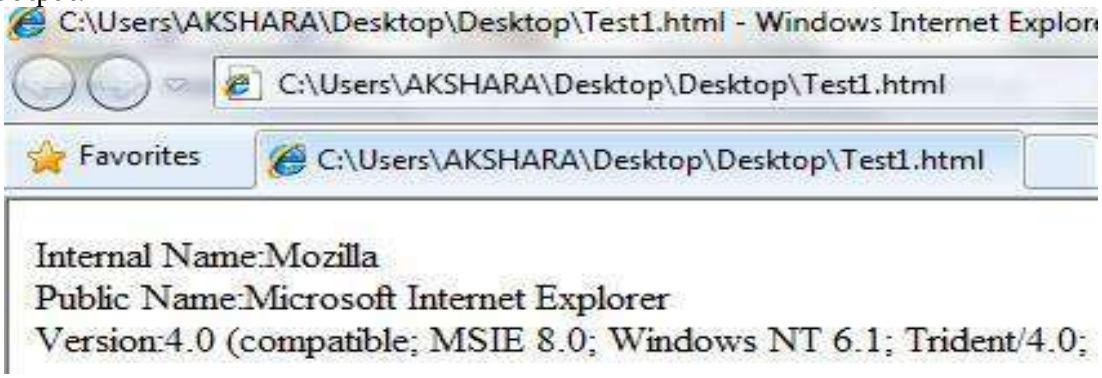
The browser is JavaScript object that can be used to know the details of browser. Some of the properties of the browser object is as follows:

Property	Description
navigator.appCodeName	It returns the internal name for the browser. For major browsers it is *Mozilla*
navigator.appName	It returns the public name of the browser – navigator or Internet Explorer
navigator.appVersion	It returns the version number, platform on which the browser is running.
navigator.userAgent	The strings appCodeName and appVersion concatenated together
navigator.plugins	An array containing details of all installed plug-ins
Navigator.mimeTypes	An array of all supported MIME Types

Example: Write javascript to display internal details of a browser (*Test.html*)

```
<script language = "javascript">
<!--
   document.writeln("Internal      Name:"+navigator.appCodeName);
   document.writeln("<br/>Public     Name:"+navigator.appName);
   document.writeln("<br/>Version:"+navigator.appVersion);
-->
</script>
```

Output:



The Math Object

The *Math* object holds all mathematical functions and values. All the functions and attributes used in complex mathematics must be accessed via this object.

Syntax:

```
Math.methodname();
Math.value;
```

Method	Description	Example

Math.abs(x)	Returns the absolute value	Math.abs(-20) is 20
Math.ceil(x)	Returns the ceil value	Math.ceil(5.8) is 6 Math.ceil(2.2) is 3
Math.floor(x)	Returns the floor value	Math.floor(5.8) is 5 Math.floor(2.2) is 2
Math.round(x)	Returns the round value, nearest integer value	Math.round(5.8) is 6 Math.round(2.2) is 2
Math.trunc(x)	Removes the decimal places it returns only integer value	Math.trunc(5.8) is 5 Math.trunc(2.2) is 2
Math.max(x,y)	Returns the maximum value	Math.max(2,3) is 3 Math.max(5,2) is 5
Math.min(x,y)	Returns the minimum value	Math.min(2,3) is 2 Math.min(5,2) is 2
Math.sqrt(x)	Returns the square root of x	Math.sqrt(4) is 2
Math.pow(a,b)	This method will compute the a^b	Math.pow(2,4) is 16
Math.sin(x)	Returns the sine value of x	Math.sin(0.0) is 0.0
Math.cos(x)	Returns cosine value of x	Math.cos(0.0) is 1.0
Math.tan(x)	Returns tangent value of x	Math.tan(0.0) is 0
Math.exp(x)	Returns exponential value i.e e^x	Math.exp(0) is 1
Math.random(x)	Generates a random number in between 0 and 1	Math.random()
Math.log(x)	Display logarithmic value	Math.log(2.7) is 1
Math.PI	Returns a ∏ value	a = Math.PI; a = 3.141592653589793

The Date Object

This object is used for obtaining the date and time. In JavaScript, dates and times represent in milliseconds since 1st January 1970 UTC. JavaScript supports two time zones: UTC and local. UTC is Universal Time, also known as Greenwich Mean Time(GMT), which is standard time throughout the world. Local time is the time on your System. A JavaScript *Date* represents date from -1,000,000,000 to -1,000,000,000 days relative to 01/01/1970.

Date Object Constructors:

new Date(); Constructs an empty date object.

new Date("String"); Creates a Date object based upon the contents of a text string.

new Date(year, month, day[,hour, minute, second]); Creates a Date object based upon the numerical values for the year, month and day.

```
var dt=new Date();
document.write(dt);          // TueDec 23 11:23:45 UTC+0530 2015
```

Methods in Date object:

Java script date object provides several methods, they can be classified is string form, get methods and set methods. All these methods are provided in the following table.

Method	Description
getDate()	Returns day of the month i.e. 1 to 31
getDay()	Returns an integer representing day of the week(0 - 6), Sunday to Saturday respectively.
getMonth()	Returns month of the year from 0 to 11, January to December respectively.
getFullYear()	Returns four-digit year number
getHours()	Returns hour field of the Date Object in 24 hours time format (0 to 23)

getMinitues()	Returns minute field of the Date Object from 0 to 59
getseconds()	Returns seconds field of the Date Object 0 to 59
setDate(v)	To set the date, day, month and full year
setDay(v)	
setMonth(v)	
setFullYear(y,m,d)	
setHours(v)	To set the hours, minutes, seconds of time
setMinutes(v)	
setSeconds(v)	
toString()	Returns the Date as a string

Boolean object:

Wrapper object for Boolean data type is Boolean object. To manipulate Boolean data types in java script program these objects are provided. Constructor to create Boolean object is

Var n=new Boolean(Boolean value)

For example,

Var n=new Boolean(true);

It accepts parameters, true, false, 0, 1 number. NaN or empty string. If it empty string or numbers. NaN it treats as false values.

Methods in Boolean object:

Tostring()-to convert to string for true or false values.

Valueof()-to get value of Boolean object.

NUMBER OBJECT:

Object wrapper in JavaScript is number object, for any number data type, this act as wrapper class. Same methods are applied for number object also. Constructor is as follows:

Var n=new number (any value)

For example

Var n=number(432,123);

Properties of number object:

Number.MAX_VALUE.number.MIN_VALUE, number.NaN,number.NEGITIVE_INFINITY and number.POSITIVE_INFINITY.

Number.NaN is for not a number.

Object:

Window.open(): syntax:window.open(file name,name,properties)

Window.close(): syntax:window.close()

Properties:

Directories -yes or no

Height -number of pixels

Width -number of pixels.

Location -yes or no

Menubar -yes or no

Resizable -yes or no

Scrollobars -yes or no

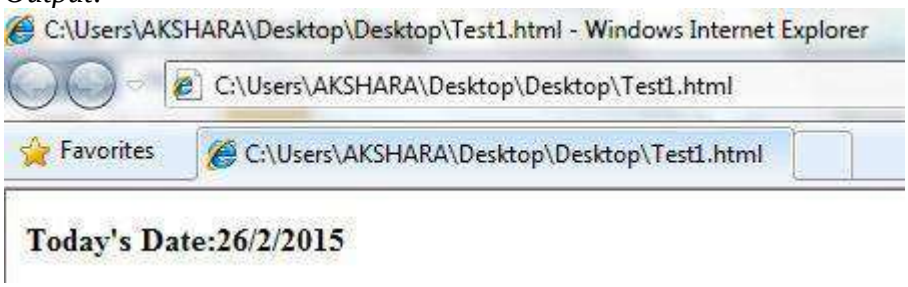
Status -yes or no Toolbar -yes or no Always lowered -yes or no

Alwaysraised -yes or no Dependent -yes or no Hotkeys -yes or no

Example: Write javascript to display internal details of a browser (*Test.html*)

```
<script language = "javascript">
<!--
      var dt = new Date(); var day = dt.getDate();
      var month = dt.getMonth()+1; var year = dt.getFullYear();
      document.writeln("<h4>Today's       Date:"+day+"/"+month+"/"+year);-
-->
</script>
```

Output:



EVENTS

ONCLICK event:

ONCLICK event is used for responding for mouse click action. when the user presses a button, it can call any function through ONCLICK event. For example:

<INPUT TYPE=”BUTTON”VALUE=”OK”ONCLICK=”fun()”>

When the button OK is clicked it calls the function display and according to action given in fun() function it will work.

Example:

Write a program to display a button OK and display your name when it is clicked.

```
<html>
<head>
<title> onclick example</title>
<script language=”java script”>
Function fun()
{
     Window .alert(“hai,this is Raju”);
}
</script>
</head>
<body>
<form>
<INPUT TYPE=”BUTTON” VALUE=”OK“ ONCLICK=”fun()”>
</form>
</body>
</html>
```

onsubmit event:

Example :

```
<!DOCTYPE html>
<html>
<body>
<p>When you submit the form, a function is triggered which alerts some text.</p>
<form action="/action_page.php" onsubmit="myFunction()">
            Enter name: <input type="text" name="fname">
           <input type="submit" value="Submit">
</form>
<script>
function myFunction() {
            alert("The form was submitted");
}
</script>
</body>
</html>
```

onload event:

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

```
<title> onload example</title>
<script language=”java script”>
Function fun()
{
    Window .alert(“hai,this is ramesh”);
}
</script>
</head>
<body onload=”fun()”>
</body>
</html>
```

Accepting values from text box:

In javascript we can refer the entered values from the textbox which is defined in HTML forms. By taking the name parameter we can perform this task. Simply we can give as follows

Variable=document. Formname.textboxname.value;

For example, form name is “form1” and text box name is “user” then you can assign its values in java script variable as follows:

Str=document .form1.user.value;

Example:Write a program to login form

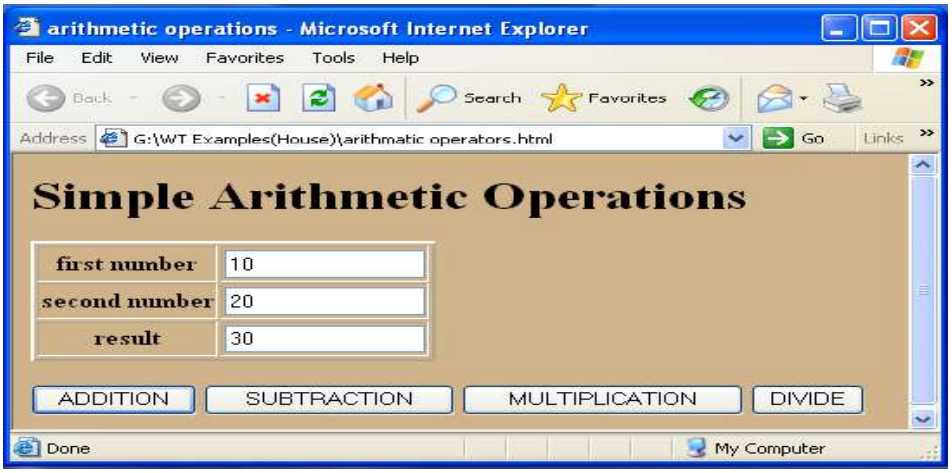
```
<html>
<head>
<script language="javascript"> function
fun()
{
var s1="vrs"; var
s2="yrn"; var s3,s4;
s3=document.f1.t1.value;
s4=document.f1.t2.value;
if((s1==s3)&&(s2==s4))
window.alert("user name and password correct");
else if(!(s1==s3)&&(s2==s4))
        window.alert("user name wrong");
else if((s1==s3)&&!(s2==s4))
        window.alert("password wrong");
    else if(!(s1==s3)&&!(s2==s4))
        window.alert("both user name and password wrong");
</script>
</head>
<body>
<center>
<form name="f1" onsubmit="fun()">
<table border="1">
<tr><td>user name:<td><input type="text" name="t1" size=20></tr>
<tr><td>password:<td><input type="password" name="t2" size=20></tr>
<tr><td colspan=2 align="center"><input type="submit" value="submit"></tr>
</table>
</form>
</center>
</body>
</html>
```

Example:

Write a program to display a form with three text fields, two for accepting numbers and third one for displaying result form should contain four buttons with labels ADD, SUBTRACT, MULTIPLY and DIVIDE .perform the respective arithmetic operations and display the result in the third text box.

```
<html>
<head>
<title>arithmetic operations</title>
<script language ="javascript">
 function cal(op)
{
var a,b,c=0; a=parseInt(document.form1.first.value);
b=parseInt(document.form1.second.value);
switch(op)
{
case '+':c=a+b; break;
 case '-':c=a-b; break;
case '*':c=a*b; break;
case '/':c=a/b; break;
}
document.form1.result.value=c;
}
</script>
</head>
<body bgcolor=tan>
<form name="form1">
<h1>Simple Arithmetic Operations</h1>
<TABLE border='2'>
<TR>
<TH> first number
<TD><input type="text" name="first" size=15>
</TR>
<TR>
<TH>second number
<TD><input type="text" name="second" size=15>
</TR>
<TR>
<TH>result
<TD><input type="text" name="result" size=15>
</TR>
</TABLE>
<p>
<INPUT TYPE =BUTTON VALUE="ADDITION" ONCLICK=cal("+")>
        <INPUT TYPE =BUTTON VALUE="SUBTRACTION" ONCLICK=cal("-")>
<INPUT TYPE =BUTTON VALUE="MULTIPLICATION" ONCLICK=cal("*")>
        <INPUT TYPE =BUTTON VALUE="DIVIDE" ONCLICK=cal("/")>
```

```
<input type=BUTTON name="reset_form" value="Reset" onclick="this.form.reset();">
</body>
</html>
```



Arrays

Array is a collection of similar type of elements which can be referred by a common name. Any element in an array is referred by an array name followed by [position of the element]. The particular position of element in an array is called array index or subscript.

Arrays are two types 1. Single dimensional arrays
2. Multi dimensional arrays

Single dimensional array:

Array Declaration: "new" operator is used to declare and allocate memory for array.
Operator new is also known as dynamic memory allocation operator.

Syntax: var variablename=new array(size);
Ex: var a=new array(10);

Initialization of array elements:

If you want to initialize the array elements with zeros then you can use for loop for that purpose, observe the following example:

```
var num=new array(10);
for(i=0;i<num.length;i++)
{
   num[i]=0;
}
```

The same thing can be achieved through for/in control structure, that enables to process each element in an array.
For example,

```
var num =new array(10);
for(var i in num)
{
    num[i]=0;
}
```

The above statements show the way of usage for/in control structure.

Two Dimensional Array:

For example we want required 3 rows 2 cols.
First create 3 rows var
a =new array(3);

a[0]
a[1]
a[2]

Then create 2 cols each row

```
for(i=0;i<3;i++)
{
 a[i]=new array(2);
}
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax: var *array_name* = [*item1*, *item2*, ...];

Example

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

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Spaces and line breaks are not important. A declaration can span multiple lines:

Example

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

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = [
  "Saab",
  "Volvo",
  "BMW"
];
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example

```
var cars = new Array("Saab", "Volvo", "BMW");
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
```

```
<p id="demo"></p>
<script>
var cars = new Array("Saab", "Volvo", "BMW");
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

The two examples above do exactly the same. There is no need to use new Array().
For simplicity, readability and execution speed, use the first one (the array literal method).

Access the Elements of an Array

You access an array element by referring to the **index number**.

This statement accesses the value of the first element in cars:

```
var name = cars[0];
```

Example

```
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
</script>
</body>
</html>
```

Note: Array indexes start with 0. [0] is the first element. [1] is the second element.

Changing an Array Element

This statement changes the value of the first element in cars:

```
cars[0] = "Opel";
```

Example

```
var cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
document.getElementById("demo").innerHTML = cars[0];
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

Example

```
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Arrays are Objects

Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.
But, JavaScript arrays are best described as arrays.
Arrays use **numbers** to access its "elements". In this example, person[0] returns John:

Array: ***var person = ["John", "Doe", 46];***

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Arrays use numbers to access its elements.</p>
<p id="demo"></p>
<script>
var person = ["John", "Doe", 46];
document.getElementById("demo").innerHTML = person[0];
</script>
</body>
</html>
```

Objects use **names** to access its "members". In this example, person.firstName returns John:

Object: ***var person = {firstName:"John", lastName:"Doe", age:46};***

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p>JavaScript uses names to access object properties.</p>
<p id="demo"></p>
<script>
var person = {firstName:"John", lastName:"Doe", age:46};
document.getElementById("demo").innerHTML = person["firstName"];
</script>
</body>
</html>
```

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.
Because of this, you can have variables of different types in the same Array.
You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

Examples

```
var x = cars.length;   // The length property returns the number of elements
```

```
var y = cars.sort();   // The sort() method sorts arrays
```
The length Property

The length property of an array returns the length of an array (the number of array elements).

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length;   // the length of fruits is 4
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The length property returns the length of an array.</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.length;
</script>
</body>
</html>
```

The length property is always one more than the highest array index.

Accessing the First Array Element

Example

```
fruits = ["Banana", "Orange", "Apple", "Mango"];
var first = fruits[0];
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accesses using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var first = fruits[0];
document.getElementById("demo").innerHTML = first;
</script>
</body>
</html>
```

Accessing the Last Array Element

Example

```
fruits = ["Banana", "Orange", "Apple", "Mango"];
var last = fruits[fruits.length - 1];
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accesses using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var last = fruits[fruits.length-1];
document.getElementById("demo").innerHTML = last;
</script>
</body>
</html>
```

Looping Array Elements

The safest way to loop through an array, is using a for loop:

Example

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The best way to loop through an array is using a standard for loop:</p>
<p id="demo"></p>
<script>
var fruits, text, fLen, i;
fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;
text = "<ul>";
for (i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
document.getElementById("demo").innerHTML = text;
</script>
</body>
</html>
```

You can also use the Array.forEach() function:

Example

```
var fruits, text;
fruits = ["Banana", "Orange", "Apple", "Mango"];
text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
function myFunction(value) {
  text += "<li>" + value + "</li>";
}
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Array.forEach() calls a function for each array element.</p>
<p>Array.forEach() is not supported in Internet Explorer 8 or earlier.</p>
<p id="demo"></p>
<script>
var fruits, text;
fruits = ["Banana", "Orange", "Apple", "Mango"];
text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
document.getElementById("demo").innerHTML = text;
function myFunction(value) {
  text += "<li>" + value + "</li>";
}
</script>
</body>
</html>
```

Adding Array Elements

The easiest way to add a new element to an array is using the push() method:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Lemon");    // adds a new element (Lemon) to fruits
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The push method appends a new element to an array.</p>
<button onclick="myFunction()">Try it</button>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
function myFunction() {
  fruits.push("Lemon");
  document.getElementById("demo").innerHTML = fruits;
}
</script>
</body>
</html>
```

New element can also be added to an array using the length property:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Lemon";    // adds a new element (Lemon) to fruits
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Array.forEach() calls a function for each array element.</p>
<p>Array.forEach() is not supported in Internet Explorer 8 or earlier.</p>
<p id="demo"></p>
<script>
var fruits, text;
fruits = ["Banana", "Orange", "Apple", "Mango"];
text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
document.getElementById("demo").innerHTML = text;
function myFunction(value) {
  text += "<li>" + value + "</li>";
}
</script>
</body>
</html>
```

Associative Arrays

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.In JavaScript, **arrays** always use **numbered indexes**.

Example

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;     // person.length will return 3
```

```
var y = person[0];         // person[0] will return "John"
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
document.getElementById("demo").innerHTML =
person[0] + " " + person.length;
</script>
</body>
</html>
```

WARNING !!

If you use named indexes, JavaScript will redefine the array to a standard object.
After that, some array methods and properties will produce **incorrect results**.

Example:

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;     // person.length will return 0
var y = person[0];         // person[0] will return undefined
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>If you use a named index when accessing an array, JavaScript will redefine the array to a standard object, and some array methods and properties will produce undefined or incorrect results.</p>
<p id="demo"></p>
<script>
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
document.getElementById("demo").innerHTML = person[0] + " " + person.length;
</script>
</body>
</html>
```

The Difference Between Arrays and Objects

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.

Arrays are a special kind of objects, with numbered indexes.

When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

Avoid new Array()

There is no need to use the JavaScript's built-in array constructor `new` Array().

Use [] instead.

These two different statements both create a new empty array named points:

```
var points = new Array();     // Bad
var points = [];              // Good
```

These two different statements both create a new array containing 6 numbers:

```
var points = new Array(40, 100, 1, 5, 25, 10); // Bad
var points = [40, 100, 1, 5, 25, 10];          // Good
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Avoid using new Array(). Use [] instead.</p>
<p id="demo"></p>
<script>
//var points = new Array(40, 100, 1, 5, 25, 10);
var points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points[0];
</script>
</body>
</html>
```

The new keyword only complicates the code. It can also produce some unexpected results:

```
var points = new Array(40, 100);  // Creates an array with two elements (40 and 100)
```

What if I remove one of the elements?

```
var points = new Array(40);  // Creates an array with 40 undefined elements !!!!!
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Avoid using new Array().</p>
<p id="demo"></p>
<script>
var points = new Array(40);
document.getElementById("demo").innerHTML = points[0];
</script>
</body>
</html>
```

RECURSION IN JAVASCRIPT

A recursive function is a function that calls itself either directly or on directly through another function.

The following recursion example in java script shows the evaluation for the factorial of n. a recursive definition of the factorial function is arrived by observation that is N!=N*(N-1)!

Example : Write a program to display the factorial of a given number.

```
<html>
<head>
<title>recursion for factorial</title>
<script language="javascript">
   var a =window.prompt("enter number for factorial:","0"); var
num=parseInt(a);
document.writeln("factorial of "+num+ "is" +factorial(num)); function
factorial(num)
{
if(num<=1)
```

```
return 1;
else
return num*factorial(num-1);
}
</script>
</head>
</html>
```

ITERATION: if a function or structure executes by following a repetition structure mechanism it is called iterative process. It terminates when a loop continuation condition fails.

Common features of recursion and iteration:

- ❖ Both are based on a control structure.
- ❖ Both involve repetition.
- ❖ Both involve a termination test.
- ❖ Both can occur infinitely.

Recursion VS iteration:

S.NO	RECURSION	ITERATION
1	Recursion uses selection structure(such as if, if/else or switch)	Iteration uses a repetition structure(such as for, while or do-while)
2	Recursion achieves repetition through repeated function calls	Iteration explicitily uses the repetition structure.
3	Recursion terminates when a base case is recognized	Iteration terminates when the loop continuation condition fails.
4.	Recursion keeps producing simpler versions of original problem until	Iteration keeps modifying the counter until the counter assumes
	the base case is reached	a value that makes the loop continuation condition fail.
5.	Infinite recursion occurs if the recursion step does not reduce the problem each time.	An infinite loop occurs if the loop continuation test never becomes false.
6.	Recursion can be expensive in both processor time and memory space.	Iteration is less expensive in both processor time and memory space.
7	Recursion consumes considerable memory	Iteration consumes considerable memory

Example :Write java script that three integers from the user and outputs their sum, average, largest use alert dialog box to display results.

Use four different functions for accepting data, finding sum, average calculation and to find largest among them every time display the data along with output. Use four buttons to call the respective functions.

```
<html>
<head>
<title> numbers</title>
<script language="javascript">
var a,b,c,data;
var aNUM,bNUM,cNUM;
function acc()
{
a=window.prompt("enter first number");
b=window.prompt("enter second number");
```

```
c=window.prompt("enter third number");
aNUM=parseInt(a);
bNUM=parseInt(b);
cNUM=parseInt(c);
data=a+" ," +b +" ," +c;
}
function sum()
{
Result=aNUM+bNUM+cNUM;
window.alert("sum of "+data+" numbers is "+Result);
}
function avg()
{
    Avg=(aNUM+bNUM+cNUM)/3;
window.alert("average of "+data+"number is "+Avg);
}
function largest()
{
large=Math.max(Math.max(aNUM,bNUM),cNUM);
window.alert("largest of " +data +"numbers" +large);
}
</script>
</head>
<body bg color="tan" text="black">
<form>
<table border="2">
<caption><h3>sum,average and largest</h3></caption>
<colgroup>
<col span="2" ALIGN="right">
</colgroup>
<tr><th>to accept numbers<td><input type="button" value="accept" onclick="acc()">
</tr>
<tr><th> to sum of those numbers<td><input type="button" value="sum"
onclick="sum()"></tr>
<tr><th>to get average of those numbers<td><input type="button" value="average"
onclick="avg()"></tr>
<tr><th>to get largest among those numbers<td><input type="button" value="largest"
onclick="largest()"></tr>
</table>
</form>
</body>
</html>
```

Example: Write a program for word equivalent of given number.

```
<html>
<head>
<title>words transformation</title>
<script language="javascript">
function dispwords()
{
var n=0,k=0,i=0,temp=0,r=" ";
var ar=new Array();
n=parseInt(document.f1.num.value);
temp=n;
while(temp>0)
```

```
{
k=temp%10;
ar[i++]=words(k);
temp=Math.floor(temp/10);
}
ar.reverse();
r=ar.join(" ");
document.f1.res.value=r;
}
function words(k)
{
switch(k)
{
case 0:return"ZERO";
case 1:return"ONE";
case 2:return"TWO";
case 3:return"THREE";
case 4:return"FOUR";
case 5:return"FIVE";
case 6:return"SIX"; case
7:return"SEVEN"; case
8:return"EIGHT"; case
9:return"NINE";
}
}
</script>
</head>
<body bgcolor=TAN>
<form name=f1>
Number <input type=text name=num>
<p>
Result<input type=text size=40 name=res>
<p>
<input type=button value="words" onclick="dispwords()">
</form>
</body>
</html>
```

Functions in Java script:

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure -a set of statements that performs a specific task when called. A function definition has these basic parts:

- The *function* keyword
- A function name
- A comma-separated list of arguments to the function in parentheses

- The statements in the function in curly braces: { }

Defining a Function

Defining the function means, name the function and specifies what to do when the function is called. You define a function within the <SCRIPT>...</SCRIPT> tags within the <HEAD> ... </HEAD> tags. While defining a function, you can also declare the variables which you will be calling in that function. Here's an example of *defining* a function:

```
function msg()
{
    window.alert("This is an alert box.");
}
```

Here's an example of a function that takes a parameter:

```
function welcome(string)
{
    window.alert("Hi"+string);
}
```

When you call this function, you need to pass a parameter (such as the word that the user clicked on) into thefunction.

Calling a Function

Calling the function actually performs the specified actions. When you call a function, this is usually within the BODY of the HTML page, and you usually pass a parameter into the function on which the function will act.

Here's an example of calling the same function: *msg();*

For the other example, this is how you may call it:

```
<inupt type="button" name="welcome" onClick="msg1("Vijay")"/>
```

Function is a piece of code that performs specific talk.

Functions are two types 1. Library Functions2.User defined Functions

Library Functions in Java script:

1. Eval: This function takes a string representing java script code to execute. The interpreter evaluates the code and executes dynamically.
2. isFinite: This function takes a numeric value and returns true if the argument results a finite numeric. Otherwise it returns false.
3. isNaN: This function takes a numeric argument and returns true if the argument is not a number otherwise returns false.
4. parseFloat: It accepts a string as argument and converts into its equivalent float value. If the conversion fails then a value NaN is returned.
5. parseInt: It accepts a string as argument and converts into its equivalent numeric. If the conversion fails then a value NaN is returned.

Example: Write a program to accept a number from user and display whether it is Armstrong or not. Before testing for Armstrong, check user has entered number or not.

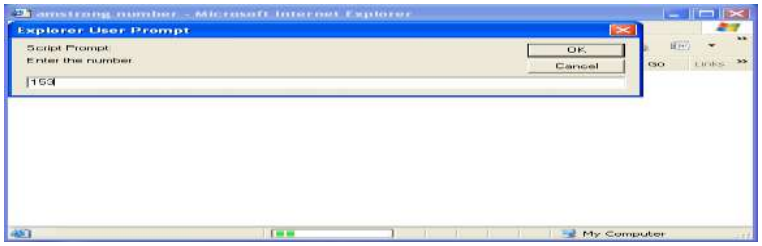
For Example n=153

$1^3+5^3+3^3=1+125+27=153=x$

x=n Armstrong x!=n not Armstrong

```
<html>
<head>
<title>amstrong number</title>
<script language="javascript">
var n,r,x,sum=0;
do
{
 n=parseInt(window.prompt("Enter the number","0"));
 }
```

```
while(isNaN(n));
x=n;
while(n!=0)
{
 r=n%10;
 sum=sum+r*r*r;
 n=parseInt(n/10);
}
if(x==sum)
        window.alert("N is Amstrong Number")
        else
        window.alert("N is Not amstrong Number");
</script>
</head>
</html>
```



User-Defined Functions:

Divide and conquer can be achieved in Javascript by using functions. Functions that you create start with the command "function" and are followed by the name for the function. A function can be defined as follows:

```
Function function-name(parameter-list)
{
        Declerations;
                Statements;
        - - - - - - -
        - - - - - - -

        return[expression];
}
```

Example: Write a program to display square and cube of a given number by using user-defined functions.

```
<html>
<head>
        <title>Square and cube of given number</title>
        <script language="JavaScript">
        var a;
        a=parseInt(window.prompt("Enter number","0"));
        document.writeln("Square of given number is
<b>"+square(a)+"</b><br>");
```

```
            document.writeln("Cube of given number is <b>"+cube(a)+"</b>");
            function square(k)
            {
               return k*k;
            }
            function cube(k)
            {
               return k*k*k;
}
</script>
</head>
</html>
```

Pattern matching with Regular Expressions

A regular expression is a sequence of characters that forms a **search pattern**.
When you search for data in a text, you can use this search pattern to describe what you are searching for.
A regular expression can be a single character, or a more complicated pattern.
Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Syntax

/*pattern*/*modifiers*;

Example

var patt = /wtclass/i;
Example explained:
/wtclas**/i** is a regular expression.
wtclas is a pattern (to be used in a search).
i is a modifier (modifies the search to be case-insensitive).
A regular expression is an object that describes a pattern of characters. The JavaScript RegExp class represents regular expressions, and both String and *RegExp* define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on the text.
A regular expression could be defined with the *RegExp ()* constructor, as follows −

```
var pattern = new RegExp(pattern, attributes);
or simply
var pattern = /pattern/attributes;
```

The following are the parameters −

- **pattern** − A string that specifies the pattern of the regular expression or another regular expression.
- **attributes** − An optional string containing any of the "g", "i", and "m" attributes that specify global, case-insensitive, and multiline matches, respectively.

Brackets

Brackets ([]) have a special meaning when used in the context of regular expressions. They are used to find a range of characters.

Sr.No.	Expression & Description

1	**[...]** Any one character between the brackets.
2	**[^...]** Any one character not between the brackets.
3	**[0-9]** It matches any decimal digit from 0 through 9.
4	**[a-z]** It matches any character from lowercase **a** through lowercase **z**.
5	**[A-Z]** It matches any character from uppercase **A** through uppercase **Z**.
6	**[a-Z]** It matches any character from lowercase **a** through uppercase **Z**.

The ranges shown above are general; you could also use the range [0-3] to match any decimal digit ranging from 0 through 3, or the range [b-v] to match any lowercase character ranging from **b** through **v**.

Quantifiers

The frequency or position of bracketed character sequences and single characters can be denoted by a special character. Each special character has a specific connotation. The +, *, ?, and $ flags all follow a character sequence.

Sr.No.	Expression & Description
1	**p+** It matches any string containing one or more p's.
2	**p*** It matches any string containing zero or more p's.
3	**p?** It matches any string containing at most one p.
4	**p{N}** It matches any string containing a sequence of **N** p's
5	**p{2,3}** It matches any string containing a sequence of two or three p's.
6	**p{2, }** It matches any string containing a sequence of at least two p's.
7	**p$** It matches any string with p at the end of it.
8	**^p** It matches any string with p at the beginning of it.

Examples

Following examples explain more about matching characters.

Sr.No.	Expression & Description

1	**[^a-zA-Z]** It matches any string not containing any of the characters ranging from **a** through **z** and **A** through Z.
2	**p.p** It matches any string containing **p,** followed by any character, in turn followed by another **p**
3	**^.{2}$** It matches any string containing exactly two characters.
4	**(.*)** It matches any string enclosed within and .
5	**p(hp)*** It matches any string containing a **p** followed by zero or more instances of the sequence **hp**.

Using String Methods

In JavaScript, regular expressions are often used with the two **string methods**:

search() and replace().

The search() method uses an expression to search for a match, and returns the position of the match.

The replace() method returns a modified string where the pattern is replaced.

Using String search() With a String

The search() method searches a string for a specified value and returns the position of the match:

Example

Use a string to do a search for "wtclass" in a string:

```
var str = "Visit wtclass!";
var n = str.search("wtclass");
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript String Methods</h2>
<p>Search a string for "W3Schools", and display the position of the match:</p>
<p id="demo"></p>
<script>
var str = "Visit wtclass ";
var n = str.search("wtclass");
document.getElementById("demo").innerHTML = n;
</script>
</body>
</html>
```

Using String search() With a Regular Expression

Example

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

```
var str = "Visit wtclass";
var n = str.search(/wtclass/i);
```

The result in *n* will be: 6

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Regular Expressions</h2>
<p>Search a string for "w3Schools", and display the position of the match:</p>
<p id="demo"></p>
```

```
<script>
var str = "Visit wtclass!";
var n = str.search(/Wtclass/i);
document.getElementById("demo").innerHTML = n;
</script>
</body>
</html>
```

Using String replace() With a String

he replace() method replaces a specified value with another value in a string:

```
var str = "Visit Microsoft!";
var res = str.replace("Microsoft", "WT CLASS");
```

Try it Yourself »

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript String Methods</h2>
<p>Replace "Microsoft" with "W3Schools" in the paragraph below:</p>

<button onclick="myFunction()">Try it</button>
<p id="demo">Please visit Microsoft!</p>
<script>
function myFunction() {
  var str = document.getElementById("demo").innerHTML;
  var txt = str.replace("Microsoft","WT CLASS");
  document.getElementById("demo").innerHTML = txt;
}
</script>
</body>
</html>
```

Use String replace() With a Regular Expression

Example

Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

```
var str = "Visit Microsoft!";
var res = str.replace(/microsoft/i, "WT CLASS");
```

The result in *res* will be:

Visit Wt class!

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Regular Expressions</h2>

<p>Replace "microsoft" with "W3Schools" in the paragraph below:</p>
<button onclick="myFunction()">Try it</button>
<p id="demo">Please visit Microsoft and Microsoft!</p>
<script>
function myFunction() {
  var str = document.getElementById("demo").innerHTML;
  var txt = str.replace(/microsoft/i,"WT CLASS");
  document.getElementById("demo").innerHTML = txt;
}
</script>
</body>
</html>
```

Regular Expression Modifiers

Modifiers can be used to perform case-insensitive more global searches:

Modifier	Description

i	**Perform case-insensitive matching**
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a case-insensitive search for "wtclass"
	in a string.
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	var str = "Visit wtclass";
	var patt1 = / wtclass/i;
	var result = str.match(patt1);
	document.getElementById("demo").innerHTML = result;
	}
	</script>
	</body>
	</html>

g	**Perform a global match (find all matches rather than s topping after the first match)**
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a global search for "is" in a string.
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	var str = "Is this all there is?";
	var patt1 = /is/g;
	var result = str.match(patt1);
	document.getElementById("demo").innerHTML = result;
	}
	</script>
	</body>
	</html>

m	**Perform multiline matching**
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a multiline search for "is"
	at the beginning of each line in a string.
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	var str = "\nIs th\nis it?";
	var patt1 = /^is/m;
	var result = str.match(patt1);

```
 document.getElementById("demo").innerHTML = result;
}
</script>
</body>
</html>
```

Regular Expression Patterns

Brackets are used to find a range of characters:

Expression	Description

[abc]	**Find any of the characters between the brackets**
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a global search for the character "h" in a string.
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	  var str = "Is this all there is?";
	  var patt1 = /[h]/g;
	  var result = str.match(patt1);
	  document.getElementById("demo").innerHTML = result;
	}
	</script>
	</body>
	</html>

[0-9]	Find any of the digits between the brackets
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a global search for the numbers 1 to 4 in a string.
	
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	  var str = "123456789";
	  var patt1 = /[1-4]/g;
	  var result = str.match(patt1);
	  document.getElementById("demo").innerHTML = result;
	}
	</script>
	</body>
	</html>

(x|y)	Find any of the alternatives separated with | <!DOCTYPE html> <html> <body> Click the button to do a global search for any of the specified alternatives (red|green). <button onclick="myFunction()">Try it</button>

```
<p id="demo"></p>
<script>
function myFunction() {
  var str = "re, green, red, green, gren, gr, blue, yellow";
  var patt1 = /(red|green)/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>
</body>
</html>
```

Metacharacters are characters with a special meaning:

Metacharacter	Description

\d	Find a digit
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a global search for digits in a string.
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	  var str = "Give 100%!";
	  var patt1 = /\d/g;
	  var result = str.match(patt1);
	  document.getElementById("demo").innerHTML = result;
	}
	</script>
	</body>
	</html>

\s	Find a whitespace character
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a global search for whitespace characters in a string.
	
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	  var str = "Is this all there is?";
	  var patt1 = /\s/g;
	  var result = str.match(patt1);
	  document.getElementById("demo").innerHTML = result;
	}
	</script>
	</body>
	</html>

\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b

	example 1:
	<!DOCTYPE html>
	<html>
	<body>
	
	Search for the characters
	
	"LO"
	
	in the beginning of a word in the phrase:
	
	"HELLO, LOOK AT YOU!"
	
	 Found in position:
	<script>
	var str = "HELLO, LOOK AT YOU!";
	var patt1 = /\bLO/;
	var result = str.search(patt1);
	document.getElementById("demo").innerHTML = result;
	</script>
	</body>
	</html>
	Example 2:
	<!DOCTYPE html>
	<html>
	<body>
	
	Search for the characters
	
	"LO"
	
	in the end of a word in the phrase:
	
	"HELLO, LOOK AT YOU!"
	
	 Found in position:
	<script>
	var str = "HELLO, LOOK AT YOU!";
	var patt1 = /LO\b/;
	var result = str.search(patt1);
	document.getElementById("demo").innerHTML = result;
	</script>
	</body>
	</html>

\uxxxx	Find the Unicode character specified by the hexadecimal
	number xxxx
	<!DOCTYPE html>
	<html>
	<body>
	 Click the button to do a global search for the hexadecimal number 0057
	(W) in a string.
	<button onclick="myFunction()">Try it</button>
	
	<script>
	function myFunction() {
	var str = "Visit W3Schools. Hello World!";
	var patt1 = /\u0057/g;
	var result = str.match(patt1);
	document.getElementById("demo").innerHTML = result;

```
}
</script>
</body>
</html>
```

Quantifiers define quantities:

Quantifier	Description

n+	Matches any string that contains at least one *n*
	`<!DOCTYPE html>`
	`<html>`
	`<body>`
	`<p>Click the button to do a global search for at least one "o" in a string.</p>`
	`<button onclick="myFunction()">Try it</button>`
	`<p id="demo"></p>`
	`<script>`
	`function myFunction() {`
	`  var str = "Hellooo World! Hello W3Schools!";`
	`  var patt1 = /o+/g;`
	`  var result = str.match(patt1);`
	`  document.getElementById("demo").innerHTML = result;`
	`}`
	`</script>`
	`</body>`
	`</html>`

n*	Matches any string that contains zero or more occurrences of *n*
	`<!DOCTYPE html>`
	`<html>`
	`<body>`
	`<p>Click the button to do a global search for an "l", followed by zero or more "o" characters.</p>`
	`<button onclick="myFunction()">Try it</button>`
	`<p id="demo"></p>`
	`<script>`
	`function myFunction() {`
	`  var str = "Hellooo World! Hello W3Schools!";`
	`  var patt1 = /lo*/g;`
	`  var result = str.match(patt1);`
	`  document.getElementById("demo").innerHTML = result;`
	`}`
	`</script>`
	`</body>`
	`</html>`

n?	Matches any string that contains zero or one occurrences of *n*
	`<!DOCTYPE html>`
	`<html>`
	`<body>`
	`<p>Click the button to do a global search for a "1", followed by zero or one "0" characters.</p>`
	`<button onclick="myFunction()">Try it</button>`
	`<p id="demo"></p>`
	`<script>`

```
function myFunction() {
  var str = "1, 100 or 1000?";
  var patt1 = /10?/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>
</body>
</html>
```

Using the RegExp Object

In JavaScript, the RegExp object is a regular expression object with predefined properties and methods.

Using test()

The test() method is a RegExp expression method.

It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":

Example

```
var patt = /e/;
patt.test("The best things in life are free!");
```

Since there is an "e" in the string, the output of the code above will be: true

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Regular Expressions</h2>
<p>Search for an "e" in the next paragraph:</p>
<p id="p01">The best things in life are free!</p>
<p id="demo"></p>
<script>
text = document.getElementById("p01").innerHTML;
document.getElementById("demo").innerHTML = /e/.test(text);
</script>
</body>
</html>
```

You don't have to put the regular expression in a variable first. The two lines above can be shortened to one:

```
/e/.test("The best things in life are free!");
```

Using exec()

The exec() method is a RegExp expression method.

It searches a string for a specified pattern, and returns the found text as an object.

If no match is found, it returns an empty *(null)* object.

The following example searches a string for the character "e":

Example 1

```
/e/.exec("The best things in life are free!");
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Regular Expressions</h2>
<p id="demo"></p>
<script>
var obj = /e/.exec("The best things in life are free!");
document.getElementById("demo").innerHTML =
"Found " + obj[0] + " in position " + obj.index + " in the text: " + obj.input;
```

```
</script>
</body>
```

DYNAMIC HTML

DHTML is combination of HTML, CSS and JavaScript. It gives pleasant appearance to web page.

Difference between HTML and DHTML

HTML	DHTML
HTML is used to create static web pages.	DHTML is used to create dynamic web pages.
HTML is consists of simple html tags.	DHTML is made up of HTML tags+cascading style sheets+javascript.
Creation of html web pages is simplest but less interactive.	Creation of DHTML is complex but more interactive.

EVENT HANDLING:

Events are triggers that call one of your function. An event could be action Such as clicking on a button or placing your mouse over an image. for example we will use the unclick event for starting our form validation scripts, and the on mouse over event for creating graphics images that change when you place your cursor over them.

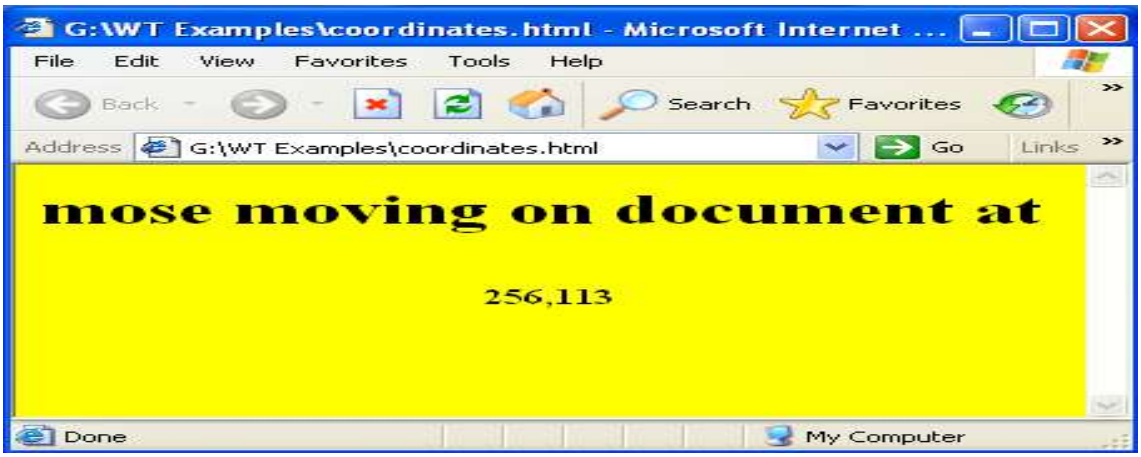
Advantages of events*:*

1. You can create user –interactive forms.
2. Validate the forms immediately when elements lost their focus
3. Display messages when some components receive focus
4. You can handle errors
5. Process the forms with help of submit button
6. You can write events when objects are selected
7. Also you can handle resize of windows or frames
8. Scripts can respond to user inter actions
9. Change the page according i.e. add dynamism to the page
10. It makes web applications more responsive and user-friendly]

Mouse Events*:*

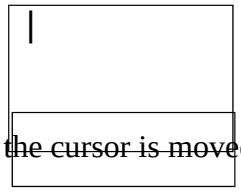
```
<HTML>
<head>
<script language="javascript">
function disp()
{
c.innerText=event.offsetX+","+event.offsetY;
}
</script>
</head>
<body bgcolor="yellow" onmousemove="disp()">
<h1>mose moving on document at</h1>
<center>
<br>
<b id=c></b>
</center>
</body>
```

</html>

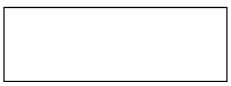


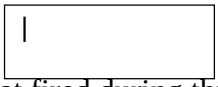
FOCUSING EVENTS:

Focus is nothing but the position of the cursor where it is placed .In forms, focus is moved between various elements on the form. For example, you have simple form as follows:



In the first field cursor is blinking, if you press tab key then the cursor is moved to second field and they appears as follows:



That means  first field is losing its focus .second field is gaining its focus. The events that fired during this are ONFOCUS and ONBLUR events.
ONFOCUS is fired when the element gained focus and ONBLUR event is fired when the element loses its focus. These events are very important if you want to handle the form elements. In general ONFOCUS event is used if user wants to display some messages when some particular graphical element receives focus. ONBLUR is used to verify given condition is satisfied after the user entered some data and pressed tab button. Following program is used to explain the concept of ONFOCUS and ONBLUR events.

Example :Write a program to display a form that accepts student name, age, father name. When age field receives its focus display message that age should be below 18 to 25. After losing its focus from age field verify user entered in between given values or not display respective message.

```
<html>
<head>
<script language="javascript">
function disp()
{
 document.f1.txt.value="Enter age between 18 to 25";
}
</script>
</head>
<body>
<form name="f1">
<h1>Data Entry Form</h1>
```

```
Student Name:<input type="text" name="sn">
<br>
Age:<input type="text" name="a" onfocus="disp()">
<br>
<input type="submit" value="submit">
<input type="reset" value="reset">
<input type=text name="txt" size=30 onkeyup="clear()"
style="position:absolute;left:300;top:90;background-color:tan;color:black;">
</form>
</body>
 </html>
```

Key events:

Sometimes you want to check keys that user pressed and according to that you want add dynamism to your web page. Java script provides many key events; with the help of them you can easily get these effects. Important

Key events that are supported by java script are

1. ONKEYDOWN
2. ONKEYPRESS
3. ONKEYUP

Key up event is used to restrict the user entry to a text field. Now the one more example is here. Numerical text fields, these fields accept only numerical values in to restrict the user to enter only numerical values into a text then he should depend on the key events. When a key is pressed, he should verify that the key is numeric or not. If is numeric display the value into the text field otherwise restrict that key.

For this we used the logic as follows

```
If   (s.length=1) {
---}
Else
  If (event.keycode<48|event. Keycode>58)
  Document. Form 1. Txt. Value=s. substring (0, s. length-1)

  If (event. Keycode<49|event. Keycode>58) Document.
   Form . txt. Value=``"";
```

First time when user presses other then number, then we clear the text field, after entering some numeric values and if he presses any alphabet we display up to previous length that is only numeric value. This can be achieved through parseInt() instead of substring().

Form processing and on change event:

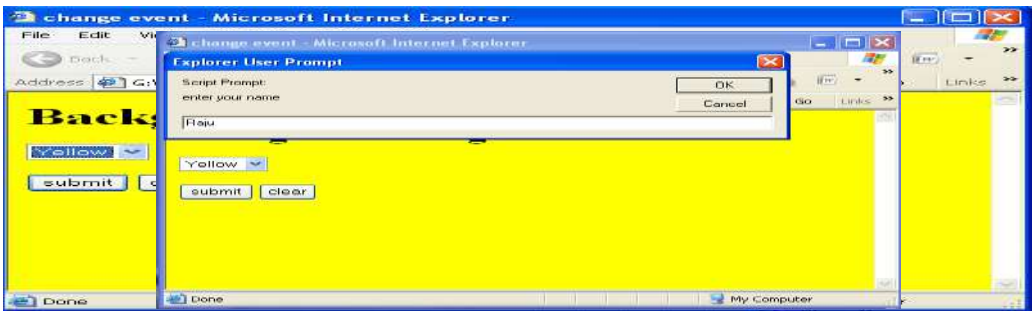
ONSUBMIT is invoked when the user submits the form. ONRESET is invoked when the user resets the form. If you want to cancel default action of the event then you can use window. Event. Return value=false.

ONCHANGE event is invoked when the user changed a value in the SELECT element, or text changed in a text field, it can be called as follows.

<SELECTID=``sel“““ onchange=change colar()“““>
If the options in the combo box are changed then this change color() method is called is called.

Example : write a program to change the background color when the user selected one of the colors from the combo box. When the user press submit button accept user name and display it.if he clicks on rest button display an alert message that he pressed resetting option.

```
<html>
<head>
<title>change of colors</title>
<script language=”java script”>
Function change color()
{
Document.body.style.backgroundcolor=document.form1.se1.value;
}
Function change text()
{
Var nam=window.prompt(“enter your name”,”name”);
s.inner text=”choose background color Mr/Ms :”+nam;
window.event.return value=false;
}
function say bye()
{
Window.alert(“now you are resetting “);
}
</script>
</head>
<boy bgcolor=tan>
<h1>BACKGROUND MAGIC</h1>
<form name=form1 onsubmit=change Text()  onreset=”saybye()”>
<h3 id=s>choose background color</h3>
<select id=”se1” onchange=”changecolor()”>
<option value=”red”>Red
<option value=”blue”>Blue
<option value=”yellow”>Yellow
<option value=”green”>Green
</select>
<input type=submit value=”submit”>
<input type=reset value=”reset”>
</form>
</body>
  </html>
```



Timer methods:

HTML elements can be modified according to code. JavaScript is having setInterval()method in window object that is used for running a function according to given time interval. For example window.setInterval("disp()",1000),the disp() method calls for every 1000 milliseconds. There is another method setTiout(),the specified function called after waiting number of milliseconds specified along with setTimeout() and clearInterval().,these are used for stopping the timeout timer or interval timer. These functions are needed because window.setInterval()program will run continuously ,to stop it we need clearInterval()method.

These two methods are used as follows.
To start timer T1.window.setInterval("disp()",100);
To stop Timer Window.clearInterval(t1);;

Example: write a program to display your name by increasing font sizes according to time intervals.

```
<html>
<head>
<title>Small animation</title>
<script language="javascript">
var t1;
var c=0;
function start()
{
        t1=window.setInterval("incr()",100);
}
function incr()
{
        c=c+10;
        t.style.fontSize=c;
        window.status=c;
        if(c==200)
                window.clearInterval(t1);
        //t.style.color=c;

}
</script>
</head>
<body bgcolor="yellow" onload="start()">
<center>
<p id="t" style="color:red">Raju</p>
</body>
</html>
```

COLLECTIONS:

Arrays of related object on a page can be called as collections dynamic HTML object model is contains various collections. Some of the collections are all, children, frames etc.

Collection all contains all the HTML elements of the current web page as all is an array of elements ,it contains the property length to get the number of tags on the web page; other property of this collection is tag name that is used to get the specified tag name. to display all the tag names of the current page, just run a for loop and add individual tag to a string.

There is a property inner text for html element; it refers to the text contained in that element property inner HTML is similar to inner Text, but it can include HTML formatting. The outer HTML property is similar to inner HTML property; it includes the enclosing HTML tags as well as the content inside them.

SUMMMARY OF COLLECTIONS:

Collection name	usage
all	All elements on the web page
children	Children elements for individual element
anchors	Collection<A name>tags
links	Collection<AHREF>tags
applets	Collection pf <APPLET>tags
embeds	Collection of <EMBED>tags
images	Collection of <IMG>tags
forms	Collection of FORMS
frames	Representing each frame
scripts	Collection contains all the <script>tags
stylesheets	Represent each style element

FILTERS AND TRANSITIONS:

Horizontal and vertical filters:

Fundamental filters supported by dynamic HTML are flipv and fliph popularly known as flip filters. These filters create mirror effects for the images or text. These are vertical in nature if filters is flipv and horizontal in the case of fliph. Simple effects are as follows:

In dynamic HTML creation of filter is very simple only thing you require is style sheets. In styles, you have a special style that is filter style, which is used for all the kinds of filters the value passed to filter is fliph for horizontal flip and flipv for vertical flip. If you require both filters just apply both of them.

Example:

Style=”filter:fliph”for horizontal flip

Style=”filter:flipV” for vertical flip

Style =”filter:fliph flipv” for both flips.

Example :

Write a program to display vertical flip for ramesh

```
<html>
<body bgcolor=tan>
```

```
<table>
<tr><th  size=40px;filter:flipv”>Ramesh</tr>
</table>
</body>
</html>
```

Example :

Write a DHTML and java script program to accept username and display the given name along with four buttons that contains horizontal flip, vertical flip, both flips and normal buttons with actions.

```
<html>
<head>
<script language="javascript">
var k;
function change(k)
{
var k;
        switch(k)
        {
                case 1:a.style.filter='fliph';break;
                case 2:a.style.filter='flipv';break;
                case 3:a.style.filter='fliph flipv';break;
                case 4:a.style.filter='';
        }
}
</script>
</head>
<body onload="a.innerText=window.prompt('Your Name')">
<form name="f1">
<p id="a" align="center" style="background-color:tan;font-size:80px;">Your
Name</p>
<input type=button onClick="change(1)" value="horizontal">
<input type=button onClick="change(2)" value="vertical">
<input type=button onClick="change(3)" value="Horizontal&vertical">
<input type=button onClick="change(4)" value="Normal">
</form>
</body>
</html>
```

Masking:

Masking effect is obtained by using mask filters. These are created as image masks, with the help of these filters you can get an effect of background image colors applied on the specified text of foreground. Image mask means adding text to image with image colors. Foreground text is in transparent color so that background image can be displayed on the text.

These effects can be achieved in DHTML by using statements.

```
<h1 style=``position: absolute: top:95 ; left:280;filter:mask(color=#000000)‘‘‘‘>
Followed by an image behind this header so that you can get the
mask feel.
<img src=``anyimg Jpg‘‘‘‘width=‘‘‘‘400‘‘‘‘height=``200‘‘‘‘> Better to get both are
placed at one location.
```

Example :

Write a DHTML program to display a text on given image. That text should start form the image and moved outwards of the image, so that background color of webpage should apply to text.

```
<html>
<head>
<title>mask filter</title>
</head>
<body bgcolor=tan>
<h1>filter masking</h1>
<div style="position:absolute;top:95;left:280;filter:mask(color= #000000)">
<p style="font-size:50px">Web Tech</div>
<img src="D:\Raju\WT\winter.jpg" width=400 height=200>
</body>
</html>
```

Other filters:

Graphics software or photo editors contain an important feature for image enhancement. Those features are applying some negative image or gray scale effects to the picture. Automatically by clicking one of the menu options. DHTML also provides this feature by using filters. There are three important filters of that kind.

1. Invert filter: this applies a negative image effect to the given image, which means that drak areas became light and area became dark.
   Example: <img src=”could06.jpg” style=”filter: invert”>
2. Gray filter: this generates a grayscale image effect; in that all colors are stripped from the image and all that remains in brightness data. Example:<img src=”could06.jpg” style=”filter:gray”>
3. X-ray filter: this generates inversion of grayscale image effect that is called as x-ray effect.
   Example:<img src=”could06.jpg” style=”filter:xray”>

Example: Write a program that displays grayscale,invert and x-ray effects for the given image.

```
<html>
<head>
<title>image filters</title>
</head><body><table>
<tr><th>normal<th>GRAYSCALE</tr>
<tr> <td><img src="C:\Documents and Settings\All Users\Documents\My Pictures\Sample Pictures\Winter.jpg" width=300 height=200></td>
<td><img src="C:\Documents and Settings\All Users\Documents\My Pictures\Sample Pictures\Winter.jpg" width=300 height=200 style="filter:gray"></td></tr>
<tr><th>XRAY <th>INVERT</tr>
<tr><td><img src="C:\Documents and Settings\All Users\Documents\My Pictures\Sample
```

```
Pictures\Winter.jpg" width=300 height=200 style="filter:xray"></td>
<td><img src="C:\Documents and Settings\All Users\Documents\My Pictures\Sample
Pictures\Winter.jpg" width=300 height=200 style="filter:invert"></td>
</tr>
</table>
</body>
</html>
```

Shadow effects:

Now a day‟s normal word packages are also providing facilities to add shadows to your text. For example a message “welcome” with shadow effect observes here.
But this slight difference we can‟t observe, but if this shadow is larger then we can easily identify it.to add some depth to your text then you can apply shadow filter to your text. A three –dimensional appearance can be observed to your text if you apply shadow effects shadow filters in DHTML takes two important parameters. One is direction that specifies in which direction you want to display your shadow and other parameters is color.

Tells the color of the shadow. For example, for letter I that shadow appears left side as follows.
As exactly may be you cannot get with simple shadoweffects in DHTML, but you can decided which color you want and which directionthat shadow can be displayed you can fix it.

```
<p style=”color:blue; font-size:75;position:absolute;
Filter:shadow(direction=50,color=black)”>ramesh
```

Then the shadow of text ramesh appears 50 degrees, means above right side and display in black color.actual text color is blue.
The shadow directions are 0-top, 45-above right,90-right,135-below right,180-below,225-below left,270-left,315-above left.

Example : Write a DHTML program that displays shadow for a text.

```
<HTML>
<head>
<title>shadow filter</title>
</head>
<body bgcolor=tan>
<p style="color:white;font-
size:75;position:absolute;top:25;left:25;padding:10;filter:shadow(direction=315,color=red)">WT
<p style="color:green;font-
size:75;position:absolute;top:25;left:300;padding:10;filter:shadow(direction=150,color=black)">WT
<p style="color:blue;font-
```

```
size:75;position:absolute;top:150;left:25;padding:10;filter:shadow(direction=150,color=black)">WT
<p style="color:yellow;font-
size:75;position:absolute;top:150;left:300;padding:10;filter:shadow(direction=150,color=blue)">WT
</body>
</html>
```

Note: But the above code will be supported for the lower versions of the browsers; higher version of the browsers will support the following code

```
<HTML>
<head>
<title>shadow filter</title>
</head>
<body bgcolor=tan>
<p style="color:white;font-size:75;position:absolute;top:25;left:25;padding:10;text-
shadow:2px 2px 0 red">SACET</p>
<p style="color:green;font-size:75;position:absolute;top:25;left:300;padding:10;box-
shadow: 5px 5px 10 black">SACET</p>
<p style="color:blue;font-size:75;position:absolute;top:150;left:25;padding:10;text-
shadow:5px 3px 2 white">SACET</p>
<p style="color:yellow;font-
size:75;position:absolute;top:150;left:300;padding:10;text- shadow:4px 5px 3
red">SACET</p>
</body>
</html>
```

Gradient effects

One of the beautiful effect of image graphics is to display an image starting with a color and ending with the complete picture. This can be called as gradient effects. The alpha filter is used to create the above said gradient effect. Alpha filter is as follows

Filter: alpha (style=2, opacity=100, finishopacity=0)">

It takes three important parameters indicates how the gradient transforms, there are four options are available for this.

0-uniform
1-linear
2-circular
3-rectangular

Other two parameters are opacity indicates the percentage at what percent at what opacity the specified gradient starts and other one finish opacity indicates where it finishes

Pic.filters("alpha")opacity=0;
Pic.filters("alpha").finish opacity=100;

Indicates that first initial percent of opacity starts with 0 and finished it after achieving complete opacity.

Example 25:

Write a simple program that explains the concept of alpha gradient.

```
<html>
```

```
<head>
<title>alpha filter</title>
</head>
<body>
<div id="pic"
style="position:absolute;filter:alpha(style=2,opacity=100,finishopacity=0)">
<img width=300 height=200 src="D:\Raju\WT\winter.jpg">
</div>
</body>
</html>
```

Note: Present versions of the browsers supports following syntax

All Filters

A demonstration of all filter functions:

```
.blur {
   -webkit-filter: blur(4px);
   filter: blur(4px);
}

.brightness {
   -webkit-filter: brightness(0.30);
   filter: brightness(0.30);
}

.contrast {
   -webkit-filter: contrast(180%);
   filter: contrast(180%);
}

.grayscale {
   -webkit-filter: grayscale(100%);
   filter: grayscale(100%);
}

.huerotate {
   -webkit-filter: hue-rotate(180deg);
   filter: hue-rotate(180deg);
}
```

```
.invert {
   -webkit-filter: invert(100%);
   filter: invert(100%);
}

.opacity {
   -webkit-filter: opacity(50%);
   filter: opacity(50%);
}

.saturate {
   -webkit-filter: saturate(7);
   filter: saturate(7);
}

.sepia {
   -webkit-filter: sepia(100%);
   filter: sepia(100%);
}

.shadow {
   -webkit-filter: drop-shadow(8px 8px 10px green);
   filter: drop-shadow(8px 8px 10px green);
}
```

PREVIOUS QUESTION PAPERS

1a)Explain the following terms with examples [7]
(i) Absolute Positioning
(ii) Relative Positioning.
b) Differentiate between Implicit and Explicit type conversion. [7]

2a)State the two approaches of Pattern matching used in JavaScript. [7]
b)How Object creation and Modification done in JavaScript? [7]

3 a) Explain about Control statements used in JavaScript with examples. [7]
b) Discuss about Moving Elements in JavaScript. [7]

4. a) Explain about the following terms: [7]
(i) The Math Object
(ii) The Number Object
b) Illustrate the concept of Screen Output and Keyboard Input. [7]

UNIT-III

AJAX

AJAX A New Approach

- Introduction to AJAX
- Integrating PHP and AJAX.

AJAX (Asynchronous JavaScript and XML)

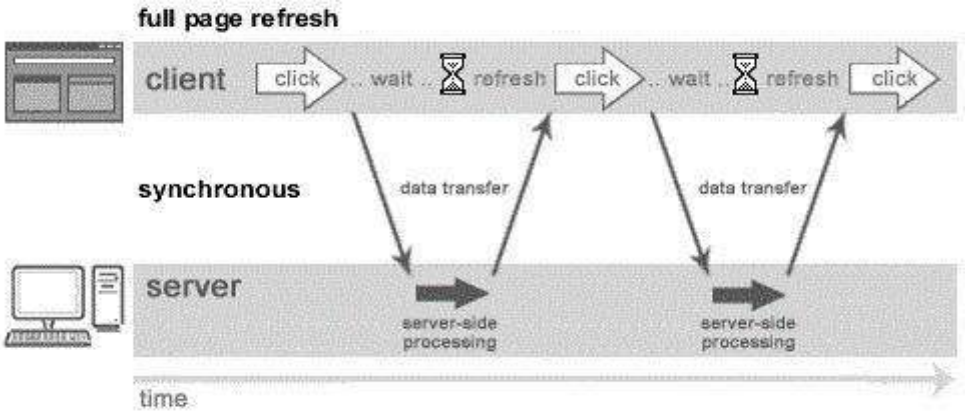
AJAX is an acronym for **Asynchronous JavaScript and XML**. It is a group of inter-related technologies like javascript, dom, xml, html, css etc. AJAX allows you to send and receive data asynchronously without reloading the entire web page. So it is fast.

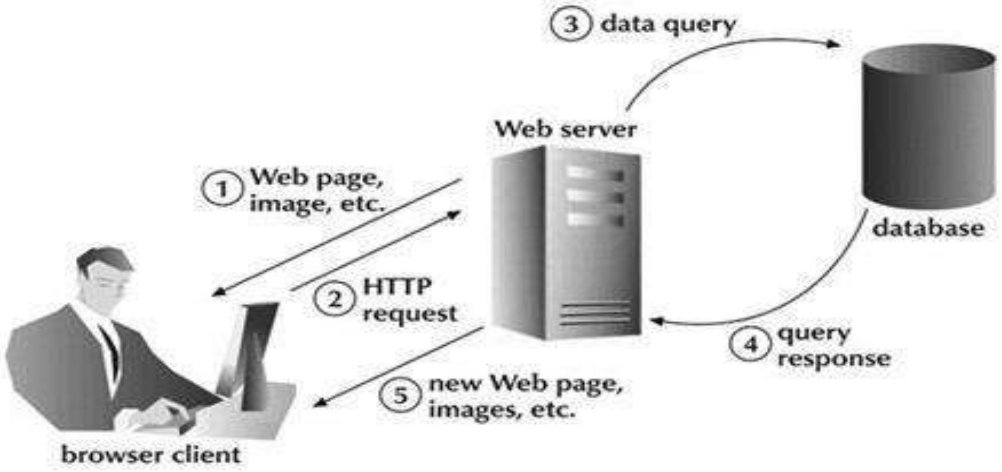
AJAX allows you to send only important information to the server not the entire page. So only valuable data from the client side is routed to the server side. It makes your application interactive and faster.

Where it is used?

There are too many web applications running on the web that are using AJAX Technology. Some are:

1. Gmail
2. Facebook
3. Twitter
4. Google maps
5. YouTube etc.,





Synchronous Vs. Asynchronous Application

Before understanding AJAX, let's understand classic web application model and AJAX Web application model.

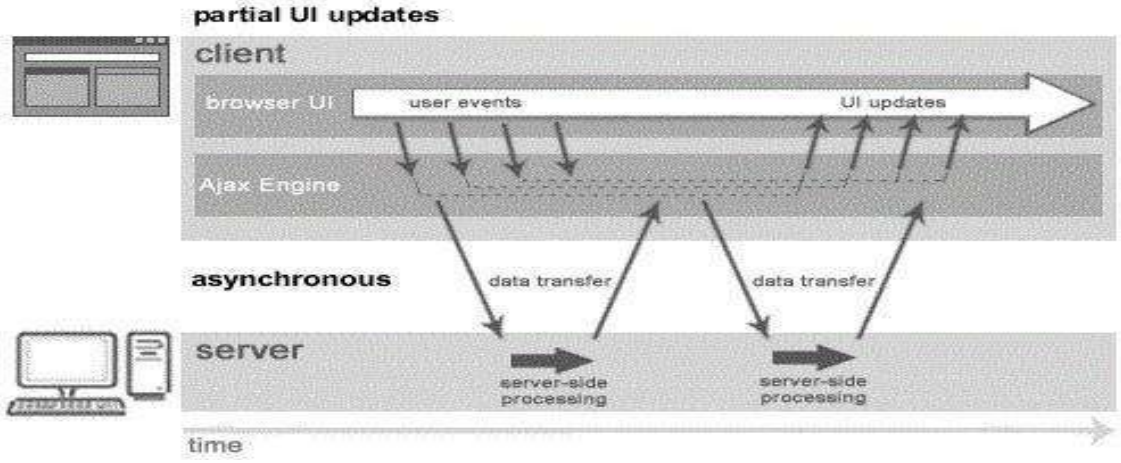
- **Synchronous (Classic Web-Application Model)**

  A synchronous request blocks the client until operation completes i.e. browser is not unresponsive. In such case, JavaScript Engine of the browser is blocked.

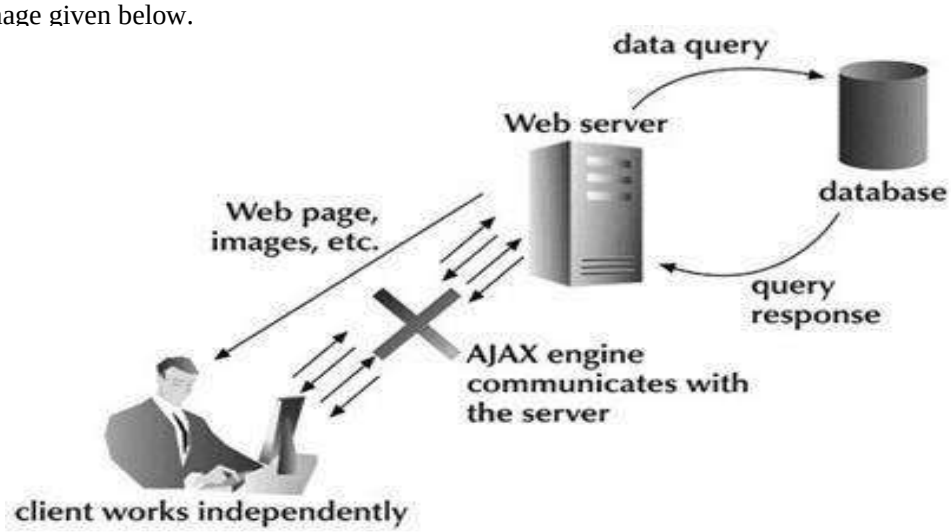
  As you can see in the above image, full page is refreshed at request time and user is blocked until request completes. Let's understand it another way.

- **Asynchronous (AJAX Web-Application Model)**

  An asynchronous request doesn't block the client i.e. browser is responsive. At that time, user can perform other operations also. In such case, JavaScript Engine of the browser is not blocked.



As you can see in the above image, full page is not refreshed at request time and user gets response from the AJAX Engine. Let's try to understand asynchronous communication by the image given below.



AJAX Technologies

AJAX is not a Technology but group of inter-related technologies. AJAX Technologies includes:

- HTML/XHTML and CSS
- DOM
- XML or JSON(JavaScript Object Notation)
- XMLHttpRequest
- JavaScript

- **HTML/XHTML and CSS**
  These technologies are used for displaying content and style. It is mainly used for presentation.

- **DOM**
  It is used for dynamic display and interaction with data.

- **XML or JSON**
  For carrying data to and from server. JSON is like XML but short and faster than XML.

- XMLHttpRequest
  For asynchronous communication between client and server.

- JavaScript
  It is used to bring above technologies together. Independently, it is used mainly for client-side validation.

Understanding XMLHttpRequest

An object of XMLHttpRequest is used for asynchronous communication between client and server. It performs following operations:

1. Sends data from the client in the background
2. Receives the data from the server
3. Updates the webpage without reloading it.

- **Properties of XMLHttpRequest object:**

Property	Description
onReadyStateChange	It is called whenever readystate attribute changes. It must not be used with synchronous requests.
readyState	Represents the state of the request. It ranges from 0 to 4. **0** UNOPENED open() is not called. **1** OPENED open is called but send() is not called. **2** HEADERS_RECEIVED send() is called, and headers and status are available. **3** LOADING Downloading data; responseText holds the data. **4** DONE The operation is completed fully.
reponseText	Returns response as TEXT.
responseXML	Returns response as XML

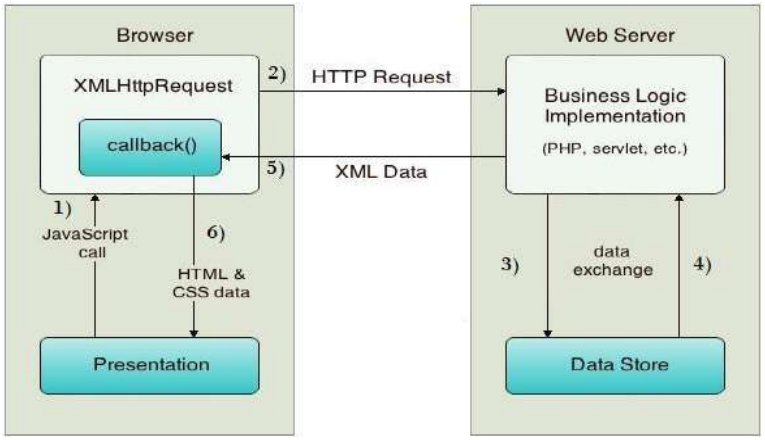
- **Methods of XMLHttpRequest object**

Method	Description
void open(method, URL)	Opens the request specifying get or post method and url.
void open(method, URL, async)	Same as above but specifies asynchronous or not.
void open(method, URL, async, username, password)	Same as above but specifies username and password.
void send()	Sends GET request.
void send(string)	Sends POST request.
setRequestHeader(header,value)	It adds request headers.

How AJAX Works?

AJAX communicates with the server using XMLHttpRequest object. Let's understand the flow of AJAX with the following figure:

1. User sends a request from the UI and a javascript call goes to XMLHttpRequest object.
2. HTTP Request is sent to the server by XMLHttpRequest object.
3. Server interacts with the database using JSP, PHP, Servlet, ASP.net etc.
4. Data is retrieved.
5. Server sends XML data or JSON data to the XMLHttpRequest callback function.
6. HTML and CSS data is displayed on the browser.



Introduction to Web Services

Technology keep on changing, users were forces to learn new application on continuous basis. With internet, focus is shifting to-wards services based software. Users may access these services using wide range of devices such as PDAs, mobile phones, desktop computers etc. Service oriented software development is possible using man known techniques such as COM, CORBA, RMI, JINI, RPC etc. some of them are capable of delivering services over web & some or not. Most of these technologies uses particular protocols for communication & with no standardization. **Web service** is the concept of creating services that can be accessed over web. Most of these

What are Web Services?

A web services may be defines as: An application component accessible via standard web protocols. It is like unit of application logic. It provides services & data to remote clients & other applications. Remote clients & application access web services with internet protocols. They use XML for data transport & SOAP for using services. Accessing service is independent of implementation.
With component development model, web service must have following characteristics:

- Registration with lookup service
- Public interface for client to invoke service

Web services should also process following characteristics:

- It should use standard web protocols for communication
- It should be accessible over web
- It should support loose coupling between uncoupled distributed systems

Web services receive information from clients as messages, containing instructions about what client wants, similar to method calls with parameters. These message delivered by web services are encoded using XML.XML enabled web services are interoperable with other web services.

Web Service Technologies:

Wide variety of technologies supports web services. Following technologies are available for creation of web services. These are vendor neutral technologies. They are:

- Simple Object Access Protocol(SOAP)
- Web Services Description Language(WSDL)
- UDDI(Universal Description Discovery and Integration)

Simple Object Access Protocol (SOAP):

SOAP is a light weight & simple XML based protocol. It enables exchange of structured & typed information on web by describing messaging format for machine to machine communication. It also enables creation of web services based on open infrastructure. SOAP consists of three parts:

- **SOAP Envelope**: defines what is in message, who is the recipient, whether message is optional or mandatory
- **SOAP Encoding Rules**: defines set of rules for exchanging instances of application defined data types
- **SOAP RPC Representation**: defines convention for representing remote procedure calls & response

SOAP can be used in combination with variety of existing internet protocols & formats including HTTP, SMTP etc. Typical SOAP message is shown below:

```
<IVORY:Envelope xmlns:IVORY="http://schemas.xmlsoap.org/soap/envelope"
                IVORY:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
<IVORY:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
    <symbol>DIS</symbol>
    </m:GetLastTradePrice>
</IVORY:Body>
</IVORY:Envelope>
```

The consumer of web service creates SOAP message as above, embeds it in HTTP POST request & sends it to web service for processing:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
    ....
SOAP Message
....
```

The message now contains requested stock price. A typical returned SOAP message may look like following:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
SOAP-ENV:encodingStyle=" http://schemas.xmlsoap.org/soap/encoding" />
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <Price>34.5</Price>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Interoperability:

The major goal in design of SOAP was to allow for easy creation of interoperable distributed web services. Few details of SOAP specifications are open for interpretation; implementation may differ across different vendors. SOAP message though it is conformant XML message, may not strictly follow SOAP specification.

Implementations:

SOAP technology was developed by DevelopMentor, IBM, Lotus, Microsoft etc. More than 50 vendors have currently implemented SOAP. Most popular implementations are by Apache which is open

source java based implementation & by Microsoft in .NET platform. SOAP specification has been submitted to W3C, which is now working on new specifications called XMLP (XML Protocol)

SOAP Messages with Attachments (SwA)

SOAP can send message with an attachment containing of another document or image etc. On Internet, GIF, JPEG data formats are treated as standards for image transmission. Second iteration of SOAP specification allowed for attachments to be combined with SOAP message by using multipart MIME structure. This multi part structure is called as **SOAP Message Package**. This new specification was developed by HP & Microsoft. Sample SOAP message attachment is shown here:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
type=text/xml; start="<myimagedoc.xml@mystie.com>"
Content-Description: This is the optional message description.
--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <myimagedoc.xml@mysite.com>
<?xmll version="1.0"?>
<SOAP-ENV: Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
<SOAP-ENV:Body>
        ...
        <theSignedForm href="cid:myimage.tiff@mysite.com" />
        ...
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
        --MIME_boundary
        Content-Type: image/tiff
        Content-Transfer-Encoding: binary
        Content-ID: <myimagedoc.xml@mysite.com>
        ...binary TIFF image...
        --MIME_boundary--
```

Web Services Description Language (WSDL)

WSDL is an XML format for describing web service interface. WSDL file defines set of operations permitted on the server & format that client must follow while requesting service. WSDL file acts like contract between client & service for effective communication between two parties. Client has to request service by sending well formed & conformant SOAP request.

If we are creating web service that offered latest stock quotes, we need to create WSDL file on server that describes service. Client obtains copy of this file, understand contract, create SOAP request based on contract & dispatch request to server using HTTP post. Server validates the request, if found valid executes request. The result which is latest stock price for requested symbol is then returned to client as SOAP response.

WSDL Document:

WSDL document is an XML document that contains of set of definitions. First we declare name spaces required by schema definition:

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
targetNameSpace=http://schemas.xmlsoap.org/wsdl/ elementFormDefault="qualified">
```

The root element is definitions as shown below:

```
<wsdl:defiinitions name="nmtoken"? targetNameSpace="uri"?>
        <import namespace="uri" location="uri"/>
<wsdl:documentation ..... />?
        ...
</wsdl:definitions>
```

The *name* attribute is optional & can serve as light weight form of documentation. The *nmtoken* represents name token that are qualified strings similar to CDATA, but character usage is limited to letters, digits, underscores, colons, periods & dashes. A *targetNamespace* may be specified by providing uri. The *import* tag may be used to associate namespace with document locations. Following code segment shows how declared namespace is associated with document location specified in *import* statement:

```
<definitions                                    name="StockQuote"
        targetNameSpace="http://example.com/stockquote/defiinitions"
        xmlns:tns="http://example.com/stockquote/definitions"
        xmlns:xsdl="http://example.com/stockquote/schemas"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
<import namespace="http://example.com/stockquote/schemas"
        Location="http://example.com/stockquote/stockquote.xsd"/>
```

Finally, optional *wsdl:documentation* element is used for declaring human readable documentation. The element may contain any arbitrary text. There are six major elements in document structure that describes service. These are as follows:

- **Types Element:** it provides definitions for data types used to describe how messages will exchange data. Syntax for types element is as follows:

```
<wsdl:types> ?
        <wsdl:documentation .../>
        <xsd:schema .../>
        <-- extensibility element -->
</wsdl:types>
```

The *wsdl:documentation* tag is optional as in case of *definitions*. The *xsd* type system may be used to define types in message. WSDL allows type systems to be added via extensibility element.

- **Message Element:** It represents abstract definition of data begin transmitted. Syntax for message element:

```
<wsdl:message name="nktoken"> *
        <wsdl;documentation .../>
            <part name="nmtoken" element="qname"? type="qname"? /> *
</wsdl:message>
```

The *message name* attribute is used for defining unique name for message with in document scope. The *wsdl:documentation* is optional & may be used for declaring human readable documentation. The message consists of one or more logical parts. The *part* describes logical abstract content of message. Each part consists of name & optional element & type attributes.\

- **Port Type Element:** It defines set of abstract operations. An operation consists of both input & output messages. The *operation* tag defines name of operation, *input* defines input for operation & *output* defines output format for result. The *fault* element is used for describing contents of SOAP fault details element. It specifies abstract message format for error messages that may be output as result of operation:

```
<wsdl:portType name="nmtoken">*
        <wsdl:documentation ..../>?
<wsdl:operation name="nmtoken">*
        <wsdl:documentation ..../>?
                <wsdl:input name="nmtoken"? message="qname">?
        <wsdl:documentation ..../>?
        </wsdl:input>
                <wsdl:output name="nmtoken"? message="qname">?
        <wsdl:documentation ..../>?
        </wsdl:output>
                <wsdl:fault name="nmtoken"? message="qname">?
```

```
                    <wsdl:documentation ..../>?
                </wsdl:fault>
            </wsdl:operation>
        </wsdl:portType>
```

- **Binding Element:** It defines protocol to be used & specifies data format for operations & messages defined by particular *portType*. The full syntax for binding is given below:

```
<wsdl:binding name="nmtoken" type="qname"> *
                    <wsdl:documentation ..../>?
                    <--Extensibility element -->*
                <wsdl:operation name="nmtoken">*
                    <wsdl:documentation ..../>?
                    <--Extensibility element -->*
                <wsdl:input> ?
                    <wsdl:documentation ..../>?
                    <--Extensibility element -->*
                </wsdl:input>
                <wsdl:output> ?
                    <wsdl:documentation ..../>?
                    <--Extensibility element -->*
                </wsdl:output>
                <wsdl:fault name="nmtoken"> *
                    <wsdl:documentation ..../>?
                    <--Extensibility element -->*
                </wsdl:fault>
                </wsdl:operation>
            </wsdl:binding>
```

The operation in WSDL file can be document oriented or remote procedure call (RPC) oriented. The style attribute of <*soap:binding*> element defines type of operation. If operation is document oriented, input & output messages will consist of XML documents. If operation is RPC oriented, input message contains operations input parameters & output message contains result of operation.

- **Port Element:** It defines individual end point by specifying single address for binding:

```
<wsdl:port name="nmtoken" binding="qname"> *
        <--Extensibility element (1) -->
</wsdl:port>
```

The *name* attribute defines unique name for port with current WSDL document. The *binding* attribute refers to binding & extensibility element is used to specify address information for port.

- **Service Element:** it aggregates set of related ports. Each port specifies address for binding:

```
<wsdl:service name="nmtoken"> *
                <wsdl:documentation ..../>?
            <wsdl:port name="nktoken" binding="qname"> *
                <wsdl:documentation .../> ?
                <--Extensibility element -->
            </wsdl:port>
                <--Extensibility element -->
        </wsdl:service>
```

Universal Description, Discovery & Integration (UDDI)

We need to publish web services so that customers & business partners can use the services. It requires common registry to register web service for clients to find it. For this several vendors including IBM, HP, Oracle, Sun Microsystem etc. formed an industry consortium known as UDDI. Today more than 250 companies have joined UDDI project. The main task of this project is to develop specifications for

web based business registry. The registry should be able to describe web service & allow others to discover registered web services.

UDDI allows any organization to publish information about its web services. The framework defines standard for businesses to share information, describe their services & their business & to decide what information is made public & what information is kept private. The interface is based on XML & SOAP, uses HTTP to interact with registry.

Registry itself holds information about business such as company name, contact etc. it holds both descriptive & technical information about web service. It provides search facilities that allow to search specific industry segment or geographic location.

Implementation:

This is global, public registry called UDDI business registry. It is possible for individuals to set up private UDDI registries. The implementations for creating private registries are available from IBM, Idoox etc. Microsoft has developed UDDI SDK that allows visual basic programmer to write program code to interact with UDDI registry. The use of SDK greatly simplifies interaction with registry & shields programmer from local level details of XML & SOAP.

Electronic Business XML (ebXML):

ebXML is set of specifications that allows businesses to collaborate. It enables global electronic market place where business can meet & tranasact with help of XML based messages. Business may be geographically located anywhere in world & could be of any size to participate in global marketplace. The framework defines specifications for sharing of web based business services. It includes specifications for message service, collaborative partner agreements, core components, business process methodology, registry & repository.

It defines registry & repository where business can register themselves by providing their contact information, address & so on. Such information is called Core Component. After business has registered with ebXML registry, other partners can look up registry to locate that business. Once business partner is located, the core components of located business are downloaded. Once buyer is satisfied with fact that seller service can meet its requirements, it negotiates contract with seller. Such collaborative partner agreements are defined in ebXML. Once both parties agree on contract terms, sign agreements & collaborative business transaction by exchanging their private documents. ebXML provides marketplace & defines several XML based documents for business to join & transact in such marketplace.

Intergrating PHP and AJAX

Example

To clearly illustrate how easy it is to access information from a database using Ajax and PHP, we are going to build MySQL queries on the fly and display the results on "ajax.html". But before we proceed, lets do ground work. Create a table using the following command.

NOTE − We are assuming you have sufficient privilege to perform following MySQL operations.

```
CREATE TABLE `ajax_example` (
  `name` varchar(50) NOT NULL,
  `age` int(11) NOT NULL,
  `sex` varchar(1) NOT NULL,
  `wpm` int(11) NOT NULL,
  PRIMARY KEY  (`name`)
)
```

Now dump the following data into this table using the following SQL statements.

```
INSERT INTO `ajax_example` VALUES ('Jerry', 120, 'm', 20);
INSERT INTO `ajax_example` VALUES ('Regis', 75, 'm', 44);
INSERT INTO `ajax_example` VALUES ('Frank', 45, 'm', 87);
INSERT INTO `ajax_example` VALUES ('Jill', 22, 'f', 72);
INSERT INTO `ajax_example` VALUES ('Tracy', 27, 'f', 0);
```

INSERT INTO `ajax_example` VALUES ('Julie', 35, 'f', 90);

Client Side HTML file

Now lets have our client side HTML file which is **ajax.html** and it will have following code

```
<html>
  <body>

    <script language = "javascript" type = "text/javascript">
      <!--
        //Browser Support Code
        function ajaxFunction(){
          var ajaxRequest;  // The variable that makes Ajax possible!

          try {
            // Opera 8.0+, Firefox, Safari
            ajaxRequest = new XMLHttpRequest();
          }catch (e) {
            // Internet Explorer Browsers
            try {
              ajaxRequest = new ActiveXObject("Msxml2.XMLHTTP");
            }catch (e) {
              try{
                ajaxRequest = new ActiveXObject("Microsoft.XMLHTTP");
              }catch (e){
                // Something went wrong
                alert("Your browser broke!");
                return false;
              }
            }
          }

          // Create a function that will receive data
          // sent from the server and will update
          // div section in the same page.

          ajaxRequest.onreadystatechange = function(){
            if(ajaxRequest.readyState == 4){
              var ajaxDisplay = document.getElementById('ajaxDiv');
              ajaxDisplay.innerHTML = ajaxRequest.responseText;
            }
          }

          // Now get the value from user and pass it to
          // server script.

          var age = document.getElementById('age').value;
          var wpm = document.getElementById('wpm').value;
          var sex = document.getElementById('sex').value;
          var queryString = "?age=" + age ;

          queryString +=  "&wpm=" + wpm + "&sex=" + sex;
          ajaxRequest.open("GET", "ajax-example.php" + queryString, true);
          ajaxRequest.send(null);
```

```
      }
    //-->
  </script>

  <form name = 'myForm'>
    Max Age: <input type = 'text' id = 'age' /> <br />
    Max WPM: <input type = 'text' id = 'wpm' />
    <br />

    Sex: <select id = 'sex'>
      <option value = "m">m</option>
      <option value = "f">f</option>
    </select>

    <input type = 'button' onclick = 'ajaxFunction()' value = 'Query MySQL'/>

  </form>

  <div id = 'ajaxDiv'>Your result will display here</div>
 </body>
</html>
```

NOTE − The way of passing variables in the Query is according to HTTP standard and the have formA.

```
URL?variable1=value1;&variable2=value2;
```

Now the above code will give you a screen as given below

NOTE − This is dummy screen and would not work.

Max Age:

Max WPM:

Sex: m QUREY MYSQL

Server Side PHP file

So now your client side script is ready. Now we have to write our server side script which will fetch age, wpm and sex from the database and will send it back to the client. Put the following code into "**ajax-example.php" file.**

```
<?php
   $dbhost = "localhost";
  $dbuser = "dbusername";
  $dbpass = "dbpassword";
  $dbname = "dbname";

  //Connect to MySQL Server
  mysql_connect($dbhost, $dbuser, $dbpass);

  //Select Database
  mysql_select_db($dbname) or die(mysql_error());

  // Retrieve data from Query String
  $age = $_GET['age'];
  $sex = $_GET['sex'];
```

```
$wpm = $_GET['wpm'];

// Escape User Input to help prevent SQL Injection
$age = mysql_real_escape_string($age);
$sex = mysql_real_escape_string($sex);
$wpm = mysql_real_escape_string($wpm);

//build query
$query = "SELECT * FROM ajax_example WHERE sex = '$sex'";

if(is_numeric($age))
$query .= " AND age <= $age";

if(is_numeric($wpm))
$query .= " AND wpm <= $wpm";

//Execute query
$qry_result = mysql_query($query) or die(mysql_error());

//Build Result String
$display_string = "<table>";
$display_string .= "<tr>";
$display_string .= "<th>Name</th>";
$display_string .= "<th>Age</th>";
$display_string .= "<th>Sex</th>";
$display_string .= "<th>WPM</th>";
$display_string .= "</tr>";

// Insert a new row in the table for each person returned
while($row = mysql_fetch_array($qry_result)) {
  $display_string .= "<tr>";
  $display_string .= "<td>$row[name]</td>";
  $display_string .= "<td>$row[age]</td>";
  $display_string .= "<td>$row[sex]</td>";
  $display_string .= "<td>$row[wpm]</td>";
  $display_string .= "</tr>";
}
echo "Query: " . $query . "<br />";

$display_string .= "</table>";
echo $display_string;
?>
```

Now try by entering a valid value in "Max Age" or any other box and then click Query MySQL button.

Max Age:

Max WPM:

Sex: QUERY MYSQL

.

INTRODUCTION TO XML

(Extensible markup language) the World Wide Web consortium (W3C) begins their work on XML, another meta markup language. The first XML version 1.0, published in 1998. The difference between XML and HTML is HTML is a markup language where as XML is a mark up language used to create new mark up languages. HTML limits you to use only fixed number of tags, where as XML allows to create new tags. For example, you are developing website for a college, then you have tags like <SNO>,<STUDENTNAME>,<DOB> etc. HTML,XML languages are derived from standard generalized markup language (SGML).

XML and related technologies and those are: SGML,XSL,W3C,SOAP,XSLT,DOM,SAX

SGML, standard generalized markup language is basis for all markup language. XSL,extensible stylesheet language is a combination of XML and style sheets.

XSLT,XSL transformations provides rules for transformations from one XML to another.

SOAP, simple object access protocol is communication protocol for internet to XML documents and it provides notifications for events. SAX , simple APL for XML, predefined application package interface.DOM, document object model.

Use of XML : i)simplifies the data exchange procedure ii)easy to organize the document iii)tags or document elements are reusable iv)XML provides consistency in display of information

Applications of XML

i)electronic commerce (popularly known as E-commerce) financial funds transfer

ii)multimedia messages and messaging exchange better environment for data transfer configuration of files, used in J2EE environment,

iii)The XML document is language natural. That means a java program can generate an XML document and this document can be parsed by perl.

iv)XML files are independent of an operating system.

Differences between XML and HTML

XML	HTML
User defined tags	Predefined tags
User has control on tags	As predefined, on such control
XML separates content from presentation	HTML specifies presentation
XML allows any kind of tag names like<UNAME></UNAME>	HTML defines set of legal tags
XML based on SGML	HTML based on SGML
XML allows users to create	HTML doesn't allow users to create new tags

new tags	
Self describing data can be possible	No possibility
You can generate new mark up languages using XML	No such possibility
XML is case sensitive	HTML is not case sensitive
Root element is user defined and only one root element allowed.	Root element is <HTML>

XML features:

i) XML allows the user to define his own tags and his own document structure. XML document is pure information wrapped in XML tags.

ii)XML is a text based language, plain text files can be used to share data. XML provides a software and hardware independent way of sharing data.

Simple XML document:

XML tags and attributes depend on the user. XML declaration always starts with xml key word.

<? xml version =’’1.0’’’?>

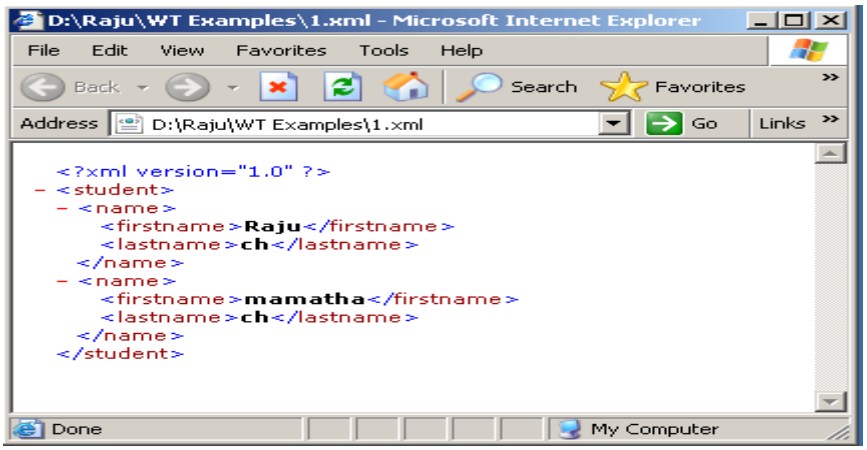
Here you can give version along with the xml key word, version is attributing and value 1.0 indicates that xml 1.0 version you are using.

First declaration tag starts with left angular bracket along with question mark(<?),ending with question mark followed by the right angular bracket(?>)second assignment is comment and then we start with actual xml program. First element<student> called root element. A root element contains other sub elements here we used <name>as sub element and its also known as container element, because it contains other sub elements<first name>,<last name>.sub elements is also called as children.

Example 1:

```
<?xml version="1.0"?>
<student>
        <name>
                <firstname>Raju</firstname>
                <lastname>ch</lastname>
        </name>
        <name>
                <firstname>mamatha</firstname>
                <lastname>ch</lastname>
        </name> </student>
```

The filename extension used for xml program is .xml the name of above program is one.xml.

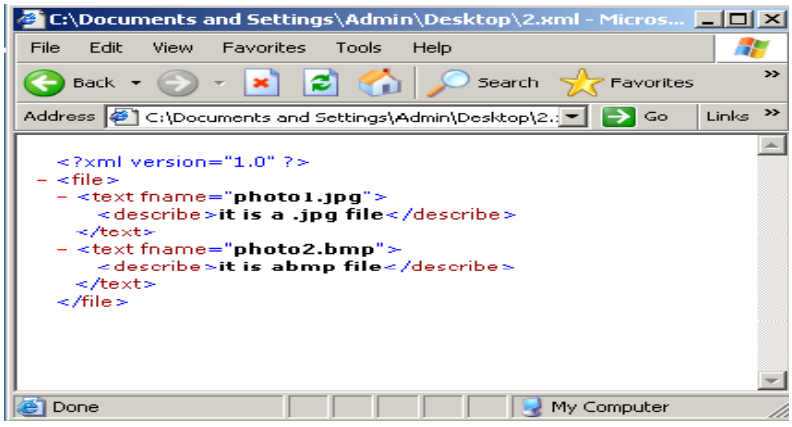


Elements and attributes: In XML the basic entity is element the elements are used for defining the tags. The elements typically consist of opening and closing tag. Mostly only one element is used to define a single tag. The syntax of writing any element for opening tag is <element name>The syntax of writing any closing element for closing tag is </element name>.An empty tag can be defined by putting a / (forward slash) before closing bracket. A space or a tab character is not allowed in the element name or in attribute name.

Namespace in XML: Sometime we need to create two different elements by the same name. The xml document allows creating different elements which are having the common name. This technique is known as name space. In some web documents it becomes necessary to have the same name for two different elements. Here different elements mean the elements which are intended for different purpose.

Example 2:

```
<?xml version="1.0"?>
<file>
<text fname="photo1.jpg">
<describe> it is a .jpg file</describe>
</text>
<text fname="photo2.bmp"> <describe>it is abmp file</describe> </text></file>
```



DOCUMENT TYPE DEFINITION (DTD): The document type definition used to define the basic building block of any xml document. Using DTD we can specify the various elements types, attributes and their relationship with in another. Basically DTD is used to specify the set of rules for structuring data in any XML file.

Various building blocks of XML are

1. Elements
2. Attribute.
3. CDATA
4. PCDATA

Elements: the basic entity is element. The elements are used for defining the tags. The elements typically consist of opening and closing tag. Mostly only one element is used to define a single tag.

Syntax: <! ELEMENT element-name (content-model)>

Ex: <!ELEMENT employee (eno,ename,salary)>

<!ELEMENT eno (#PCDATA)> // PCDATA(datatype) allows only text data

<!ELEMENT ename (#PCDATA)>

<!ELEMENT salary (#PCDATA)>

Attributes: it provides extra information about elements. These are always placed inside the starting tag of an element. Attribute always come in name/value pairs.

Ex:<img src='raj.jpg">

Element –img attribute-src value of attribute-raj.jpg.

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.

<ELEMENT student(name)*>

Here * indicates zero or more occurrences, other characters are

+ indicates 1,2,n.

?indicate 0,1

Default exactly once.

Types of DTD:

1. Internal DTD 2.External DTD

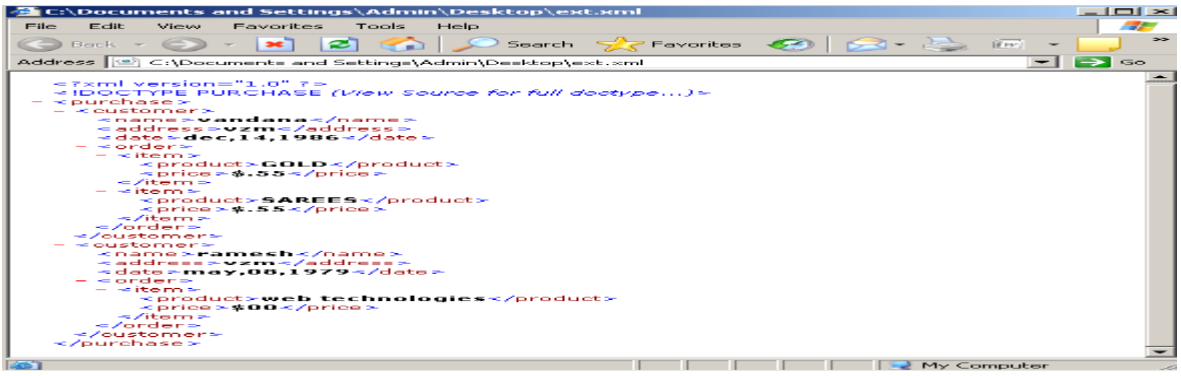
Internal DTD:

In an internal DTD we enclose the DTD in a <DOCTYPE> element, providing the name of the root element of the document in the <DOCTYPE>

EXAMPLE 3

```
<?xml version="1.0"?>
<!DOCTYPE DOCUMENT[
<!ELEMENT PURCHASE (CUSTOMER*)>
<!ELEMENT CUSTOMER (name,address,date,order)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT order (items*)>
<!ELEMENT item (product,price)>
<!ELEMENT product (#PCDATA)>
 <!ELEMENT price (#PCDATA)> ]>
<purchase>
<customer>
<name>vandana</name>
<address>vzm</address>
<date>dec,14,1986</date>
<order>
<item>
<product>GOLD</product>
<price>$.55</price>
</item>
<item>
<product>SAREES</product>
<price>$.55</price>
</item>
</order>
</customer>
<customer>
```

<name>ramesh</name>
<address>vzm</address>
<date>may,08,1979</date>
<order>
<item>
<product> web technologies</product>
<price>$00</price>
</item>
</order>
</customer>
</purchase>



External DTD:

In this type, an external DTD file is created ans its name must be specified in the corresponding XML file. XML document illustrate the use of external DTD.

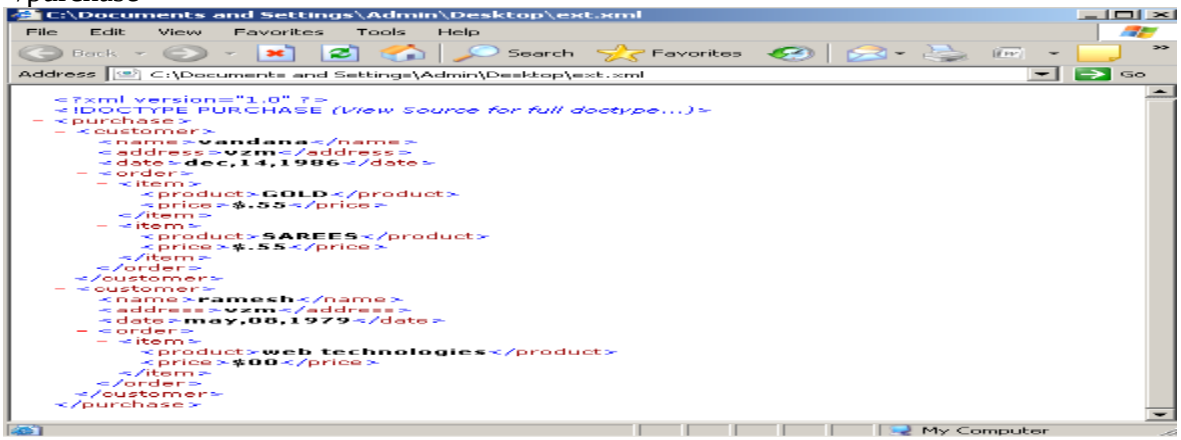
Step1: creation of DTD file[student.dtd] Open notepad type following code into it-

<!ELEMENT PURCHASE (CUSTOMER*)>
<!ELEMENT CUSTOMER (name, address, date, order)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT order (items*)>
<!ELEMENT item (product, price)>
<!ELEMENT product (#PCDATA)>
<!ELEMENT price (#PCDATA)>

Step2: creation of XML document[example3.html] Now create a XML document as follows-

<?xml version="1.0"?>
<!DOCTYPE PURCHASE SYSTEM "student.dtd">
<purchase>
<customer>
<name>vandana</name>
<address>vzm</address>
<date>dec,14,1986</date>
<order>
<item>
<product>GOLD</product>
<price>$.55</price>
</item>
<item>
<product>SAREES</product>
<price>$.55</price>
</item>
</order>

</customer>
<customer>
<name>ramesh</name>
<address>vzm</address>
<date>may,08,1979</date>
<order>
<item>
<product> web technologies</product>
<price>$00</price>
</item>
</order>
</customer>
</purchase>



Cardinality Operators:

- * --- 0 to n
- + --- 1 to n
- ? ---- 0 or 1

Example:

books.dtd

<! ELEMENT books (book)>
<! ELEMENT book((book-name | title),author+,price,publications?)>
<! ELEMENT book-name (#PCDATA)>
<! ELEMENT title (#PCDATA)>
<! ELEMENT author (#PCDATA)>
<! ELEMENT price (#PCDATA)>
<! ELEMENT publications (#PCDATA)>

Books.xml

<!DOCTYPE books SYSTEM "books.dtd">
<books>
<book>
<book-name>CP</book-name>
<author>YSK</author>
<author>Sivakoti</author>
<price>777</price>
<publications>NIT</publications>
</book>
</books>

TYPES OF ELEMENTS:

1. **Text Only Elements**

<! ELEMENT ename (#PCDATA)>
<! ELEMENT salary (#PCDATA)>

2. **Child Only Elements**

<! ELEMENT employee (ename,salary)>

3. Mixed Elements

<employee>
The employee name is <ename>YSK</ename> and employee salary is <salary>50000</salary>
</employee>

Example: **employees.dtd**

<! ELEMENT employees (employee *)>
<! ELEMENT employee (#PCDATA | ename | salary*)>
<! ELEMENT ename (#PCDATA)>
<! ELEMENT salary (#PCDATA)>

Employees.xml

<! DOCTYPE employees SYSTEM "employees.dtd">
<employees>
<employee>
The employee name is <ename>YSK</ename> and employee salary is <salary>50000</salary>
</employee>
</employees>

4. EMPTY elements

Syntax: <! ELEMENT employee EMPTY>

EMPTY.xml

<!DOCTYPE employees [
<! ELEMENT employees (employee*)>
<! ELEMENT employee EMPTY>
<employees>
<employee></employee>Or<employee/>
</employees>

5.ANY ELEMENTS

Anyexmaple.xml

<!DOCTYPE employees [
< ! ELEMENT employees (employee+)>
< ! ELEMENT employee ANY>
< ! ELEMENT empname (#PCDATA)>
< ! ELEMENT empsalary (#PCDATA)>
]>
<employees>
<employee>
The empname is <empname>YSK</empname>
</employee>
<employee>This employee information is </employee>
<employee></employee>
<employee>The employee salary is
<empsalary>35000</empsalary>
</employee>
</employees>

Types of Attributes:

Attribute is a name value pair. Attributes are used to provide some extra information about of an element. Attribute is associated with opening tag of an element.

Attribute values must be enclosed with either single quotes or double quotes

Syntax: <! ATTLIST element-name attrName attrDatatype attrSpecifier other-information>

Example:

Employ.dtd

<! ELEMENT employees (employee+)>
<! ELEMENT employee EMPTY>
<! ATTLIST employee empno CDATA #REQUIRED>
<! ATTLIST employee name CDATA #REQUIRED>

<! ATTLIST employee salary CDATA #REQUIRED>

Employ.xml

<employees>
<employee empno='101' name='siva' salary='28000'/>
<employee empno='102' name='koti' salary='32000'/>
</employees>

Attribute Specifiers: Used to specify an attribute is a Mandatory attribute, Optional attribute,fixed attribute or default attribute

i) #REQUIRED ----- mandatory attribute
ii) #IMPLIED ----- optional attribute
iii) #FIXED ---- fixed attribute
iv) Default ---- default attribute (not specify any type)

Example:

students.dtd

<! ELEMENT students (student+)>
<! ELEMENT student EMPTY>
<! ATTLIST student std-id CDATA #REQUIRED>
<! ATTLIST student std-name CDATA #IMPLIED>
<! ATTLIST student course CDATA "WT">
<! ATTLIST student std-fee CDATA #FIXED "1000">

students.xml

<! DOCTYPE students SYSTEM "students.dtd">
<students>
<student std-id="101" course="c" std-fee="1000"/>
<student std-name="raja"/>
<student std-id="102" course="java" std-fee="1000"/>
</students>

Attribute Datatypes:

i) CDATA ii) Enumerated Data type iii) ID iv) IDREF v) IDREFs vi) ENTITY vii) ENTITIES viii) NOTATAION ix) NMTOKEN x) NMTOKENS

XML SCHEMAS:

Xml schema is an XML alternative to DTD. The XML schemas are used to represent the structure of XML document. XML schema language is also referred to as XML schema definition. The XML schema language is called as XML schema definition language (XSD).XML schema are written in XML . XML schema supports name spaces.

XML schema definition:

The <schema >element is the root element of every XML schema.<schema>element may contains some attributes.

<?xml version="1.0"?>
<xs:schema>
<xs:schema>

XSD elements are two types

1. Simple element. 2.complex element.

simple element is an XML element that can contain only text. It can't contain any other elements or attributes. Syntax: <xs:element name:"aaa" datatype="int"/>

2.Complex Element :If any element contains child elements or attributes or both is called as complex element. The following four ways to create a complex elements.

i)Elements with text data and attributes ii)Elements with empty content and attributes
iii)Elements with child elements and/or attributes iv)Elements with mixed content

Syntax

A complex element with child element

<element name ="element name" >
<complexType>
<sequence>

<element name=”child1” type=”dt”/>
</sequence>
</complexType>
</element>
--Example—emp.xsd
<schema>
<element name=”employee”>
<complexType>
<sequence>
<element name=”eno” type=”int”/>
<element name=”ename” type=”stirng”/>
</sequence>
</complexType>
</element>
</schema>
Emp.xml
<employee>
<eno>101</eno>
<ename>WT</ename>
</employee>

EXAMPLE :

XML schema[student schema.xsd]
<?xml version=”1.0”?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XML schema>
<xs:element name=”student”>
<xs:complex type>
<xs:sequence>
<xs:element name=”name” type=”xs:string”/>
<xs:element name=”address” type=”xs:string”/>
<xs:element name=”std” type=”xs:string”/>
<xs:element name=”marks” type=”xs:string”/>
</xs:sequence>
</xs:complex type>
</xs:element>
</xs:schema>

The xs is qualifier used to identify the schema elements and types, the document element of schema is xs:schema. The xs:schema is root element, it takes the attribute xmlns:xs which has the value http://www.w3.org/2001/XML schema, this declaration indicates that document should follow the rules of XML schema.

Then comes xs:element which is used to define the xml element. In above case the element student is of complex type who have four childs elements:name, address, std and marks. All these elements are of simple type string.

XML document [examp.xml]
<?xml version =”1.0”?>
<student xmlns:xsi=http://www.w3.org/2001/XML schema-instance
xsi:no namespace schema location=”studentschema.xsd”>
<name>vandana</name>
<address>Vzm</address>
<std>first</std>
<marks>80 percent</marks>
</student>

XML datatypes:

i)Simple Types ii)Complex Types

Simple Types → i) user derived types ii) Built-in types(44)

User Derived types → i) atomic ii) non atomic

Built-in types→ i)primitive (19) ::

string,boolean,decimal,float,double,duration,datetime,time,date,gYearMonth,gYear,gMonthday,gday, gmonth,hexbinary,base64Binary,anyURI,Qname,NOTATION

ii)derived (25) ::

normalizedstring, token, language, NMTOKEN, NMTOKENS,Name,NCName,ID,IDREF,IDREFS,ENTITY,ENTITIES,integer,nonpositiveInteger, negativeInteger,long,int,short,byte,nonNegativeInteger,unsignedLong,unsignedInt,unsignedShort, unsignedByte,positiveInteger.

Complex Types→ i) Empty ii)Simple content iii)Complex content

Complex Content → i)Sequence ii)Choice iii)All

Various data types are String, date, numeric, Boolean

Xs:string it contains group of characters, lines, tabs or white spaces.

Xs:date used to represent date the format of this date is YYYY-mm-dd xs:time use to represent time. The format of this time is hh:mm:ss

Xs:decimal used to represent float values.

Xs:integer used to represent integer values.

Xs:Boolean used to resent Boolean values either true or false.

PRESENTING XML:

XSL: XSL is a language capable of transforming as well as formatting given XML document.XSL means extensible style sheet language.

Working of XSL:

1. Start with a raw XML document (create a xml file)

2. Create an XSL style sheet (create a XSL file)

3. Link the XSL style sheet to the XML document.

4. Use XSLT complaint browser to transform your XML into XHTML.

EXAMPLE :

First create .xml file[demo.xml]

```
<?xml-stylesheet type="text/xsl" href="demo1.xsl"?>
<menu>
<dvd>
<name>Raju</name>
<actor>chiru</actor>
<place>London</place>
<amount>$00</amount>
</dvd>
<dvd>
<name>Ravi</name>
<actor>pavan</actor>
<place>America</place>
<amount>$00</amount>
</dvd>
</menu>
```

Demo1.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<table border="1">
<tr>
<th>name</th>
<th>actor</th>
```

```
</tr>
<xsl:for-each select="menu/dvd">
<tr>
<td><xsl:value-of select="name"/>
</td>
<td><xsl:value-of select="actor"/>
</td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

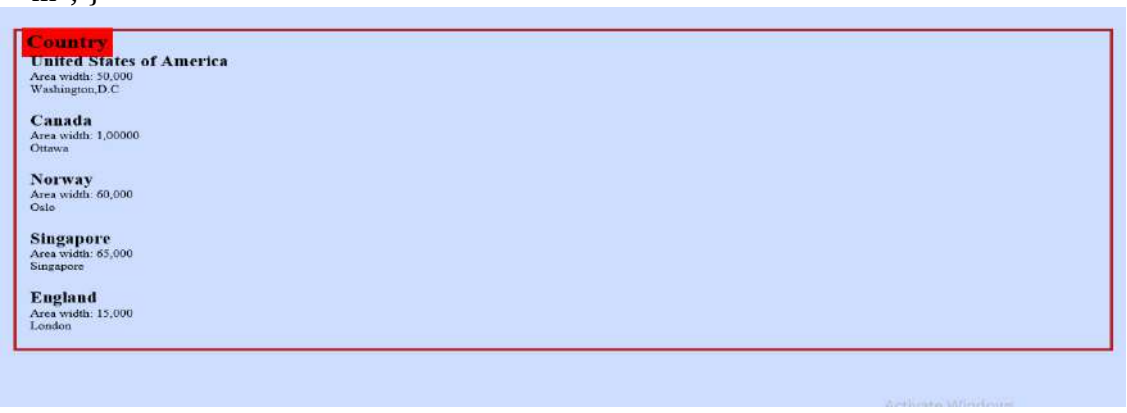
XML WITH CSS

EXAMPLE: [samplecss.xml]

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="sampleemp.css"?>
<!DOCTYPE globe>
<globe>
<Country>
<name>United States of America</name>
<areawidth>50,000</areawidth>
<capital>Washington,D.C</capital>
</Country>
<Country>
<name>Canada</name>
<areawidth>1,00000</areawidth>
<capital>Ottawa</capital>
</Country>
<Country>
<name>Norway</name>
<areawidth>60,000</areawidth>
<capital>Oslo</capital>
</Country>
<Country>
<name>Singapore</name>
<areawidth>65,000</areawidth>
<capital>Singapore</capital>
</Country>
<Country>
<name>England</name>
<areawidth>15,000</areawidth>
<capital>London</capital>
</Country>
</globe>
```

[Sampleemp.css]

```
globe:before {
display: semi-block;
width: 9em;
font-weight: bold;
font-size: 170%;
content: "Country";
margin: -.85em 0px .35em -.35em;
padding: .2em .20em;
background-color: #FF0000;
}
globe {
display: block;
margin: 2em 1em;
border: 4px solid #B22222;
padding: 0px 1em;
background-color:  #cdf;
}
Country {
display: block;
margin-bottom: 1.5em;
}
name {
display: block;
font-weight: bold;
font-size: 140%;
}
areawidth {
display: block;
}
areawidth:before {
content: "Area width: ";
}
area:after {
content: " million km\B2";
}
depth {
display: block;
}
depth:before {
content: "Mean depth: ";
}
depth:after {
content: " m"; }
```



XML Parsers

An XML parser is a software library or package that provides interfaces for client applications to work with an XML document. The XML Parser is designed to read the XML and create a way for programs to use XML.

XML parser validates the document and check that the document is well formatted.

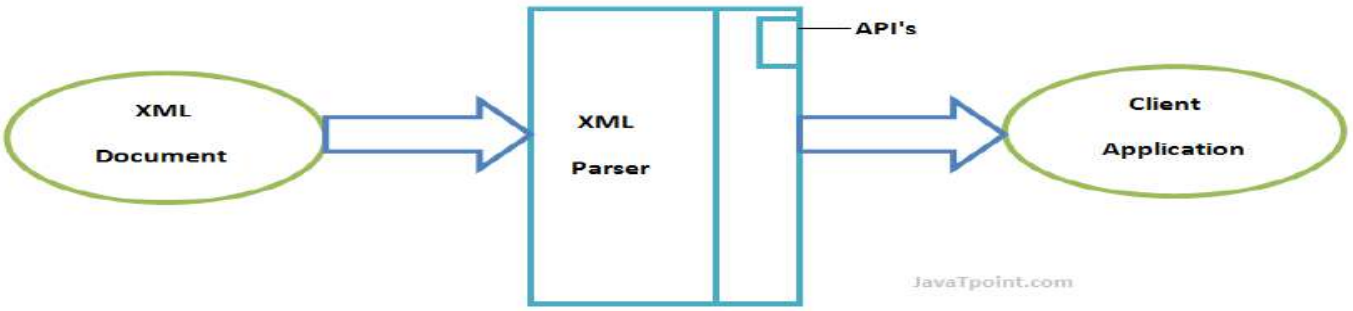
All modern browsers have a build –in XML parser that can be used to read and manipulate XML.

The parser reads XML into memory and converts it into an XML DOM object that can be access with java script.

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, all parsers contain functions to traverse XML trees, access, insert, and delete nodes.

Let's understand the working of XML parser by the figure given below:



Types of XML Parsers

These are the two main types of XML Parsers:

1. DOM
2. SAX

DOCUMENT OBJECT MODEL (DOM):

- DOM is a set of platform independent and language neutral application programming interface (API) which describes how to access and manipulate the information stored in XML or in HTML documents.
- XML DOM for loading the XML document accessing the elements of XML document. Deleting the elements of XML documents changing the elements of XML document.
- The document object model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accesses and manipulated. In the DOM specification, the term "document" is used in the broad sense-increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.
- With the document object model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the document object model, with a few exceptions-in particular; the DOM interfaces for the XML internal and externals subsets have not yet been specified.

The DOM is separated into 3 different parts/levels:

1. Core DOM- standard model for any structured document.

2. XML DOM- standard model for XML documents.

3. HTML DOM-standard model for HTML documents.

What is the XML DOM?

The XML DOM is:

- A standard object model for XML
- A standard programming interface for XML.
- Platform-and language –independent.

- A W3C standard

THE XML DOM defines the objects and properties of all XML elements, and the methods to access them.

The DOM is a programming API for documents. It closely resembles the structure of the documents it models. For instance, consider this table, taken from an HTML document:

```
<TABLE>
<TBODY>
<TR>
<TD> shady grove</TD>
<TD>Aeolian</TD>
</TR>
<TR>
<TD> over the river, Charlie</TD>
<TD> dorian</TD>
</TR>
</TBODY>
</TABLE>
```

Example:[overseas.html]

```
<html>
<body>
<script type =”text/javascript”>
 Try
{
Xmldocument=new
activeXobject(“Microsoft.XMLDOM”);
}
Catch(e)
{
Try
{
Xmldocument=document.implementation.createdocument(“,”null);
}
Catch(e)
{
Alert(e.message)
}
}
Try
{
Xmldocument.async=false;
Xmldocument.load(“example3.xml);
Document.write(“xml document example3.xml is loaded”);
}
Catch(e)
{
Alert(e.message);
}
</script>
</body>
</html>
```

SAX (Simple API for XML)

A SAX Parser implements SAX API. This API is an event based API and less intuitive.

Features of SAX Parser

It does not create any internal structure.

Clients does not know what methods to call, they just overrides the methods of the API and place his own code inside method.
It is an event based parser, it works like an event handler in Java.

Advantages

1) It is simple and memory efficient.
2) It is very fast and works for huge documents.

Disadvantages

1) It is event-based so its API is less intuitive.
2) Clients never know the full information because the data is broken into pieces.

Difference between DOM and SAX:

S.NO	DOM	SAX
1	DOM is a tree based parsing method	SAX is an event based parsing method
2	We can insert or delete a node	We can insert or delete a node
3	Traverse in any direction	Top to bottom traversing
4	Stores the entire XML document in to memory before processing.	Parses node by node
5	Occupies more memory	Doesn't store the XML in memory
6	DOM preserves comments	SAX doesn't preserve comments.
7	Import javax.xml.parsers.*; Import org.w3c.dom.*;	Import javax.xml.sax.*; Import org.xml.sax.helpers.*;

XML and HTML

We can combine XML and HTML into single document. This is possible with XML tag. The following program explains how data of XML is embedded into HTML table form.
Write a program to combine XML and HTML

Reg.xml

```
<?xml version="1.0"?>
<members>
        <member>
                  <sno>01</sno>
                 <sname>Y. Ramesh kumar</sname>
                  <branch>H O D</branch>
         </member>
         <member>
           <sno>02</sno>
           <sname>N.suresh</sname>
           <branch>MCA</branch>
</member>
<member>
          <sno>03</sno>
          <sname>ch.santhosh kumar</sname>
         <branch>MCA</branch>
</member>
<member>
          <sno>04</sno>
           <sname>k.ramesh babu</sname>
            <branch>MCA</branch>
</member>
<member>
           <sno>05</sno>
            <sname>srinu</sname>
```

```
            <branch>IT</branch>
</member>
<member>
            <sno>06</sno>
            <sname>guru charan</sname>
             <branch>IT</branch>
</member>
<member>
            <sno>07</sno>
             <sname>K SV K srikanth</sname>
            <branch>IT</branch>
</member>
<member>
           <sno>08</sno>
          <sname>Praveen kumar</sname>
          <branch>IT</branch>
</member>
<member>
    <sno>09</sno>
      <sname> santhosh</sname>
       <branch> IT</branch>
</member>
<member>
       <sno>10</sno>
     <sname> murthy</sname>
    <branch> IT</branch>
</member>
<member>
<sno> 11</sno>
<sname>suresh </sname>
<branch>IT</branch>
</member>
<member>
<sno>12</sno>
<sname>raju</sname>
<branch>IT</branch>
</member>
<member>
<sno>13</sno>
<sname>praveen</sname>
<branch>IT</branch>
</member>
<member>
<sno>14</sno>
<sname>praneeth</sname>
<branch>IT</branch>
</member>
<member>
<sno>15</sno>
<sname>praveen</sname>
<branch>IT</branch>
</member>
</members>
```

Regmembers.html

```
<html>
```

```
<head><title>registered members:portal team</title></head>
<body bgcolor="wheat">
<center><font size=5 face="bookman old style" color="blue"> portal team details</font></center>
<br><br>
<xml id="rRecords" src="D:\Raju\WT Examples\reg.xml"></xml>
<table id ="emptable" width="50%" align="center" border="2" bordercolor="blue" cellspacing="0"
datasrc="#rRecords" dataPageSize="5">
<thead>
<tr style="background-color:cyan ;color:blue;">
<th>S.No.</th>
<th>Name</th>
<th>branch</th>
</tr>
</thead>
<tbody>
<tr>
<td align ="center"><span datafld="sno"></span></td>
<td><span datafld="sname"></span></td>
<td><span datafld="branch"></span></td>
</tr>
</tbody>
</table>
<br><br>
<center>
<input type="button" value="<<first" onClick="emptable.firstPage()">
<input type="button" value="<previous" onClick="emptable.previousPage()">
<input type="button" value="next>" onClick="emptable.nextPage()">
<input type="button" value="last>>" onClick="emptable.lastPage()">
</center></body></html>
```

Difference between HTML and XHTML

S.NO	HTML	XHTML
1	Not case sensitive	Case sensitive
2	Need not be paired tag	Need to be paired tag
3	No short cut tags.	Shortcut tags allowed as ending tags
4	All attributes may or may not be in quotations	All attributes must be in double quotations

UNIT-IV
SYLLABUS

- **PHP Programming**
- Introducing PHP
- Creating PHP script
- Running PHP script.
- Working with variables and constants
- Using variables
- Using constants
- Data types
- Operators
- Controlling program flow
- Conditional statements
- Control statements
- Arrays
- Functions
- Working with forms and Databases such as MySQL.

What You Should Already Know
Before you continue you should have a basic understanding of the following:

- HTML
- JavaScript

What is PHP?
- PHP stands for **P**HP: **H**ypertext **P**reprocessor
- PHP is a widely-used, open source scripting language
- PHP scripts are executed on the server
- PHP is free to download and use

What is a PHP File?
- PHP files can contain text, HTML, JavaScript code, and PHP code
- PHP code are executed on the server, and the result is returned to the browser as plain HTML
- PHP files have a default file extension of ".php"

What Can PHP Do?
- PHP can generate dynamic page content
- PHP can create, open, read, write, and close files on the server
- PHP can collect form data
- PHP can send and receive cookies
- PHP can add, delete, modify data in your database
- PHP can restrict users to access some pages on your website
- PHP can encrypt data

With PHP you are not limited to output HTML. You can output images, PDF files, and even Flash movies. You can also output any text, such as XHTML and XML.

Why PHP?
- PHP runs on different platforms (Windows, Linux, Unix, Mac OS X, etc.)
- PHP is compatible with almost all servers used today (Apache, IIS, etc.)
- PHP has support for a wide range of databases
- PHP is free. Download it from the official PHP resource
-

What Do I Need?
To start using PHP, you can:

- Find a web host with PHP and MySQL support
- Install a web server on your own PC, and then install PHP and MySQL

Use a Web Host With PHP Support
If your server has activated support for PHP you do not need to do anything.

Just create some .php files, place them in your web directory, and the server will automatically parse them for you.

You do not need to compile anything or install any extra tools. Because PHP is free, most web hosts offer PHP support.

Set Up PHP on Your Own PC

However, if your server does not support PHP, you must:

- install a web server
- install PHP
- install a database, such as MySQL

The official PHP website (PHP.net) has installation instructions for PHP: http://php.net/manual/en/install.php

The PHP script is executed on the server, and the plain HTML result is sent back to the browser.

Basic PHP Syntax

A PHP script always starts with **<?php** and ends with **?>**. A PHP script can be placed anywhere in the document.

On servers with shorthand-support, you can start a PHP script with <? and end with ?>.

For maximum compatibility, we recommend that you use the standard form (<?php) rather than the shorthand form.

```
<?php
// PHP code goes here
?>
```

The default file extension for PHP files is ".php".

A PHP file normally contains HTML tags, and some PHP scripting code.

Below, we have an example of a simple PHP script that sends the text "Hello World!" back to the browser:

Example

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

<?php
echo "Hello World!";
?>
</body>
</html>
```

Each code line in PHP must end with a semicolon. The semicolon is a separator and is used to distinguish one set of instructions from another.

There are two basic statements to output text with PHP: **echo** and **print**.

In the example above we have used the echo statement to output the text "Hello World".

Comments in PHP

In PHP, we use **//** to make a one-line comment, or **/*** and ***/** to make a comment block:

Example

```
<!DOCTYPE html>
<html>
<body>
<?php
//This is a comment
/* This
is
a comment block
*/
```

```
?>
</body>
</html>
```

PHP Variables
As with algebra, PHP variables are used to hold values or expressions.

A variable can have a short name, like x, or a more descriptive name, like carName. Rules for PHP variable names:

- Variables in PHP starts with a $ sign, followed by the name of the variable
- The variable name must begin with a letter or the underscore character
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0- 9, and _)
- A variable name should not contain spaces
- Variable names are case sensitive (y and Y are two different variables)

Creating (Declaring) PHP Variables
PHP has no command for declaring a variable.
A variable is created the moment you first assign a value to it:
$myCar="Volvo";
After the execution of the statement above, the variable **myCar** will hold the value **Volvo**.
Tip: If you want to create a variable without assigning it a value, then you assign it the value of *null*.
Let's create a variable containing a string, and a variable containing a number:

```
<?php
$txt="Hello World!";
$x=16;
?>
```

Note: When you assign a text value to a variable, put quotes around the value.

PHP is a Loosely Typed Language
In PHP, a variable does not need to be declared before adding a value to it.

In the example above, notice that we did not have to tell PHP which data type the variable is. PHP automatically converts the variable to the correct data type, depending on its value.
In a strongly typed programming language, you have to declare (define) the type and name of the variable before using it.

PHP Variable Scope
The scope of a variable is the portion of the script in which the variable can be referenced. PHP has four different variable scopes:

- local
- global
- static
- parameter

Local Scope
A variable declared **within** a PHP function is local and can only be accessed within that function. (the variable has local scope):

```
<?php
$a = 5; // global scope

function myTest()
{
echo $a; // local scope
}

myTest();
?>
```

The script above will not produce any output because the echo statement refers to the local scope variable $a, which has not been assigned a value within this scope.

You can have local variables with the same name in different functions, because local variables are only recognized by the function in which they are declared.

Local variables are deleted as soon as the function is completed.

Global Scope
Global scope refers to any variable that is defined outside of any function.
Global variables can be accessed from any part of the script that is not inside a function. To access a global variable from within a function, use the **global** keyword:

```
<?php
$a = 5;
$b = 10;

function myTest()
{
global $a, $b;
$b = $a + $b;
}

myTest();
echo $b;
?>
```

The script above will output 15.

PHP also stores all global variables in an array called $GLOBALS[*index*]. Its index is the name of the variable. This array is also accessible from within functions and can be used to update global variables directly.

The example above can be rewritten as this:

```
<?php
$a = 5;
$b = 10;

function myTest()
{
$GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
}

myTest();
echo $b;
?>
```

Static Scope
When a function is completed, all of its variables are normally deleted. However, sometimes you want a local variable to not be deleted.

To do this, use the **static** keyword when you first declare the variable: static
$rememberMe;
Then, each time the function is called, that variable will still have the information it contained from the last time the function was called.

Note: The variable is still local to the function.

Parameters
A parameter is a local variable whose value is passed to the function by the calling code. Parameters are declared in a parameter list as part of the function declaration:

```
function myTest($para1,$para2,...)
{
// function code
}
```

Parameters are also called arguments. We will discuss them in more detail when we talk about functions.
A string variable is used to store and manipulate text.
String Variables in PHP
String variables are used for values that contain characters.

In this chapter we are going to look at the most common functions and operators used to manipulate strings in PHP.

After we create a string we can manipulate it. A string can be used directly in a function or it can be stored in a variable.

Below, the PHP script assigns the text "Hello World" to a string variable called $txt:

```
<?php
$txt="Hello World";
echo $txt;
?>
```

The output of the code above will be:
Hello World

Now, lets try to use some different functions and operators to manipulate the string.

The Concatenation Operator
There is only one string operator in PHP.

The concatenation operator (.) is used to put two string values together.

To concatenate two string variables together, use the concatenation operator:

```
<?php
$txt1="Hello World!";
$txt2="What a nice day!"; echo
$txt1 . " " . $txt2;
?>
```

The output of the code above will be:

Hello World! What a nice day!

If we look at the code above you see that we used the concatenation operator two times. This is because we had to insert a third string (a space character), to separate the two strings.

The strlen() function
The strlen() function is used to return the length of a string. Let's find the length of a string:

```
<?php
echo strlen("Hello world!");
?>
```

The output of the code above will be:

12

The length of a string is often used in loops or other functions, when it is important to know when the string ends. (i.e. in a loop, we would want to stop the loop after the last character in the string).

The strpos() function
The strpos() function is used to search for a character/text within a string.

If a match is found, this function will return the character position of the first match. If no match is found, it will return FALSE.

Let's see if we can find the string "world" in our string:

```
<?php
echo strpos("Hello world!","world");
?>
```

The output of the code above will be: 6

The position of the string "world" in the example above is 6. The reason that it is 6 (and not 7), is that the first character position in the string is 0, and not 1.

The assignment operator = is used to assign values to variables in PHP. The arithmetic operator + is used to add values together.

Arithmetic Operators
The table below lists the arithmetic operators in PHP:

Operator	Name	Description	Example	Result
x + y	Addition	Sum of x and y	2 + 2	4
x - y	Subtraction	Difference of x and y	5 - 2	3
x * y	Multiplication	Product of x and y	5 * 2	10
x / y	Division	Quotient of x and y	15 / 5	3
x % y	Modulus	Remainder of x divided by y	5 % 2	1
			10 % 8	2
			10 % 2	0
- x	Negation	Opposite of x	- 2	
a . b	Concatenation	Concatenate two strings	"Hi" . "Ha"	HiHa

Assignment Operators
The basic assignment operator in PHP is "=". It means that the left operand gets set to the value of the expression on the right. That is, the value of "$x = 5" is 5.

Assignment	Same as...	Description
x = y	x = y	The left operand gets set to the value of the expression on the right
x += y	x = x + y	Addition
x -= y	x = x - y	Subtraction
x *= y	x = x * y	Multiplication
x /= y	x = x / y	Division
x %= y	x = x % y	Modulus
a .= b	a = a . b	Concatenate two strings

Incrementing/Decrementing Operators

Operator	Name	Description
++ x	Pre-increment	Increments x by one, then returns x
x ++	Post-increment	Returns x, then increments x by one
-- x	Pre-decrement	Decrements x by one, then returns x
x --	Post-decrement	Returns x, then decrements x by one

Comparison Operators
Comparison operators allows you to compare two values:

Operator	Name	Description	Example
x == y	Equal	True if x is equal to y	5==8 returns false
x === y	Identical	True if x is equal to y, and they are of same type	5==="5" returns false
x != y	Not equal	True if x is not equal to y	5!=8 returns true
x <> y	Not equal	True if x is not equal to y	5<>8 returns true
x !== y	Not identical	True if x is not equal to y, or they are not of same type	5!=="5" returns true
x > y	Greater than	True if x is greater than y	5>8 returns false
x < y	Less than	True if x is less than y	5<8 returns true

x >= y	Greater than or equal to	True if x is greater than or equal to y	5>=8 returns false
x <= y	Less than or equal	True if x is less than or equal to y	5<=8 returns true to the same.

Logical Operators

Operator	Name	Description	Example
x and y	And	True if both x and y are true	x=6 y=3 (x < 10 and y > 1) returns true
x or y	Or	True if either or both x and y are true	x=6 y=3 (x==6 or y==5) returns true
x xor y	Xor	True if either x or y is true, but not both	x=6 y=3 (x==6 xor y==3) returns false
x && y	And	True if both x and y are true	x=6 y=3 (x < 10 && y > 1) returns true
x || y	Or	True if either or both x and y are true	x=6 y=3 (x==5 || y==5) returns false
! x	Not	True if x is not true	x=6 y=3 !(x==y) returns true

Array Operators

Operator	Name	Description
x + y	Union	Union of x and y
x == y	Equality	True if x and y have the same key/value pairs
x === y	Identity	True if x and y have the same key/value pairs in the same order and of the same types
x != y	Inequality	True if x is not equal to y
x <> y	Inequality	True if x is not equal to y
x !== y	Non-identity	True if x is not identical to y

Conditional statements are used to perform different actions based on different conditions.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this. In PHP we have the following conditional statements:

- **if statement** - use this statement to execute some code only if a specified condition is true
- **if...else statement** - use this statement to execute some code if a condition is true and another code if the condition is false
- **if...elseif .....else statement** - use this statement to select one of several blocks of code to be executed
- **switch statement** - use this statement to select one of many blocks of code to be executed

The if Statement
Use the if statement to execute some code only if a specified condition is true.

Syntax
if (*condition*) *code to be executed if condition is true;*

The following example will output "Have a nice weekend!" if the current day is Friday:

```
<html>
<body>

<?php
$d=date("D");
if ($d=="Fri") echo "Have a nice weekend!";
?>

</body>
</html>
```

Notice that there is no ..else.. in this syntax. The code is executed **only if the specified condition is true**.

The if...else Statement
Use the if.....else statement to execute some code if a condition is true and another code if a condition is false.

Syntax
```
if (condition)
  {
  code to be executed if condition is true;
  }
else
  {
  code to be executed if condition is false; }
```
Example

The following example will output "Have a nice weekend!" if the current day is Friday, otherwise it will output "Have a nice day!":

```
<html>
<body>

<?php
$d=date("D");
if ($d=="Fri")
  {
  echo "Have a nice weekend!";
  }
else
  {
  echo "Have a nice day!";
  }
?>

</body>
</html>
```

The if...elseif....else Statement
Use the if....elseif...else statement to select one of several blocks of code to be executed.

Syntax
```
if (condition)
  {
  code to be executed if condition is true;
  }
```

```
elseif (condition)
  {
  code to be executed if condition is true;
  }
else
  {
  code to be executed if condition is false;
  }
```

Example

The following example will output "Have a nice weekend!" if the current day is Friday, and "Have a nice Sunday!" if the current day is Sunday. Otherwise it will output "Have a nice day!":

```
<html>
<body>

<?php
$d=date("D");
if ($d=="Fri")
  {
  echo "Have a nice weekend!";
  }
elseif ($d=="Sun")
  {
  echo "Have a nice Sunday!";
  }
else
  {
  echo "Have a nice day!";
  }
?>

</body>
</html>
```

The PHP Switch Statement

Use the switch statement to select one of many blocks of code to be executed.

Syntax

```
switch (n)
{
case label1:
  code to be executed if n=label1;
  break; case
label2:
  code to be executed if n=label2;
  break;
default:
  code to be executed if n is different from both label1 and label2;
}
```

This is how it works: First we have a single expression *n* (most often a variable), that is evaluated once. The value of the expression is then compared with the values for each case in the structure. If there is a match, the block of code associated with that case is executed. Use **break** to prevent the code from running into the next case automatically. The default statement is used if no match is found.

Example

```
<html>
<body>

<?php
$x=1;
switch ($x)
{
```

```
case 1:
  echo "Number 1";
  break;
case 2:
  echo "Number 2";
  break;
case 3:
  echo "Number 3";
  break;
default:
  echo "No number between 1 and 3";
}
?>
</body>
</html>
```

An array stores multiple values in one single variable.

What is an Array?

A variable is a storage area holding a number or text. The problem is, a variable will hold only one value.

An array is a special variable, which can store multiple values in one single variable.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
$cars1="Saab";
$cars2="Volvo";
$cars3="BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The best solution here is to use an array!

An array can hold all your variable values under a single name. And you can access the values by referring to the array name.

Each element in the array has its own index so that it can be easily accessed. In PHP, there are three kind of arrays:

- **Numeric array** - An array with a numeric index
- **Associative array** - An array where each ID key is associated with a value
- **Multidimensional array** - An array containing one or more arrays

Numeric Arrays

A numeric array stores each array element with a numeric index. There are two methods to create a numeric array.

1. In the following example the index are automatically assigned (the index starts at 0):

```
$cars=array("Saab","Volvo","BMW","Toyota");
```

2. In the following example we assign the index manually:

```
$cars[0]="Saab";
$cars[1]="Volvo";
$cars[2]="BMW";
$cars[3]="Toyota";
```

Example

In the following example you access the variable values by referring to the array name and index:

```
<?php
$cars[0]="Saab";
$cars[1]="Volvo";
$cars[2]="BMW";
$cars[3]="Toyota";
```

```
echo $cars[0] . " and " . $cars[1] . " are Swedish cars.";
?>
```

The code above will output:
Saab and Volvo are Swedish cars.

Associative Arrays

An associative array, each ID key is associated with a value.
When storing data about specific named values, a numerical array is not always the best way to do it.
With associative arrays we can use the values as keys and assign values to them.

Example 1

In this example we use an array to assign ages to the different persons:

```
$ages = array("Peter"=>32, "Quagmire"=>30, "Joe"=>34);
```

Example 2

This example is the same as example 1, but shows a different way of creating the array:

```
$ages['Peter'] = "32";
$ages['Quagmire'] = "30";
$ages['Joe'] = "34";
```

The ID keys can be used in a script:

```
<?php
$ages['Peter'] = "32";
$ages['Quagmire'] = "30";
$ages['Joe'] = "34";

echo "Peter is " . $ages['Peter'] . " years old.";
?>
```

The code above will output:
Peter is 32 years old.

Multidimensional Arrays

In a multidimensional array, each element in the main array can also be an array. And each element in the sub-array can be an array, and so on.

Example

In this example we create a multidimensional array, with automatically assigned ID keys:

```
$families = array (
 "Griffin"=>array
 (
 "Peter",
 "Lois",
 "Megan"
 ),
 "Quagmire"=>array (
 "Glenn"
 ),
 "Brown"=>array (
 "Cleveland",
 "Loretta",
 "Junior"
 )
 );
```

The array above would look like this if written to the output:

```
Array
(
[Griffin] => Array
  (
  [0] => Peter
  [1] => Lois
  [2] => Megan
  )
[Quagmire] => Array (
```

```
  [0] => Glenn
  )
[Brown] => Array
  (
  [0] => Cleveland
  [1] => Loretta
  [2] => Junior
  )
)
```

Example 2

Lets try displaying a single value from the array above:

```
echo "Is " . $families['Griffin'][2] . "
a part of the Griffin family?";
```

The code above will output:
Is Megan a part of the Griffin family?

PHP Loops
Often when you write code, you want the same block of code to run over and over again in a row. Instead of adding several almost equal lines in a script we can use loops to perform a task like this.

In PHP, we have the following looping statements:

- **while** - loops through a block of code while a specified condition is true
- **do...while** - loops through a block of code once, and then repeats the loop as long as a specified condition is true
- **for** - loops through a block of code a specified number of times
- **foreach** - loops through a block of code for each element in an array

The while Loop
The while loop executes a block of code while a condition is true.

Syntax

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

Example

The example below first sets a variable *i* to 1 ($i=1;).

Then, the while loop will continue to run as long as *i* is less than, or equal to 5. *i* will increase by 1 each time the loop runs:

```
<html>
<body>

<?php
$i=1;
while($i<=5)
  {
  echo "The number is " . $i . "<br>";
  $i++;
  }
?>

</body>
</html>
```

Output:

```
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

The do...while Statement
The do...while statement will always execute the block of code once, it will then check the condition, and repeat the loop while the condition is true.

Syntax

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

Example

The example below first sets a variable *i* to 1 ($i=1;).

Then, it starts the do...while loop. The loop will increment the variable *i* with 1, and then write some output. Then the condition is checked (is *i* less than, or equal to 5), and the loop will continue to run as long as *i* is less than, or equal to 5:

```
<html>
<body>

<?php
$i=1;
do
  {
  $i++;
  echo "The number is " . $i . "<br>";
  }

while ($i<=5);
?>

</body>
</html>
```

Output:

```
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
```

Loops execute a block of code a specified number of times, or while a specified condition is true.
The for Loop
The for loop is used when you know in advance how many times the script should run.

Syntax

```
for (init; condition; increment)
  {
  code to be executed;
  }
```

Parameters:

- *init*: Mostly used to set a counter (but can be any code to be executed once at the beginning of the loop)
- *condition*: Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it

evaluates to FALSE, the loop ends.
- *increment*: Mostly used to increment a counter (but can be any code to be executed at the end of the iteration)

Note: The *init* and *increment* parameters above can be empty or have multiple expressions (separated by commas).

Example

The example below defines a loop that starts with i=1. The loop will continue to run as long as the variable *i* is less than, or equal to 5. The variable *i* will increase by 1 each time the loop runs:

```
<html>
<body>
<?php
for ($i=1; $i<=5; $i++)
  {
  echo "The number is " . $i . "<br>";
  }
?>

</body>
</html>
```

Output:

```
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

The foreach Loop
The foreach loop is used to loop through arrays.

Syntax

```
foreach ($array as $value)
  {
  code to be executed;
  }
```

For every loop iteration, the value of the current array element is assigned to $value (and the array pointer is moved by one) - so on the next loop iteration, you'll be looking at the next array value.

Example

The following example demonstrates a loop that will print the values of the given array:

```
<html>
<body>

<?php
$x=array("one","two","three");
foreach ($x as $value)
  {
  echo $value . "<br>";
  }
?>
</body>
</html>
```

Output:

```
one
two
three
```

The real power of PHP comes from its functions. In PHP,
there are more than 700 built-in functions.

PHP Built-in Functions
For a complete reference and examples of the built-in functions, please visit our PHP Reference.

PHP Functions
In this chapter we will show you how to create your own functions.

To keep the script from being executed when the page loads, you can put it into a function. A function will be executed by a call to the function.
You may call a function from anywhere within a page.

Create a PHP Function
A function will be executed by a call to the function.

Syntax
function *functionName*()
{
code to be executed;
}
PHP function guidelines:

- Give the function a name that reflects what the function does
- The function name can start with a letter or underscore (not a number)

Example
A simple function that writes my name when it is called:

```
<html>
<body>

<?php
function writeName()
{
echo "Kai Jim Refsnes";
}

echo "My name is ";
writeName();
?>

</body>
</html>
```

Output:

My name is Kai Jim Refsnes

PHP Functions - Adding parameters
To add more functionality to a function, we can add parameters. A parameter is just like a variable.

Parameters are specified after the function name, inside the parentheses.

Example 1

The following example will write different first names, but equal last name:

```
<html>
<body>

<?php
```

```
function writeName($fname)
{

echo $fname . " Refsnes.<br>";
}

echo "My name is ";
writeName("Kai  Jim"); echo
"My sister's name is ";
writeName("Hege");
echo "My brother's name is ";
writeName("Stale");
?>

</body>
</html>
```

Output:

My name is Kai Jim Refsnes.
My sister's name is Hege Refsnes. My
brother's name is Stale Refsnes.

Example 2

The following function has two parameters:

```
<html>
<body>

<?php
function writeName($fname,$punctuation)
{
echo $fname . " Refsnes" . $punctuation . "<br>";
}

echo "My name is ";
writeName("Kai Jim",".");
echo "My sister's name is ";
writeName("Hege","!");
echo "My brother's name is ";
writeName("Ståle","?");
?>

</body>
</html>
```

Output:

My name is Kai Jim Refsnes.
My sister's name is Hege Refsnes! My
brother's name is Ståle Refsnes?

PHP Functions - Return values

To let a function return a value, use the return statement.

Example

```
<html>
<body>

<?php
function add($x,$y)
{
$total=$x+$y;
return $total;
```

```
}

echo "1 + 16 = " . add(1,16);
?>

</body>
</html>
```

Output:

1 + 16 = 17

The PHP $_GET and $_POST variables are used to retrieve information from forms, like user input.

PHP Form Handling
The most important thing to notice when dealing with HTML forms and PHP is that any form element in an HTML page will **automatically** be available to your PHP scripts.

Example

The example below contains an HTML form with two input fields and a submit button:

```
<html>
<body>

<form action="welcome.php" method="post">
Name: <input type="text" name="fname">

Age: <input type="text" name="age">
<input type="submit">
</form>

</body>
</html>
```

When a user fills out the form above and clicks on the submit button, the form data is sent to a PHP file, called "welcome.php":

"welcome.php" looks like this:

```
<html>
<body>

Welcome <?php echo $_POST["fname"]; ?>!<br>
You are <?php echo $_POST["age"]; ?> years old.

</body>
</html>
```

Output could be something like this:

Welcome John!
You are 28 years old.

The PHP $_GET and $_POST variables will be explained in the next chapters.

Form Validation
User input should be validated on the browser whenever possible (by client scripts). Browser validation is faster and reduces the server load.

You should consider server validation if the user input will be inserted into a database. A good way to validate a form on the server is to post the form to itself, instead of jumping to a different page. The user will then get the error messages on the same page as the form. This makes it easier to discover the error.

In PHP, the predefined $_GET variable is used to collect values in a form with method="get".

The $_GET Variable
The predefined $_GET variable is used to collect values in a form with method="get"

Information sent from a form with the GET method is visible to everyone (it will be displayed in the browser's address bar) and has limits on the amount of information to send.

Example

```
<form action="welcome.php" method="get">
Name: <input type="text" name="fname"> Age:
<input type="text" name="age">
<input type="submit">
</form>
```

When the user clicks the "Submit" button, the URL sent to the server could look something like this:

http://www.w3schools.com/welcome.php?fname=Peter&age=37

The "welcome.php" file can now use the $_GET variable to collect form data (the names of the form fields will automatically be the keys in the $_GET array):

```
Welcome <?php echo $_GET["fname"]; ?>.<br>
You are <?php echo $_GET["age"]; ?> years old!
```

When to use method="get"?
When using method="get" in HTML forms, all variable names and values are displayed in the URL.

Note: This method should not be used when sending passwords or other sensitive information!

However, because the variables are displayed in the URL, it is possible to bookmark the page. This can be useful in some cases.

Note: The get method is not suitable for very large variable values. It should not be used with values exceeding 2000 characters.

In PHP, the predefined $_POST variable is used to collect values in a form with method="post".

The $_POST Variable

The predefined $_POST variable is used to collect values from a form sent with method="post".

Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send.

Note: However, there is an 8 MB max size for the POST method, by default (can be changed by setting the post_max_size in the php.ini file).

Example

```
<form action="welcome.php" method="post">
Name: <input type="text" name="fname"> Age:
<input type="text" name="age">
<input type="submit">
</form>
```

When the user clicks the "Submit" button, the URL will look like this:

http://www.w3schools.com/welcome.php

The "welcome.php" file can now use the $_POST variable to collect form data (the names of the form fields will automatically be the keys in the $_POST array):

```
Welcome <?php echo $_POST["fname"]; ?>!<br>
You are <?php echo $_POST["age"]; ?> years old.
```

When to use method="post"?
Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send.

However, because the variables are not displayed in the URL, it is not possible to bookmark the page.

The PHP $_REQUEST Variable
The predefined $_REQUEST variable contains the contents of both $_GET, $_POST, and $_COOKIE.

The $_REQUEST variable can be used to collect form data sent with both the GET and POST methods.

Example
Welcome <?php echo $_REQUEST["fname"]; ?>!
You are <?php echo $_REQUEST["age"]; ?> years old.

What is MySQL?
- MySQL is a database server
- MySQL is ideal for both small and large applications
- MySQL supports standard SQL
- MySQL compiles on a number of platforms
- MySQL is free to download and use

The data in MySQL is stored in database objects called tables.

A table is a collection of related data entries and it consists of columns and rows.

Databases are useful when storing information categorically. A company may have a database with the following tables: "Employees", "Products", "Customers" and "Orders".

PHP + MySQL
- PHP combined with MySQL are cross-platform (you can develop in Windows and serve on a Unix platform)

Database Tables
A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.

Below is an example of a table called "Persons":

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

The table above contains three records (one for each person) and four columns (LastName, FirstName, Address, and City).

Queries
A query is a question or a request.

With MySQL, we can query a database for specific information and have a recordset returned.

Look at the following query:

SELECT LastName FROM Persons

The query above selects all the data in the "LastName" column from the "Persons" table, and will return a recordset like this:

LastName
Hansen
Svendson

Pettersen

Download MySQL Database
If you don't have a PHP server with a MySQL Database, you can download MySQL for free here: http://www.mysql.com/downloads/

Facts About MySQL Database
One great thing about MySQL is that it can be scaled down to support embedded database applications. Perhaps it is because of this reputation that many people believe that MySQL can only handle small to medium-sized systems.

The truth is that MySQL is the de-facto standard database for web sites that support huge volumes of both data and end users (like Friendster, Yahoo, Google).

Look at http://www.mysql.com/customers/ for an overview of companies using MySQL. The free MySQL database is very often used with PHP.

Create a Connection to a MySQL Database
Before you can access data in a database, you must create a connection to the database. In PHP, this is done with the mysql_connect() function.

Syntax

mysql_connect(servername,username,password);

Parameter	Description
servername	Optional. Specifies the server to connect to. Default value is "localhost:3306" username Optional. Specifies the username to log in with. Default value is the name of the user that owns the server process
password	Optional. Specifies the password to log in with. Default is ""

Note: There are more available parameters, but the ones listed above are the most important. Visit our full PHP MySQL Reference for more details.

Example

In the following example we store the connection in a variable ($con) for later use in the script. The "die" part will be executed if the connection fails:

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

// some code
?>
```

Closing a Connection
The connection will be closed automatically when the script ends. To close the connection before, use the mysql_close() function:

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }
// some code
mysql_close($con);
?>
```

A database holds one or multiple tables.

Create a Database

The CREATE DATABASE statement is used to create a database in MySQL.

Syntax

```
CREATE DATABASE database_name
```

To learn more about SQL, please visit our SQL tutorial.

To get PHP to execute the statement above we must use the mysql_query() function. This function is used to send a query or command to a MySQL connection.

Example

The following example creates a database called "my_db":

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

if (mysql_query("CREATE DATABASE my_db",$con))
  {
  echo "Database created";
  }
else
  {
  echo "Error creating database: " . mysql_error();
  }

mysql_close($con);
?>
```

Create a Table

The CREATE TABLE statement is used to create a table in MySQL.

Syntax

```
CREATE                TABLE
table_name (
column_name1      data_type,
column_name2      data_type,
column_name3 data_type,
....
)
```

To learn more about SQL, please visit our SQL tutorial.

We must add the CREATE TABLE statement to the mysql_query() function to execute the command.

Example

The following example creates a table named "Persons", with three columns. The column names will be "FirstName", "LastName" and "Age":

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

// Create database
if (mysql_query("CREATE DATABASE my_db",$con))
  {
```

```
  echo "Database created";
  }
else
  {
  echo "Error creating database: " . mysql_error();
  }

// Create table
mysql_select_db("my_db", $con);
$sql    =    "CREATE    TABLE
Persons (
FirstName    varchar(15),
LastName    varchar(15),
Age int
)";

// Execute query
mysql_query($sql,$con);
mysql_close($con);
?>
```

Important: A database must be selected before a table can be created. The database is selected with the mysql_select_db() function.

Note: When you create a database field of type varchar, you must specify the maximum length of the field, e.g. varchar(15).

The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MySQL, go to our complete Data Types reference.

Primary Keys and Auto Increment Fields
Each table should have a primary key field.

A primary key is used to uniquely identify the rows in a table. Each primary key value must be unique within the table. Furthermore, the primary key field cannot be null because the database engine requires a value to locate the record.

The following example sets the personID field as the primary key field. The primary key field is often an ID number, and is often used with the AUTO_INCREMENT setting.
AUTO_INCREMENT automatically increases the value of the field by 1 each time a new record is added.
To ensure that the primary key field cannot be null, we must add the NOT NULL setting to the field.

Example

```
$sql = "CREATE TABLE
Persons (
personID int NOT NULL
AUTO_INCREMENT, PRIMARY
KEY(personID),
FirstName    varchar(15),
LastName    varchar(15),
Age int
)";

mysql_query($sql,$con);
```

Insert Data Into a Database Table
The INSERT INTO statement is used to add new records to a database table.

Syntax

It is possible to write the INSERT INTO statement in two forms.

The first form doesn't specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
VALUES (value1, value2, value3,...)
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES (value1, value2, value3,...)
```

To learn more about SQL, please visit our SQL tutorial.

To get PHP to execute the statements above we must use the mysql_query() function. This function is used to send a query or command to a MySQL connection.

Example

In the previous chapter we created a table named "Persons", with three columns; "Firstname", "Lastname" and "Age". We will use the same table in this example. The following example adds two new records to the "Persons" table:

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

mysql_select_db("my_db", $con);

mysql_query("INSERT INTO Persons (FirstName, LastName, Age)
VALUES ('Peter', 'Griffin',35)");

mysql_query("INSERT INTO Persons (FirstName, LastName, Age)
VALUES ('Glenn', 'Quagmire',33)");

mysql_close($con);
?>
```

Insert Data From a Form Into a Database
Now we will create an HTML form that can be used to add new records to the "Persons" table.

Here is the HTML form:

```
<html>
<body>

<form action="insert.php" method="post"> Firstname:
<input type="text" name="firstname"> Lastname:
<input type="text" name="lastname"> Age: <input
type="text" name="age">
<input type="submit">
</form>

</body>
</html>
```

When a user clicks the submit button in the HTML form in the example above, the form data is sent to "insert.php".

The "insert.php" file connects to a database, and retrieves the values from the form with the PHP $_POST variables.

Then, the mysql_query() function executes the INSERT INTO statement, and a new record will be added to the "Persons" table.

Here is the "insert.php" page:

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

mysql_select_db("my_db", $con);

$sql="INSERT INTO Persons (FirstName, LastName, Age)
VALUES
('$_POST[firstname]','$_POST[lastname]','$_POST[age]')";

if (!mysql_query($sql,$con))
  {
  die('Error: ' . mysql_error());
  }
echo "1 record added";

mysql_close($con);
?>
```

The WHERE clause

The WHERE clause is used to extract only those records that fulfill a specified criterion.

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator value
```

To learn more about SQL, please visit our SQL tutorial.

To get PHP to execute the statement above we must use the mysql_query() function. This function is used to send a query or command to a MySQL connection.

Example

The following example selects all rows from the "Persons" table where "FirstName='Peter'":

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

mysql_select_db("my_db", $con);

$result = mysql_query("SELECT * FROM Persons
WHERE FirstName='Peter'");

while($row = mysql_fetch_array($result))
  {
  echo $row['FirstName'] . " " . $row['LastName'];
  echo "<br>";
  }
?>
```

The output of the code above will be:

Peter Griffin

The ORDER BY Keyword

The ORDER BY keyword is used to sort the data in a recordset.

The ORDER BY keyword sort the records in ascending order by default.

If you want to sort the records in a descending order, you can use the DESC keyword.

Syntax

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s) ASC|DESC
```

To learn more about SQL, please visit our SQL tutorial.

Example

The following example selects all the data stored in the "Persons" table, and sorts the result by the "Age" column:

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

mysql_select_db("my_db", $con);
$result = mysql_query("SELECT * FROM Persons ORDER BY age");
while($row = mysql_fetch_array($result))
  {
  echo $row['FirstName']; echo "
  " . $row['LastName'];
  echo " " . $row['Age'];
  echo "<br>";
  }

mysql_close($con);
?>
```

The output of the code above will be:

Glenn Quagmire 33
Peter Griffin 35

Order by Two Columns

It is also possible to order by more than one column. When ordering by more than one column, the second column is only used if the values in the first column are equal:

```
SELECT column_name(s)
FROM table_name
ORDER BY column1, column2
```

Update Data In a Database

The UPDATE statement is used to update existing records in a table.

Syntax

```
UPDATE table_name
SET column1=value, column2=value2,...
WHERE some_column=some_value
```

Note: Notice the WHERE clause in the UPDATE syntax. The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

To learn more about SQL, please visit our SQL tutorial.

To get PHP to execute the statement above we must use the mysql_query() function. This function is used

to send a query or command to a MySQL connection.

Example

Earlier in the tutorial we created a table named "Persons". Here is how it looks:

FirstName LastName Age

Peter Griffin 35

Glenn Quagmire 33

The following example updates some data in the "Persons" table:

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }
mysql_select_db("my_db", $con);
mysql_query("UPDATE Persons SET Age=36
WHERE FirstName='Peter' AND LastName='Griffin'");

mysql_close($con);
?>
```

After the update, the "Persons" table will look like this:

FirstName LastName Age

Peter Griffin 36

Glenn Quagmire 33

Delete Data In a Database
The DELETE FROM statement is used to delete records from a database table.

Syntax

```
DELETE FROM table_name
WHERE some_column = some_value
```

Note: Notice the WHERE clause in the DELETE syntax. The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

To learn more about SQL, please visit our SQL tutorial.

To get PHP to execute the statement above we must use the mysql_query() function. This function is used to send a query or command to a MySQL connection.

Example

Look at the following "Persons" table:

FirstName LastName Age

Peter Griffin 35

Glenn Quagmire 33

The following example deletes all the records in the "Persons" table where

LastName='Griffin':

```
<?php
$con = mysql_connect("localhost","peter","abc123"); if
(!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

mysql_select_db("my_db", $con);
mysql_query("DELETE FROM Persons WHERE LastName='Griffin'");
mysql_close($con);
?>
```

After the deletion, the table will look like this:

FirstName LastName Age

Glenn Quagmire 33

Create an ODBC Connection

With an ODBC connection, you can connect to any database, on any computer in your network, as long as an ODBC connection is available.

Here is how to create an ODBC connection to a MS Access Database:

1. Open the **Administrative Tools** icon in your Control Panel.
2. Double-click on the **Data Sources (ODBC)** icon inside.
3. Choose the **System DSN** tab.
4. Click on **Add** in the System DSN tab.
5. Select the **Microsoft Access Driver**. Click **Finish.**
6. In the next screen, click **Select** to locate the database.
7. Give the database a **Data Source Name (DSN)**.
8. Click **OK**.

Note that this configuration has to be done on the computer where your web site is located. If you are running Internet Information Server (IIS) on your own computer, the instructions above will work, but if your web site is located on a remote server, you have to have physical access to that server, or ask your web host to to set up a DSN for you to use.

Connecting to an ODBC

The odbc_connect() function is used to connect to an ODBC data source. The function takes four parameters: the data source name, username, password, and an optional cursor type.

The odbc_exec() function is used to execute an SQL statement.

Example

The following example creates a connection to a DSN called northwind, with no username and no password. It then creates an SQL and executes it:

```
$conn=odbc_connect('northwind',",");
$sql="SELECT * FROM customers";
$rs=odbc_exec($conn,$sql);
```

Retrieving Records

The odbc_fetch_row() function is used to return records from the result-set. This function returns true if it is able to return rows, otherwise false.

The function takes two parameters: the ODBC result identifier and an optional row number:

```
odbc_fetch_row($rs)
```

Retrieving Fields from a Record

The odbc_result() function is used to read fields from a record. This function takes two parameters: the ODBC result identifier and a field number or name.

The code line below returns the value of the first field from the record:

```
$compname=odbc_result($rs,1);
```

The code line below returns the value of a field called "CompanyName":

```
$compname=odbc_result($rs,"CompanyName");
```

Closing an ODBC Connection
The odbc_close() function is used to close an ODBC connection.

```
odbc_close($conn);
```

An ODBC Example
The following example shows how to first create a database connection, then a result-set, and then display the data in an HTML table.

```
<html>
<body>

<?php
$conn=odbc_connect('northwind',",''); if
(!$conn)
  {exit("Connection Failed: " . $conn);}
$sql="SELECT * FROM customers";
```

```
$rs=odbc_exec($conn,$sql); if
(!$rs)
  {exit("Error in SQL");}
echo "<table><tr>";
echo "<th>Companyname</th>"; echo
"<th>Contactname</th></tr>"; while
(odbc_fetch_row($rs))
  {
  $compname=odbc_result($rs,"CompanyName");
  $conname=odbc_result($rs,"ContactName"); echo
  "<tr><td>$compname</td>";
  echo "<td>$conname</td></tr>";
  }
odbc_close($conn);
echo "</table>";
?>

</body>
</html>
```

UNIT-V
PERL

- Introduction to PERL
- Operators and if statements
- Program design and control structures
- Arrays
- Hashs and File handling
- Regular expressions
- Subroutines
- Retrieving documents from the web with Perl.

What is Perl?

Perl is a programming language. Perl stands for Practical Report and Extraction Language. You'll notice people refer to 'perl' and "Perl". "Perl" is the programming language as a whole whereas 'perl' is the name of the core executable. There is no language called "Perl5" -- that just means "Perl version 5". Versions of Perl prior to 5 are very old and very unsupported.

Some of Perl's many strengths are:

- **Speed of development.**
- **Power**
- **Usability**
- **Portability**
- **Editing tools**
- **Price.**

What can I do with Perl ?

- It is the most popular language for CGI programming for many reasons, most of which are mentioned above.
- Perl powers a good deal of the Internet.
  - A few examples of how I use Perl to ease NT sysadmin life::
- **User account creation**. If you have a text file with the user's names in it, that is all One and create the account, plus create and share the home directory, and set the permissions.
- **Event log munging.** NT has great Event Logging. Not so great Event Reading. You can use Perl to create reports on the event logs from multiple NT servers.

Perl History

Version	Date	Version Details
Perl 0		Introduced Perl to Larry Wall's office associates
Perl 1	Jan 1988	Introduced Perl to the world
Perl 2	Jun 1988	Introduced Harry Spencer's regular expression package
Perl 3	Oct 1989	Introduced the ability to handle binary data
Perl 4	Mar 1991	Introduced the first ―Camel‖ book (Programming Perl, by Larry Wall, Tom Christiansen, and Randal L Schwartz; O'Reilly & Associates). The book drove the name change, just so it could refer to Perl 4, instead of Perl 3.
Feb 1993	Feb 1993	The last stable release of Perl 4
Perl 5	Oct 1994	The first stable release of Perl 5, which introduced a number of new features and a complete rewrite.
Perl .005_02	Aug 1998	The next major stable release
Perl .005_03	Mar 1999	The last stable release before 5.6
Perl 5.6	Mar 2000	Introduced unified **fork** support, better threading, an updated Perl compiler, and the **our** keyword

Main Perl Features:

1. **Perl Is Free:**

Perl's source code is open and free anybody can download the C source that constitutes a Perl

interpreter.

2. **Perl Is Simple to Learn, Concise, and Easy to Read:**
   It has a syntax similar to C and shell script, among others, but with a less restrictive format.
3. **Perl Is Fast**
   Compared to most scripting languages, this makes execution almost as fast as compiled C code. But, because the code is still interpreted, there is no compilation process.
4. **Perl Is Extensible**
   You can write Perl-based packages and modules that extend the functionality of the language.
5. **Perl Has Flexible Data Types**
   You can create simple variables that contain text or numbers, and Perl will treat the variable data accordingly at the time it is used.
6. **Perl Is Object Oriented**
   Perl supports all of the object-oriented features—inheritance, polymorphism, and encapsulation.
7. **Perl Is Collaborative**
   There is a huge network of Perl programmers worldwide. Most programmers supply, and use, the modules and scripts available via CPAN, the Comprehensive Perl Archive Network

Perl is interpreter or compiler?

Neither, and both. Perl is a scripting language. There is a tool, called perl, intended to run programs written in the perl language.

"Compiled" languages are ones like C and C++, where you have to take the source code, compile it into an executable file, and THEN run it.

"Interpreted" languages, like Perl, PHP, and Ruby, are ones which do NOT require pre-compiling.

They are generally compiled on-the-fly (which is what the perl command-line tool does) into *opcodes*, and then run. So, Perl is an interpreted language because a tool reads the source code and immediately runs it.Perl is a compiler because it has to compile that source code before it can be run while it's being interpreted.

Installing and using Perl

- Perl was developed by Larry Wall. It started out as a scripting language to supplement *rn*, the USENET reader. It available on virtually every computer platform.
- Perl is an interpreted language that is optimized for string manipulation, I/O, and system tasks. It has built in for most of the functions in section 2 of the UNIX manuals -- very popular with sys administrators. It incorporates syntax elements from the *Bourne shell*,*csh*, *awk*, *sed*, *grep*, and C. It provides a quick and effective way to write interactive web applications

Writing a Perl Script

Perl scripts are just text files, so in order to actually "write" the script, all you need to do is create a text file using your favorite text editor. Once you've written the script, you tell Perl to execute the text file you created.

Under Unix, you would use

$ perl myscript.pl

and the same works under Windows:

C:\> perl myscript.pl

Under Mac OS, you need to drag and drop the file onto the *MacPerl* application. Perl scripts have a *.pl* extension, even under Mac OS and Unix.

Perl Under Unix

The easiest way to install Perl modules on Unix is to use the CPAN module. For example:

shell> perl -MCPAN -e shell
cpan> install DBI
cpan> install DBD::mysql

The DBD::mysql installation runs a number of tests. These tests attempt to connect to the local MySQL server using the default user name and password. (The default user name is your login name on Unix, and ODBC on Windows. The default password is "no password.") If you cannot connect to the server with those values (for example, if your account has a password), the tests fail. You can use force install DBD::mysql to ignore the failed tests.

DBI requires the Data::Dumper module. It may be installed; if not, you should install it before

installing DBI.

Perl Under Windows

1. Log on to the Web server computer as an administrator.
2. Download the ActivePerl installer from the following ActiveState Web site: http://www.activestate.com/ (http://www.activestate.com/)
3. Double-click the **ActivePerl** installer.
4. After the installer confirms the version of ActivePerl that it is going to be installed, click **Next**.
5. If you agree with the terms of the license agreement, click **I accept the terms in the license agreement**, and then click **Next**. Click **Cancel** if you do not accept the license agreement. If you do so, you cannot continue the installation.
6. To install the whole ActivePerl distribution package (this step is recommended), click **Next** to continue the installation. The software is installed in the default location (typically C:\Perl).
7. To customize the individual components or to change the installation folder, follow the instructions that appears on the screen.
8. When you are prompted to confirm the addition features that you want to configure during the installation, click any of the following settings, and then click **Next**:
    a. **Add Perl to the PATH environment variable**: Click this setting if you want to use Perl in a command prompt without requiring the full path to the Perl interpreter.
    b. **Create Perl file extension association**: Click this setting if you want to allow Perl scripts to be automatically run when you use a file that has the Perl file name extension (.pl) as a command name.
    c. **Create IIS script mapping for Perl**: Click this setting to configure IIS to identify Perl scripts as executable CGI programs according to their file name extension.
    d. **Create IIS script mapping for Perl ISAPI**: Click this setting to use Perl scripts as an ISAPI filter.
9. Click **Install** to start the installation process.
10. After the installation has completed, click **Finish**.

Comments in Perl

- Comments can be used to make program user friendly and they are simply skipped by the interpreter without impacting the code functionality. For example, in the above program, a line starting with hash **#** is a comment.
- Simply saying comments in Perl start with a hash symbol and run to the end of the line −

```
# This is a comment in perl.
```

Lines starting with = are interpreted as the start of a section of embedded documentation (pod), and all subsequent lines until the next =cut are ignored by the compiler. Following is the example −

```
#!/usr/bin/perl (shebang)
# This is a single line comment
print "Hello, world\n";
=begin comment
This is all part of multiline comment.
You can use as many lines as you like
These comments will be ignored by the
compiler until the next =cut is encountered.
=cut
```

Whitespaces in Perl

A Perl program does not care about whitespaces. Following program works perfectly fine −

```
#!/usr/bin/perl
print      "Hello, world\n";
```

But if spaces are inside the quoted strings, then they would be printed as is. For example −

```
#!/usr/bin/perl
# This would print with a line break in the middle
print "Hello
       world\n";
```

Single and Double Quotes in Perl

You can use double quotes or single quotes around literal strings as follows −

```
#!/usr/bin/perl
print "Hello, world\n";
print 'Hello, world\n';
```

This will produce the following result −

```
Hello, world
Hello, world\n$
```

There is an important difference in single and double quotes.

Only double quotes **interpolate** variables and special characters such as newlines \n, whereas single quote does not interpolate any variable or special character. Check below example where we are using $a as a variable to store a value and later printing that value −

```
#!/usr/bin/perl
$a = 10;
print "Value of a = $a\n";
print 'Value of a = $a\n';
```

This will produce the following result −

```
Value of a = 10
Value of a = $a\n$
```

use strict

The use strict statement is called pragma and it can be placed at the beginning of the script like this:

```
#!/usr/local/bin/perl
use strict;
```

Example:

If you use “use strict” but don’t declare a variable.

```
#!/usr/local/bin/perl
use strict;
$s = "Hello!\n";
print $s;
```

It would throw this error:

```
Global symbol "$s" requires explicit package name at st.pl line 3.
Global symbol "$s" requires explicit package name at st.pl line 4.
Execution of st.pl aborted due to compilation errors.
```

To avoid the error you must declare the variable using my keyword.

```
#!/usr/local/bin/perl
use strict;
my $s = "Hello!\n";
print $s;
```

Output:

```
Hello!
```

use warnings

This is another pragma, together they are used like this:

```
#!/usr/local/bin/perl
use strict;
use warnings;
```

Note: use warnings pragma got introduced in Perl 5.6 so if you are using Perl 5.6 or later you are good to go. In case you are using older version you can turn on the warning like this: By putting -w on the ‘shebang’ line.

```
#!/usr/local/bin/perl -w
```

This would work everywhere even on Perl 5.6 or later.

What is the use of “use warnings”?

It helps you find typing mistakes, it warns you whenever it sees something wrong with your program. It would help you find mistakes in your program faster.

Perl print() and say()

Perl print() function is one of the most commonly used function by the Perl users. It will print a string, a number, a variable or anything it gets as its arguments.

Perl print syntax is given below:

```
1. #!/usr/bin/perl
2. print "Welcome to perl!!";
3. print "This is our Perl tutorial!!";
```

Output:

```
Welcome to Perl!! This is our Perl tutorial!!
```

```
1. #!/usr/bin/perl
2. print "Welcome to perl!!\n";
3. print "This is our Perl tutorial!!\n";
```

Output:

```
Welcome to perl!!
This is our Perl tutorial!!
```

Perl print Example with Variables

To print the variables value, you need to pass the variable in print function.

```
1. #!/usr/bin/perl
2. $site = 'perl';
3. print "Welcome to $site!!\n";
4. print "$site provides you all type of tutorials!!\n";
```

Output:

```
Welcome to perl!!
perl provides you all type of tutorials!!
```

Perl print Example with Single and Double Quotes

Perl print function does not interpolate inside single quotes. It means it prints the characters inside single quote as it is. It neither evaluates the variables value nor does it interpret the escaping characters.

When the above example runs inside a single quote, you will notice the following changes.

```
#!/usr/bin/perl
$site = 'perl';
print 'Welcome to $site!!\n';
print '$site provides you all type of tutorials!!\n';
```

Output:

```
Welcome to $site!! \n$site provides you all type of tutorials!!\n
```

Perl say()

The say() function is not supported by the older perl versions. It acts like print() function with only difference that it automatically adds a new line at the end without mentioning (\n).

Perl say() syntax is given below: say "";

Perl say() Example

Let's see a simple example of perl say function.

```
#!/usr/bin/perl
use 5.32.0;
say "Welcome to perl!!";
say "This is our Perl tutorial!!";
```

Output:

```
Welcome to perl!! This is our Perl tutorial!!
```

Perl STDIN

In Perl programming, we can get input from standard console using <STDIN>. It stands for Standard Input. It can be abbreviated by <>. So,

```
my $name = <STDIN>;
```

is equivalent to:

```
my $name = <>;
```

Perl Input from User using <STDIN>

Let's see an example where we are getting data from the user using Standard Input <STDIN>.

```
use 5.32.0;
use strict;
use warnings;
say "Enter your Name:";
```

```
my $name = <STDIN>;
say "Welcome $name at Perl Scripting.";
```

Output:

```
Enter your Name:
John
Welcome John
  at Perl Scripting.
```

As you can see in the above output, a new line is added after printing the name. To overcome this problem, add the **chomp** function with $name as shown below.

```
use 5.32.0;
use strict;
use warnings;
say "Enter your Name:";
my $name = <STDIN>;
chomp $name;
print "Welcome $name at perl.\n";
```

Output:

```
Enter your Name:
John
Welcome John at perl.
```

Escaping Characters

Perl uses the backslash (\) character to escape any type of character that might interfere with our code. Let's take one example where we want to print double quote and $ sign −

```
#!/usr/bin/perl
$result = "This is \"number\"";
print "$result\n";
print "\$result\n";
```

Perl Identifiers

A Perl identifier is a name used to identify a variable, function, class, module, or other object. A Perl variable name starts with either $, @ or % followed by zero or more letters, underscores, and digits (0 to 9).

Perl does not allow punctuation characters such as @, $, and % within identifiers. Perl is a **case sensitive** programming language. Thus **$Manpower** and **$manpower** are two different identifiers in Perl.

Perl Variables

Perl language is a loosely typed language, the Perl interpreter itself chooses the data type. Hence, there is no need to specify data type in Perl programming.

There are basically three data types in Perl:

- **Scalars**: Perl scalars are a single data item. They are simple variables, preceded by a ($) sign. A scalar can be a number, a reference (address of a variable) or a string.
- **Arrays**: Perl arrays are an ordered list of scalars. They are preceded by (@) sign and accessed by their index number which starts with 0.
- **Hashes**: Perl hashes are an unordered collection of key-value pairs. They are preceded by (%) sign and accessed using keys.

Sr.No.	Types & Description
1	**Scalar** Scalars are simple variables. They are preceded by a dollar sign ($). A scalar is either a number, a string, or a reference. A reference is actually an address of a variable
2	**Arrays** Arrays are ordered lists of scalars that you access with a numeric index, which starts with 0. They are preceded by an "at" sign (@).

3	**Hashes** Hashes are unordered sets of key/value pairs that you access using the keys as subscripts. They are preceded by a percent sign (%).

Perl Literals

In Perl there are two different types of scalar constants:

1. Numeric literal
2. String literal

Perl Numeric Literals

Perl numeric literals are numbers. Perl stores number internally as either signed integers or floating-point values.

Perl numeric literals can be assigned following types of formats:

Number	Type
526	Integer
5.5	Floating point
5e10	Scientific notation
5.5E	Scientific notation
5_549_63	A large number
010101	Binary number
0175	Octal number
AF0230	Hexadecimal number

- Integers are a group of consecutive digits.
- Floating-point numbers contain a decimal in between. A number containing '0' value on the right hand side of the number (234.00) is also a floating point number .
- A number containing an exponent notation (e or E) is the scientific notation.
- Commas are not allowed into a numeric literal but you can use underscores (_) instead of commas. Perl will remove underscores while using this value.
- Combination of 0 and 1 represents a binary number with base 2.
- Number with a leading 0 comes in the category of octal numbers with base 8.
- Number containing alphabets (a, b, c, d, e, f) are hexadecimal numbers with base 16.

Perl String Literals

- Perl string literals contain an empty string, ASCII text, ASCII with high bits or binary data. There is no limit in a string to contain data. They are surrounded by either a single quote (′) or double quote (″).
- Variable interpolation is allowed in double quote string but not in single quote string. Also special characters preceding with backslash (\) are supported by double quote strings only.

Escape Characters in string literals

Characters	Purpose
\n	Denotes newline
\r	Denotes carriage return
\ t	Denotes horizontal tab
\v	Denotes vertical tab

\Q	Backslash following all non-alphanumeric character
\a	Denotes alert
\f	Denotes form feed
\b	Denotes backspace
\u	Change next character to uppercase
\U	change all following characters to uppercase
\l	Change next character to lowercase
\L	Change all following character to lowercase
\E	Denotes \U, \L, \Q
\cX	Controls characters, X is a variable
\0nn	Create octal formatted numbers
\xnn	Create hexadecimal formatted numbers
\\	Denote backslash
\'	Denote single quote
\"	Denote double quote

Example:

```
#!/usr/bin/perl
# This is case of interpolation.
$str = "Welcome to \nperlscript.com!";
print "$str\n";
# This is case of non-interpolation.
$str = 'Welcome to \nperlscript.com!';
print "$str\n";
# Only W will become upper case.
$str = "\uwelcome to perlscript.com!";
print "$str\n";
# Whole line will become capital.
$str = "\UWelcome to perlscript.com!";
print "$str\n";
# A portion of line will become capital.
$str = "Welcome to \Uperlscript\E.com!";
print "$str\n";
# Backsalash non alpha-numeric including spaces.
$str = "\QWelcome to perlscript's family";
print "$str\n";
```

Perl Variable Declaration

The equal sign (=) is used to assign values to variables. At the left of (=) is the variable name and on the right it is the value of the variable.

1. $name = "Anastasia";
2. $rank = "9th";
3. $marks = 756.5;

Here we have created three variables $name, $rank and $marks.

Creating Variables

Perl variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

```
$age = 25;             # An integer assignment
$name = "John Paul";   # A string
$salary = 1445.50;     # A floating point
```

Perl use strict

If you are using **use strict** statement in a program, then you have to declare your variable before using it. It is mandatory. Otherwise you'll get an error.

The $a and $b are special variables used in perl sort function. There is no need to declare these two variables. So it is recommended not to use these two variables except in connection to **sort.**

Variables can be declared using **my, our, use vars, state and $person::name** (explicit package name). Although, they all have different meanings.

Example

```
use 5.32.0;
use strict;
my $x = 23;
say $x;
state $name = "Anastasia";
say $name;
our $rank = "9th";
say $rank;
use vars qw($marks);
$marks = 756.5;
say $marks;
$Person::name = 'John';
say $Person::name;
$a = 1224365;
say $a;
 $b = "Welcome at perl";
say $b;
```

Variable Context

Perl treats same variable differently based on Context, i.e., situation where a variable is being used.

```
#!/usr/bin/perl
@names = ('John Paul', 'Lisa', 'Kumar');
@copy = @names;
$size = @names;
print "Given names are : @copy\n";
print "Number of names are : $size\n";
```

Scalar Variables

A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page. Simply saying it could be anything, but only a single thing.

Here is a simple example of using scalar variables −

```
#!/usr/bin/perl
$age = 25;             # An integer assignment
$name = "John Paul";   # A string
$salary = 1445.50;     # A floating point
print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

Example:

```
use strict;
use warnings;
use 5.32.0;
#Declairing and assigning value together
```

```
my $color = "Red";
say $color;
#Declairing the variable first and then assigning value
my $city;
$city = "Delhi";
say $city;
```

Output:

```
Red
Delhi
```

String Scalars

Following example demonstrates the usage of various types of string scalars. Notice the difference between single quoted strings and double quoted strings −

```
#!/usr/bin/perl
$var = "This is string scalar!";
$quote = 'I m inside single quote - $var';
$double = "This is inside single quote - $var";
$escape = "This example of escape -\tHello, World!";
print "var = $var\n";
print "quote = $quote\n";
print "double = $double\n";
print "escape = $escape\n";
```

Multiline Strings

If you want to introduce multiline strings into your programs, you can use the standard single quotes as below −

```
#!/usr/bin/perl
$string = 'This is
a multiline
string';
print "$string\n";
```

You can use "here" document syntax as well to store or print multilines as below −

```
#!/usr/bin/perl
print <<EOF;
This is
a multiline
string
EOF
```

This will also produce the same result −

```
This is
a multiline
string
```

Perl Scalar Operations

In this example we'll perform different operations with two scalar variables **$x** and **$y**. In Perl, operator tells operand how to behave.

Example:

```
use strict;
use warnings;
use 5.010;
my $x = 5;
say $x;
my $y = 3;
say $y;
say $x + $y;
say $x . $y;
say $x x $y;
```

Perl Special Literals

There are three special literals in Perl:

__FILE__ : It represent the current file name.

__LINE__ : It represent the current line number.
__PACKAGE__ : It represent the package name at that point in your program.

Example:

```
use strict;
use warnings;
use 5.010;
#!/usr/bin/perl
print "File name ". __FILE__ . "\n";
print "Line Number " . __LINE__ ."\n";
print "Package " . __PACKAGE__ ."\n";
# they can?t be interpolated
print "__FILE__ __LINE__ __PACKAGE__\n";
```

Perl String Context

Perl automatically converts strings to numbers and numbers
For example, 5 is same as "5", 5.123 is same as "5.123".
But if a string has some characters other than numbers, how would they behave in the arithmetic operations.

Example:

```
use strict;
use warnings;
use 5.010;
my $x = "5";
my $y = "2cm";
say $x + $y;
say $x . $y;
say $x x $y;
```

Perl undef

If you'll not define anything in the variable, it is considered as **undef**. In numerical context, it acts as **0**. In string context, it acts as **empty** string.

```
use strict;
use warnings;
use 5.010;
my $x = "5";
my $y;
say $x + $y;
say $x . $y;
say $x x $y;
if (defined $y) {
  say "defined";
} else {
  say "NOT";
}
```

Output:

```
Use of uninitialized value $y in addition (+) at hw.pl line 9.
5
Use of uninitialized value $y in concatenation (.) or string at hw.pl line 10.
5
Use of uninitialized value $y in repeat (x) at hw.pl line 11.
NOT
```

PERL ARRAYS

- An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign.
- To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

Here is a simple example of using the array variables –
Syntax @variablename=value;

```
#!/usr/bin/perl
@ages = (25, 30, 40);
@names = ("John Paul", "Lisa", "Kumar");
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

Array Creation

Array variables are prefixed with the @ sign and are populated using either parentheses or the qw operator. For example −

```
@array = (1, 2, 'Hello');
@array = qw/This is an array/;
```

The second line uses the qw// operator, which returns a list of strings, separating the delimited string by white space. In this example, this leads to a four-element array; the first element is 'this' and last (fourth) is 'array'. This means that you can use different lines as follows −

```
@days = qw/Monday
Tuesday
...
Sunday/;
```

You can also populate an array by assigning each value individually as follows −

```
$array[0] = 'Monday';
...
$array[6] = 'Sunday';
```

Accessing Array Elements

When accessing individual elements from an array, you must prefix the variable with a dollar sign ($) and then append the element index within the square brackets after the name of the variable. For example −

```
#!/usr/bin/perl
@days = qw/Mon Tue Wed Thu Fri Sat Sun/;
print "$days[0]\n";
print "$days[1]\n";
print "$days[2]\n";
print "$days[6]\n";
print "$days[-1]\n";
print "$days[-7]\n";
```

Array indices start from zero, so to access the first element you need to give 0 as indices. You can also give a negative index, in which case you select the element from the end, rather than the beginning, of the array. This means the following −

```
print $days[-1]; # outputs Sun
print $days[-7]; # outputs Mon
```

Sequential Number Arrays

Perl offers a shortcut for sequential numbers and letters. Rather than typing out each element when counting to 100 for example, we can do something like as follows −

```
#!/usr/bin/perl
@var_10 = (1..10);
@var_20 = (10..20);
@var_abc = (a..z);
print "@var_10\n";   # Prints number from 1 to 10
print "@var_20\n";   # Prints number from 10 to 20
print "@var_abc\n";  # Prints number from a to z
```

Here double dot (..) is called **range operator**. This will produce the following result −

```
1 2 3 4 5 6 7 8 9 10
10 11 12 13 14 15 16 17 18 19 20
```

a b c d e f g h i j k l m n o p q r s t u v w x y z

Array Size

The size of an array can be determined using the scalar context on the array - the returned value will be the number of elements in the array −

```
@array = (1,2,3);
print "Size: ",scalar @array,"\n";
```

The value returned will always be the physical size of the array, not the number of valid elements. You can demonstrate this, and the difference between scalar @array and $#array, using this fragment is as follows −

```
#!/usr/bin/perl
@array = (1,2,3);
$array[50] = 4;
$size = @array;
$max_index = $#array;
print "Size:  $size\n";
print "Max Index: $max_index\n";
```

Adding and Removing Elements in Array

Perl provides a number of useful functions to add and remove elements in an array. You may have a question what is a function? So far you have used **print** function to print various values. Similarly there are various other functions or sometime called sub-routines, which can be used for various other functionalities.

Sr.No.	Types & Description
1	**push @ARRAY, LIST** Pushes the values of the list onto the end of the array.
2	**pop @ARRAY** Pops off and returns the last value of the array.
3	**shift @ARRAY** Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down.
4	**unshift @ARRAY, LIST** Prepends list to the front of the array, and returns the number of elements in the new array.

```
#!/usr/bin/perl
# create a simple array
@coins = ("Quarter","Dime","Nickel");
print "1.\@coins  = @coins\n";
# add one element at the end of the array
push(@coins, "Penny");
print "2. \@coins  = @coins\n";
# add one element at the beginning of the array
unshift(@coins, "Dollar");
print "3. \@coins  = @coins\n";
# remove one element from the last of the array.
pop(@coins);
print "4. \@coins  = @coins\n";
# remove one element from the beginning of the array.
shift(@coins);
print "5. \@coins  = @coins\n";
```

Slicing Array Elements

You can also extract a "slice" from an array - that is, you can select more than one item from an array in order to produce another array.

```
#!/usr/bin/perl
@days = qw/Mon Tue Wed Thu Fri Sat Sun/;
```

```
@weekdays = @days[3,4,5];
print "@weekdays\n";
```

```
#!/usr/bin/perl
@days = qw/Mon Tue Wed Thu Fri Sat Sun/;
@weekdays = @days[3..5];
print "@weekdays\n";
```

Replacing Array Elements

Now we are going to introduce one more function called **splice()**, which has the following syntax −

```
splice @ARRAY, OFFSET [ , LENGTH [ , LIST ] ]
```

This function will remove the elements of @ARRAY designated by OFFSET and LENGTH, and replaces them with LIST, if specified. Finally, it returns the elements removed from the array. Following is the example −

```
#!/usr/bin/perl
@nums = (1..20);
print "Before - @nums\n";
splice(@nums, 5, 5, 21..25);
print "After - @nums\n";
```

Transform Strings to Arrays

Let's look into one more function called **split()**, which has the following syntax −

split [PATTERN [, EXPR [, LIMIT]]]

This function splits a string into an array of strings, and returns it. If LIMIT is specified, splits into at most that number of fields. If PATTERN is omitted, splits on whitespace. Following is the example −

```
#!/usr/bin/perl
# define Strings
$var_string = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$var_names = "Larry,David,Roger,Ken,Michael,Tom";
# transform above strings into arrays.
@string = split('-', $var_string);
@names  = split(',', $var_names);
print "$string[3]\n";  # This will print Roses
print "$names[4]\n";   # This will print Michael
```

Transform Arrays to Strings

We can use the **join()** function to rejoin the array elements and form one long scalar string. This function has the following syntax −

join EXPR, LIST

This function joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns the string. Following is the example −

```
#!/usr/bin/perl
# define Strings
$var_string = "Rain-Drops-On-Roses-And-Whiskers-On-Kittens";
$var_names = "Larry,David,Roger,Ken,Michael,Tom";
# transform above strings into arrays.
@string = split('-', $var_string);
@names  = split(',', $var_names);
$string1 = join( '-', @string );
$string2 = join( ',', @names );
print "$string1\n";
print "$string2\n";
```

Sorting Arrays

The **sort()** function sorts each element of an array according to the ASCII Numeric standards. This function has the following syntax −

sort [SUBROUTINE] LIST

This function sorts the LIST and returns the sorted array value. If SUBROUTINE is specified then specified logic inside the SUBTROUTINE is applied while sorting the elements.

```
#!/usr/bin/perl
# define an array
@foods = qw(pizza steak chicken burgers);
print "Before: @foods\n";
# sort this array
@foods = sort(@foods);
print "After sorting: @foods\n";
```

The $[Special Variable

So far you have seen simple variable we defined in our programs and used them to store and print scalar and array values. Perl provides numerous special variables, which have their predefined meaning.

We have a special variable, which is written as **$[**. This special variable is a scalar containing the first index of all arrays. Because Perl arrays have zero-based indexing, $[will almost always be 0. But if you set $[to 1 then all your arrays will use on-based indexing. It is recommended not to use any other indexing other than zero. However, let's take one example to show the usage of $[variable −

```
#!/usr/bin/perl
# define an array
@foods = qw(pizza steak chicken burgers);
print "Foods: @foods\n";
# Let's reset first index of all the arrays.
$[ = 1;
print "Food at \@foods[1]: $foods[1]\n";
print "Food at \@foods[2]: $foods[2]\n";
```

This will produce the following result −

```
Foods: pizza steak chicken burgers
Food at @foods[1]: pizza
Food at @foods[2]: steak
```

Merging Arrays

Because an array is just a comma-separated sequence of values, you can combine them together as shown below −

```
#!/usr/bin/perl
@numbers = (1,3,(4,5,6));
print "numbers = @numbers\n";
```

This will produce the following result −

```
numbers = 1 3 4 5 6
```

The embedded arrays just become a part of the main array as shown below −

```
#!/usr/bin/perl
@odd = (1,3,5);
@even = (2, 4, 6);
@numbers = (@odd, @even);
print "numbers = @numbers\n";
```

This will produce the following result −

```
numbers = 1 3 5 2 4 6
```

Selecting Elements from Lists

The list notation is identical to that for arrays. You can extract an element from an array by appending square brackets to the list and giving one or more indices −

```
#!/usr/bin/perl
$var = (5,4,3,2,1)[4];
print "value of var = $var\n"
```

This will produce the following result −

```
value of var = 1
```

Similarly, we can extract slices, although without the requirement for a leading @ character −

```
#!/usr/bin/perl
@list = (5,4,3,2,1)[1..3];
print "Value of list = @list\n";
```

This will produce the following result −

Value of list = 4 3 2

PERL HASHES

A hash is a set of **key/value** pairs. Hash variables are preceded by a percent (%) sign. To refer to a single element of a hash, you will use the hash variable name preceded by a "$" sign and followed by the "key" associated with the value in curly brackets..

```
#!/usr/bin/perl
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
print "\$data{'John Paul'} = $data{'John Paul'}\n";
print "\$data{'Lisa'} = $data{'Lisa'}\n";
print "\$data{'Kumar'} = $data{'Kumar'}\n";
```

Creating Hashes

Hashes are created in one of the two following ways. In the first method, you assign a value to a named key on a one-by-one basis −

```
$data{'John Paul'} = 45;
$data{'Lisa'} = 30;
$data{'Kumar'} = 40;
```

In the second case, you use a list, which is converted by taking individual pairs from the list: the first element of the pair is used as the key, and the second, as the value. For example −

```
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
```

For clarity, you can use => as an alias for , to indicate the key/value pairs as follows −

```
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
```

Here is one more variant of the above form, have a look at it, here all the keys have been preceded by hyphen (-) and no quotation is required around them −

```
%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
```

But it is important to note that there is a single word, i.e., without spaces keys have been used in this form of hash formation and if you build-up your hash this way then keys will be accessed using hyphen only as shown below.

```
$val = %data{-JohnPaul}
$val = %data{-Lisa}
```

Accessing Hash Elements

When accessing individual elements from a hash, you must prefix the variable with a dollar sign ($) and then append the element key within curly brackets after the name of the variable. For example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
print "$data{'John Paul'}\n";
print "$data{'Lisa'}\n";
print "$data{'Kumar'}\n";
```

Extracting Slices

You can extract slices of a hash just as you can extract slices from an array. You will need to use @ prefix for the variable to store the returned value because they will be a list of values −

```
#!/uer/bin/perl
%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
@array = @data{-JohnPaul, -Lisa};
print "Array : @array\n";
```

Extracting Keys and Values

You can get a list of all of the keys from a hash by using **keys** function, which has the following syntax −

keys %HASH

This function returns an array of all the keys of the named hash. Following is the example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@names = keys %data;
print "$names[0]\n";
```

```
print "$names[1]\n";
print "$names[2]\n";
```

Similarly, you can use **values** function to get a list of all the values. This function has the following syntax −

values % HASH

This function returns a normal array consisting of all the values of the named hash. Following is the example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@ages = values %data;
print "$ages[0]\n";
print "$ages[1]\n";
print "$ages[2]\n";
```

Checking for Existence

If you try to access a key/value pair from a hash that doesn't exist, you'll normally get the **undefined** value, and if you have warnings switched on, then you'll get a warning generated at run time. You can get around this by using the **exists** function, which returns true if the named key exists, irrespective of what its value might be −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
if( exists($data{'Lisa'} ) ) {
   print "Lisa is $data{'Lisa'} years old\n";
} else {
   print "I don't know age of Lisa\n";
}
```

Here we have introduced the IF...ELSE statement, which we will study in a separate chapter. For now you just assume that **if(condition)** part will be executed only when the given condition is true otherwise **else** part will be executed. So when we execute the above program, it produces the following result because here the given condition *exists($data{'Lisa'}* returns true −

Lisa is 30 years old

Getting Hash Size

You can get the size - that is, the number of elements from a hash by using the scalar context on either keys or values. Simply saying first you have to get an array of either the keys or values and then you can get the size of array as follows −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@keys = keys %data;
$size = @keys;
print "1 - Hash size:  is $size\n";
@values = values %data;
$size = @values;
print "2 - Hash size:  is $size\n";
```

Add and Remove Elements in Hashes

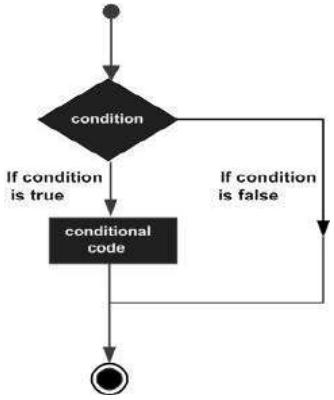
Adding a new key/value pair can be done with one line of code using simple assignment operator. But to remove an element from the hash you need to use **delete** function as shown below in the example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@keys = keys %data;
$size = @keys;
print "1 - Hash size:  is $size\n";
# adding an element to the hash;
$data{'Ali'} = 55;
@keys = keys %data;
$size = @keys;
print "2 - Hash size:  is $size\n";
# delete the same element from the hash;
```

```
delete $data{'Ali'};
@keys = keys %data;
$size = @keys;
print "3 - Hash size:  is $size\n";
```

Perl Conditional Statements - IF...ELSE

Perl conditional statements helps in the decision making, which require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.
Perl programming language provides the following types of conditional statements.

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**.
3	if...elsif...else statement An **if statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
4	unless statement An **unless statement** consists of a boolean expression followed by one or more statements.
5	unless...else statement An **unless statement** can be followed by an optional **else statement**.
6	unless...elsif..else statement An **unless statement** can be followed by an optional **elsif statement** and then by an optional **else statement**.
7	switch statement With the latest versions of Perl, you can make use of the **switch** statement. which allows a simple way of comparing a variable value against various conditions.

A Perl **if** statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an **if** statement in Perl programming language is −

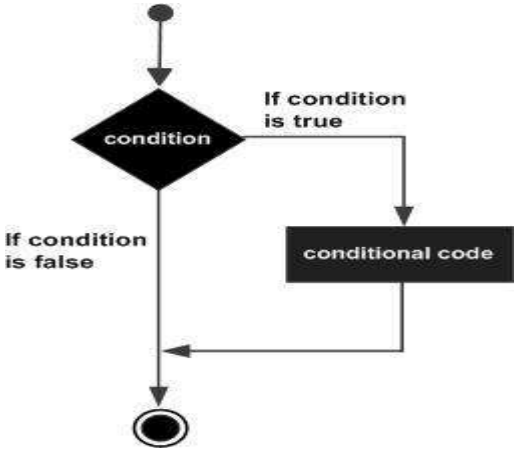
```
if(boolean_expression) {
```

```
  # statement(s) will execute if the given condition is true
}
```

If the boolean expression evaluates to **true** then the block of code inside the **if** statement will be executed. If boolean expression evaluates to **false** then the first set of code after the end of the **if** statement (after the closing curly brace) will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram



Example

```
#!/usr/local/bin/perl
 $a = 10;
# check the boolean condition using if statement
if( $a < 20 ) {
   # if condition is true then print the following
   printf "a is less than 20\n";
}
print "value of a is : $a\n";
$a = "";
# check the boolean condition using if statement
if( $a ) {
   # if condition is true then print the following
   printf "a has a true value\n";
}
print "value of a is : $a\n";
```

A Perl **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

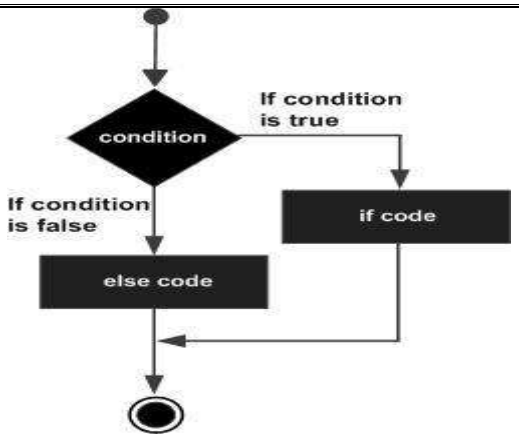
The syntax of an **if...else** statement in Perl programming language is −

```
if(boolean_expression) {
   # statement(s) will execute if the given condition is true
} else {
   # statement(s) will execute if the given condition is false
}
```

If the boolean expression evaluates to **true**, then the **if block** of code will be executed otherwise **else block** of code will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram




```
#!/usr/local/bin/perl
 $a = 100;
# check the boolean condition using if statement
if( $a < 20 ) {
   # if condition is true then print the following
   printf "a is less than 20\n";
} else {
   # if condition is false then print the following
   printf "a is greater than 20\n";
}
print "value of a is : $a\n";
$a = "";
# check the boolean condition using if statement
if( $a ) {
   # if condition is true then print the following
   printf "a has a true value\n";
} else {
   # if condition is false then print the following
   printf "a has a false value\n";
}
print "value of a is : $a\n";
```

An **if** statement can be followed by an optional **elsif...else** statement, which is very useful to test the various conditions using single if...elsif statement.

When using **if , elsif , else** statements there are few points to keep in mind.

- An **if** can have zero or one **else**'s and it must come after any **elsif**'s.
- An **if** can have zero to many **elsif**'s and they must come before the **else**.
- Once an **elsif** succeeds, none of the remaining **elsif**'s or **else**'s will be tested.

Syntax

The syntax of an **if...elsif...else** statement in Perl programming language is −

```
if(boolean_expression 1) {
   # Executes when the boolean expression 1 is true
} elsif( boolean_expression 2) {
   # Executes when the boolean expression 2 is true
} elsif( boolean_expression 3) {
   # Executes when the boolean expression 3 is true
} else {
   # Executes when the none of the above condition is true
}
```

Example

```
#!/usr/local/bin/perl
 $a = 100;
# check the boolean condition using if statement
if( $a  ==  20 ) {
   # if condition is true then print the following
   printf "a has a value which is 20\n";
} elsif( $a ==  30 ) {
   # if condition is true then print the following
```

```
  printf "a has a value which is 30\n";
} else {
  # if none of the above conditions is true
  printf "a has a value which is $a\n";
}
```

A Perl **unless** statement consists of a boolean expression followed by one or more statements.

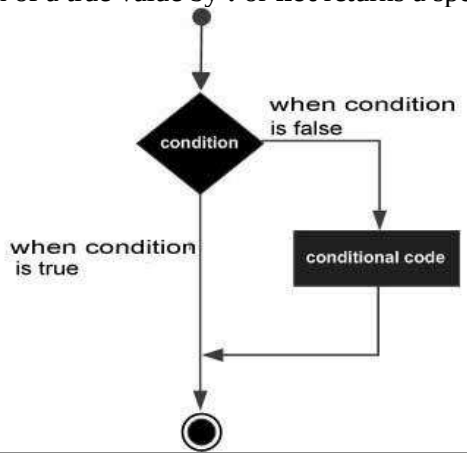
Syntax

The syntax of an unless statement in Perl programming language is −

```
unless(boolean_expression) {
  # statement(s) will execute if the given condition is false
}
```

If the boolean expression evaluates to **false**, then the block of code inside the unless statement will be executed. If boolean expression evaluates to **true** then the first set of code after the end of the unless statement (after the closing curly brace) will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.




```
#!/usr/local/bin/perl
 $a = 20;
# check the boolean condition using unless statement
unless( $a < 20 ) {
  # if condition is false then print the following
  printf "a is not less than 20\n";
}
print "value of a is : $a\n";
$a = "";
# check the boolean condition using unless statement
unless ( $a ) {
  # if condition is false then print the following
  printf "a has a false value\n";
}
print "value of a is : $a\n";
```

A Perl **unless** statement can be followed by an optional **else** statement, which executes when the boolean expression is true.

Syntax

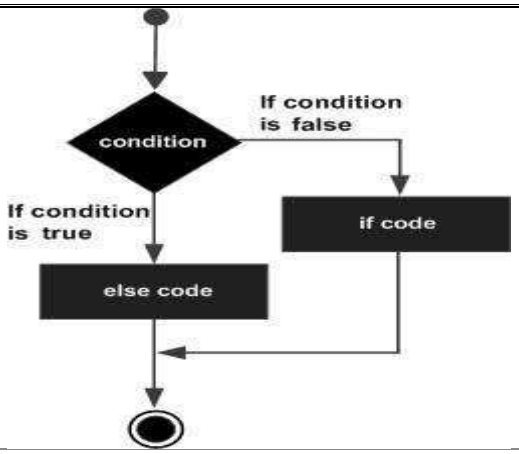
The syntax of an **unless...else** statement in Perl programming language is −

```
unless(boolean_expression) {
  # statement(s) will execute if the given condition is false
} else {
  # statement(s) will execute if the given condition is true
}
```

If the boolean expression evaluates to **true** then the **unless block** of code will be executed otherwise **else block** of code will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram




```
#!/usr/local/bin/perl
 $a = 100;
# check the boolean condition using unless statement
unless( $a == 20 ) {
   # if condition is false then print the following
   printf "given condition is false\n";
} else {
   # if condition is true then print the following
   printf "given condition is true\n";
}
print "value of a is : $a\n";
$a = "";
# check the boolean condition using unless statement
unless( $a ) {
   # if condition is false then print the following
   printf "a has a false value\n";
} else {
   # if condition is true then print the following
   printf "a has a true value\n";
}
print "value of a is : $a\n";
```

An **unless** statement can be followed by an optional **elsif...else** statement, which is very useful to test the various conditions using single unless...elsif statement.

When using unless, elsif, else statements there are few points to keep in mind.

- An **unless** can have zero or one **else**'s and it must come after any **elsif**'s.
- An **unless** can have zero to many **elsif**'s and they must come before the **else**.
- Once an **elsif** succeeds, none of the remaining **elsif**'s or **else**'s will be tested.

Syntax

The syntax of an **unless...elsif...else** statement in Perl programming language is −

```
unless(boolean_expression 1) {
   # Executes when the boolean expression 1 is false
} elsif( boolean_expression 2) {
   # Executes when the boolean expression 2 is true
} elsif( boolean_expression 3) {
   # Executes when the boolean expression 3 is true
} else {
   # Executes when the none of the above condition is met
}
```

```
#!/usr/local/bin/perl
 $a = 20;
# check the boolean condition using if statement
unless( $a  ==  30 ) {
   # if condition is false then print the following
   printf "a has a value which is not 20\n";
} elsif( $a ==  30 ) {
   # if condition is true then print the following
   printf "a has a value which is 30\n";
```

```
} else {
   # if none of the above conditions is met
   printf "a has a value which is $a\n";
}
```

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

A switch case implementation is dependent on **Switch** module and **Switch** module has been implemented using *Filter::Util::Call* and *Text::Balanced* and requires both these modules to be installed.

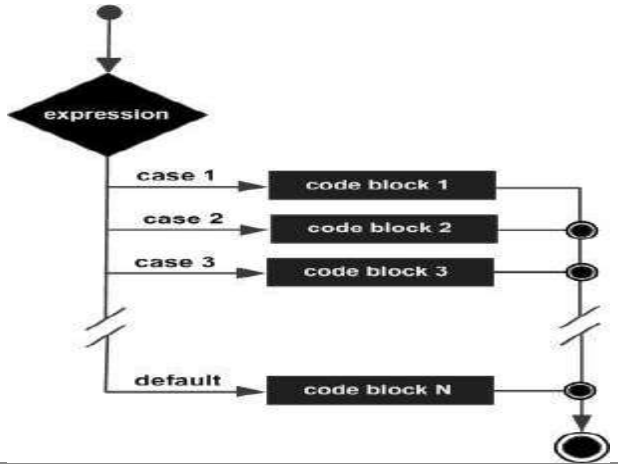
Syntax

The synopsis for a **switch** statement in Perl programming language is as follows −

```
use Switch;

switch(argument) {
   case 1            { print "number 1" }
   case "a"          { print "string a" }
   case [1..10,42]   { print "number in list" }
   case (\@array)    { print "number in list" }
   case /\w+/        { print "pattern" }
   case qr/\w+/      { print "pattern" }
   case (\%hash)     { print "entry in hash" }
   case (\&sub)      { print "arg to subroutine" }
   else              { print "previous case not true" }
}
```

The following rules apply to a **switch** statement −

- The **switch** statement takes a single scalar argument of any type, specified in parentheses.
- The value is followed by a block, which may contain one or more case statement followed by a block of Perl statement(s).
- A case statement takes a single scalar argument and selects the appropriate type of matching between the case argument and the current switch value.
- If the match is successful, the mandatory block associated with the case statement is executed.
- A **switch** statement can have an optional **else** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is matched.
- If a case block executes an untargeted **next**, control is immediately transferred to the statement after the case statement (i.e., usually another case), rather than out of the surrounding switch block.
- Not every case needs to contain a **next**. If no **next** appears, the flow of control will *not fall through* subsequent cases.




```
#!/usr/local/bin/perl
use Switch;
$var = 10;
@array = (10, 20, 30);
%hash = ('key1' => 10, 'key2' => 20);
switch($var) {
   case 10           { print "number 100\n" }
   case "a"          { print "string a" }
   case [1..10,42]   { print "number in list" }
```

```
  case (\@array)    { print "number in list" }
  case (\%hash)     { print "entry in hash" }
  else              { print "previous case not true" }
}
```

When the above code is executed, it produces the following result −

```
number 100
```

Fall-though is usually a bad idea in a switch statement. However, now consider a fall-through case, we will use the **next** to transfer the control to the next matching case, which is a list in this case −

```
#!/usr/local/bin/perl
use Switch;
$var = 10;
@array = (10, 20, 30);
%hash = ('key1' => 10, 'key2' => 20);
switch($var) {
  case 10           { print "number 100\n"; next; }
  case "a"          { print "string a" }
  case [1..10,42]   { print "number in list" }
  case (\@array)    { print "number in list" }
  case (\%hash)     { print "entry in hash" }
  else              { print "previous case not true" }
}
```

The ? : Operator

Let's check the **conditional operator ? :**which can be used to replace **if...else** statements. It has the following general form −

```
Exp1 ? Exp2 : Exp3;
```

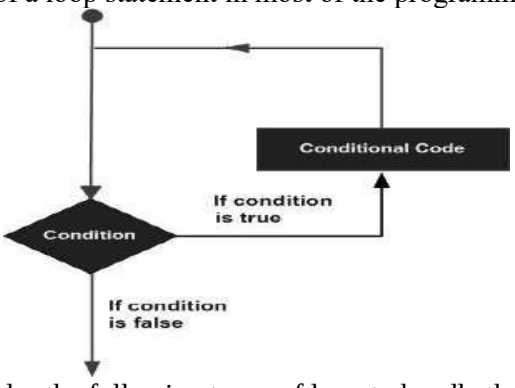
Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression. Below is a simple example making use of this operator −

```
#!/usr/local/bin/perl
 $name = "Ali";
$age = 10;
$status = ($age > 60 )? "A senior citizen" : "Not a senior citizen";
print "$name is  - $status\n";
```

Perl – Loops

- There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



Perl programming language provides the following types of loop to handle the looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

2	until loop Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.
3	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	foreach loop The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.
5	do...while loop Like a while statement, except that it tests the condition at the end of the loop body
6	nested loops You can use one or more loop inside any another while, for or do..while loop.

A **while** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

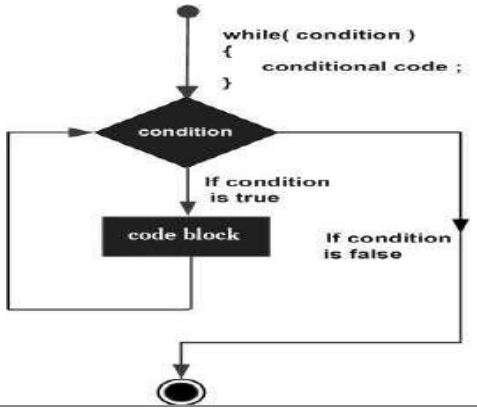
The syntax of a **while** loop in Perl programming language is −

```
while(condition)
{
  statement(s);
}
```

Here **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram




```
#!/usr/local/bin/perl
 $a = 10;
# while loop execution
while( $a < 20 ) {
   printf "Value of a: $a\n";
   $a = $a + 1;
}
```

An **until** loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is false.

Syntax

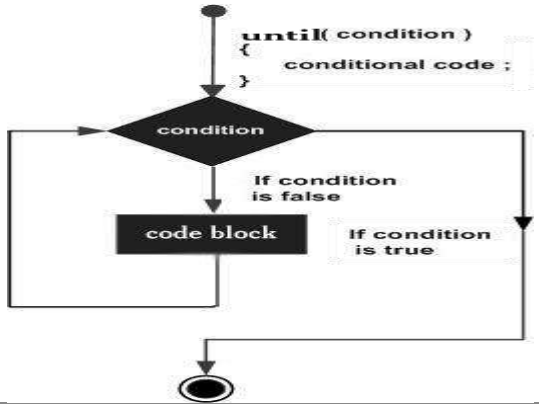
The syntax of an **until** loop in Perl programming language is −

```
until(condition) {
  statement(s);
}
```

Here **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression. The loop iterates until the condition becomes true. When the condition becomes true, the program control passes to the line immediately following the loop.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram




```
#!/usr/local/bin/perl
 $a = 5;
# until loop execution
until( $a > 10 ) {
   printf "Value of a: $a\n";
   $a = $a + 1;
}
```

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

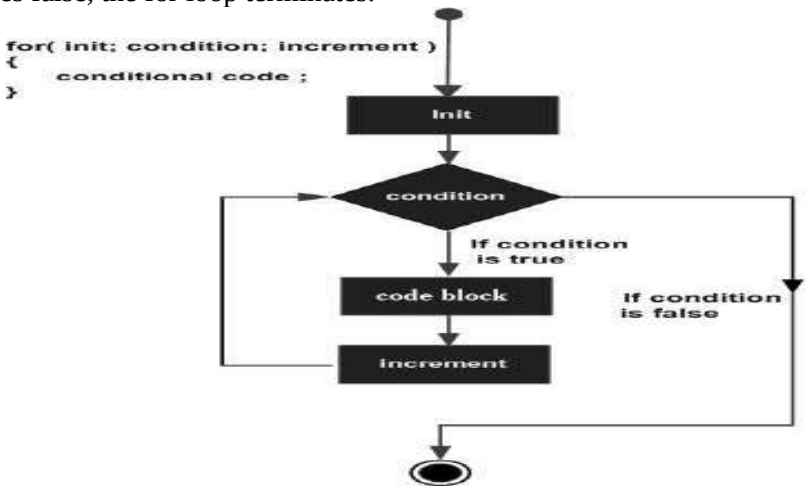
Syntax

The syntax of a **for** loop in Perl programming language is −

```
for ( init; condition; increment )
 {
   statement(s);
}
```

Here is the flow of control in a **for** loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.



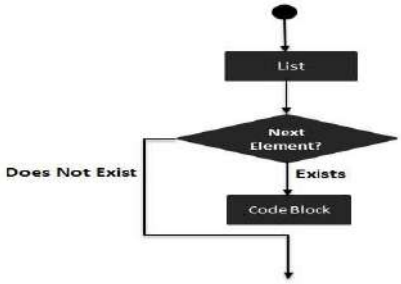
```
#!/usr/local/bin/perl
 # for loop execution
for( $a = 10; $a < 20; $a = $a + 1 ) {
   print "value of a: $a\n";
}
```

The **foreach** loop iterates over a list value and sets the control variable (var) to be each element of the list in turn −

Syntax

The syntax of a **foreach** loop in Perl programming language is −

```
foreach var (list) {
...
}
```




```
#!/usr/local/bin/perl
 @list = (2, 20, 30, 40, 50);
# foreach loop execution
foreach $a (@list) {
   print "value of a: $a\n";
}
```

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

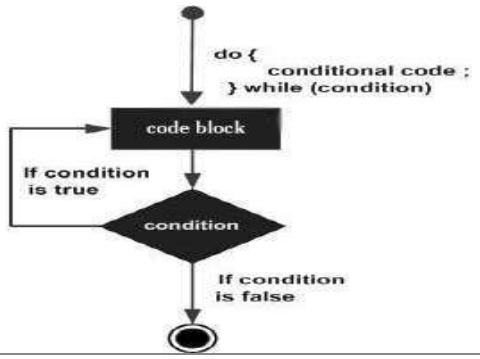
The syntax of a **do...while** loop in Perl is −

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

It should be noted that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

Flow Diagram




```
#!/usr/local/bin/perl
 $a = 10;
# do..while loop execution
do{
   printf "Value of a: $a\n";
   $a = $a + 1;
```

```
}while( $a < 20 );
```

A loop can be nested inside of another loop. Perl allows to nest all type of loops to be nested.

Syntax

The syntax for **a nested for loop** statement in Perl is as follows −

```
for ( init; condition; increment ) {
   for ( init; condition; increment ) {
      statement(s);
   }
   statement(s);
}
```

The syntax for a **nested while loop** statement in Perl is as follows −

```
while(condition) {
   while(condition) {
      statement(s);
   }
   statement(s);
}
```

The syntax for a **nested do...while loop** statement in Perl is as follows −

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

}while( condition );
```

The syntax for a **nested until loop** statement in Perl is as follows −

```
until(condition) {
   until(condition) {
      statement(s);
   }
   statement(s);
}
```

The syntax for a **nested foreach loop** statement in Perl is as follows −

```
foreach $a (@listA) {
   foreach $b (@listB) {
      statement(s);
   }
   statement(s);
}
```

```
#/usr/local/bin/perl
   $a = 0;
$b = 0;
# outer while loop
while($a < 3) {
   $b = 0;
   # inner while loop
   while( $b < 3 ) {
      print "value of a = $a, b = $b\n";
      $b = $b + 1;
   }
   $a = $a + 1;
   print "Value of a = $a\n\n";
}
```

Loop Control Statements

Loop control statements change the execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Perl supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description
1	next statement

	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
2	last statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
3	continue statement A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.
4	redo statement The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed.
5	goto statement Perl supports a goto command with three forms: goto label, goto expr, and goto &name.

The Perl **next** statement starts the next iteration of the loop. You can provide a LABEL with **next** statement where LABEL is the label for a loop. A **next** statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

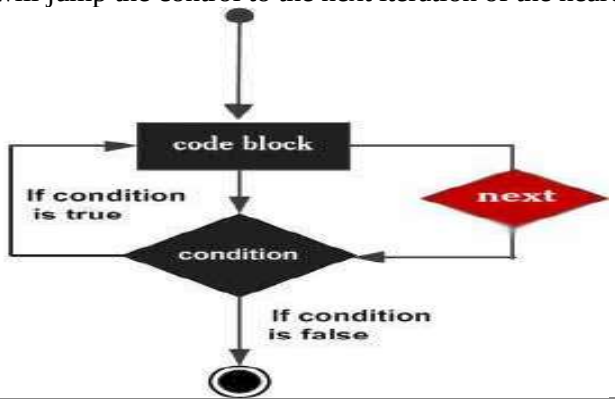
If there is a **continue** block on the loop, it is always executed just before the condition is about to be evaluated. You will see the continue statement in separate chapter.

Syntax

The syntax of a **next** statement in Perl is −

```
next [ LABEL ];
```

A LABEL inside the square braces indicates that LABEL is optional and if a LABEL is not specified, then next statement will jump the control to the next iteration of the nearest loop.




```
#!/usr/local/bin/perl
$a = 10;
while( $a < 20 )
 {
   if( $a == 15)
   {
      # skip the iteration.
      $a = $a + 1;
      next;
   }
   print "value of a: $a\n";
   $a = $a + 1;
}
```

Let's take one example where we are going to use a LABEL along with next statement −

```
#!/usr/local/bin/perl

$a = 0;
OUTER: while( $a < 4 ) {
   $b = 0;
   print "value of a: $a\n";
   INNER:while ( $b < 4) {
      if( $a == 2) {
```

```
      $a = $a + 1;
      # jump to outer loop
      next OUTER;
    }
    $b = $b + 1;
    print "Value of b : $b\n";
  }
  print "\n";
  $a = $a + 1;
}
```

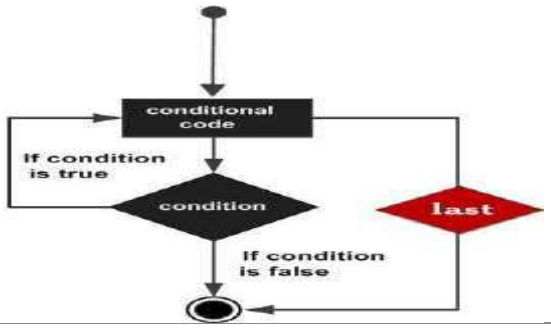
When a **last** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. You can provide a LABEL with last statement where LABEL is the label for a loop. A **last** statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

If there is any **continue** block on the loop, then it is not executed. You will see the continue statement in a separate chapter.

Syntax

The syntax of a **last** statement in Perl is −

```
last [LABEL];
```




```
#!/usr/local/bin/perl
$a = 10;
while( $a < 20 )
 {
  if( $a == 15) {
    # terminate the loop.
    $a = $a + 1;
    last;
  }
  print "value of a: $a\n";
  $a = $a + 1;
}
```

Let's take one example where we are going to use a LABEL along with next statement −

```
#!/usr/local/bin/perl
$a = 0;
OUTER: while( $a < 4 ) {
  $b = 0;
  print "value of a: $a\n";
  INNER:while ( $b < 4) {
    if( $a == 2) {
      # terminate outer loop
      last OUTER;
    }
    $b = $b + 1;
    print "Value of b : $b\n";
  }
  print "\n";
  $a = $a + 1;
}
```

A **continue** BLOCK, is always executed just before the conditional is about to be evaluated again. A continue statement can be used with *while* and *foreach* loops. A continue statement can also be used

alone along with a BLOCK of code in which case it will be assumed as a flow control statement rather than a function.

Syntax

The syntax for a **continue** statement with **while** loop is as follows −

```
while(condition) {
   statement(s);
} continue {
   statement(s);
}
```

The syntax for a **continue** statement with **foreach** loop is as follows −

```
foreach $a (@listA) {
   statement(s);
} continue {
   statement(s);
}
```

The syntax for a **continue** statement with a BLOCK of code is as follows −

```
continue {
   statement(s);
}
```

```
#/usr/local/bin/perl

$a = 0;
while($a < 3) {
   print "Value of a = $a\n";
} continue {
   $a = $a + 1;
}
```

The following program shows the usage of **continue** statement with **foreach** loop −

```
#/usr/local/bin/perl

@list = (1, 2, 3, 4, 5);
foreach $a (@list) {
   print "Value of a = $a\n";
} continue {
   last if $a == 4;
}
```

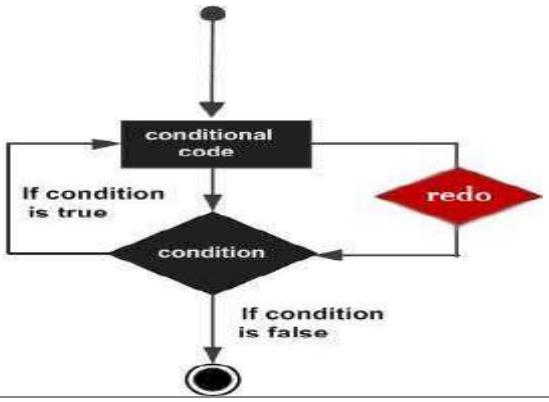
The **redo** command restarts the loop block without evaluating the conditional again. You can provide a LABEL with **redo** statement where LABEL is the label for a loop. A **redo** statement can be used inside a nested loop where it will be applicable to the nearest loop if a LABEL is not specified.

If there is any **continue** block on the loop, then it will not be executed before evaluating the condition.

Syntax

The syntax for a **redo** statement is as follows −

```
redo [LABEL]
```




```
#/usr/local/bin/perl
   $a = 0;
while($a < 10) {
   if( $a == 5 ) {
      $a = $a + 1;
      redo;
   }
```

```
   print "Value of a = $a\n";
} continue {
   $a = $a + 1;
}
```

Perl does support a **goto** statement. There are three forms: goto LABEL, goto EXPR, and goto &NAME.

Sr.No.	goto type
1	**goto LABEL** The goto LABEL form jumps to the statement labeled with LABEL and resumes execution from there.
2	**goto EXPR** The goto EXPR form is just a generalization of goto LABEL. It expects the expression to return a label name and then jumps to that labeled statement.
3	**goto &NAME** It substitutes a call to the named subroutine for the currently running subroutine.

Syntax

The syntax for a **goto** statements is as follows −

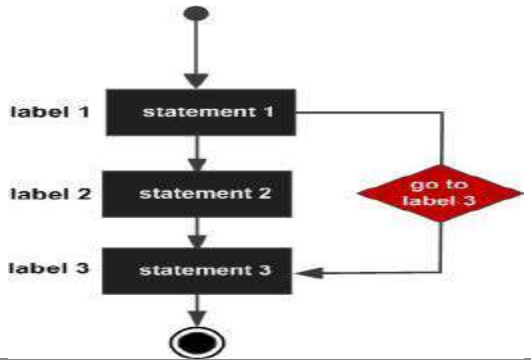
goto LABEL

or

goto EXPR

or

goto &NAME




```
#/usr/local/bin/perl
   $a = 10;
LOOP:do {
   if( $a == 15) {
      # skip the iteration.
      $a = $a + 1;
      # use goto LABEL form
      goto LOOP;
   }
   print "Value of a = $a\n";
   $a = $a + 1;
} while( $a < 20 );
```

Following example shows the usage of goto EXPR form. Here we are using two strings and then concatenating them using string concatenation operator (.). Finally, its forming a label and goto is being used to jump to the label −

```
#/usr/local/bin/perl
   $a = 10;
$str1 = "LO";
$str2 = "OP";
LOOP:do {
   if( $a == 15) {
```

```
      # skip the iteration.
      $a = $a + 1;
      # use goto EXPR form
      goto $str1.$str2;
   }
   print "Value of a = $a\n";
   $a = $a + 1;
} while( $a < 20 );
```

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#!/usr/local/bin/perl

for( ; ; ) {
   printf "This loop will run forever.\n";
}
```

You can terminate the above infinite loop by pressing the Ctrl + C keys.

Perl - Operators

What is an Operator?

Perl language supports many operator types, but following is a list of important and most frequently used operators −

- Arithmetic Operators
- Equality Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Quote-like Operators
- Miscellaneous Operators

Perl Arithmetic Operators

Assume variable $a holds 10 and variable $b holds 20, then following are the Perl arithmetic operators

Sr.No.	Operator & Description
1	**+ (Addition)** Adds values on either side of the operator **Example** − $a + $b will give 30
2	**- (Subtraction)** Subtracts right hand operand from left hand operand **Example** − $a - $b will give -10
3	*** (Multiplication)** Multiplies values on either side of the operator **Example** − $a * $b will give 200
4	**/ (Division)** Divides left hand operand by right hand operand **Example** − $b / $a will give 2
5	**% (Modulus)** Divides left hand operand by right hand operand and returns remainder **Example** − $b % $a will give 0
6	**** (Exponent)** Performs exponential (power) calculation on operators **Example** − $a**$b will give 10 to the power 20

```
#!/usr/local/bin/perl
 $a = 21;
$b = 10;
print "Value of \$a = $a and value of \$b = $b\n";
$c = $a + $b;
print 'Value of $a + $b = ' . $c . "\n";
$c = $a - $b;
print 'Value of $a - $b = ' . $c . "\n";
$c = $a * $b;
print 'Value of $a * $b = ' . $c . "\n";
$c = $a / $b;
print 'Value of $a / $b = ' . $c . "\n";
$c = $a % $b;
print 'Value of $a % $b = ' . $c. "\n";
$a = 2;
$b = 4;
$c = $a ** $b;
print 'Value of $a ** $b = ' . $c . "\n";
```

Perl Equality Operators

These are also called relational operators. Assume variable $a holds 10 and variable $b holds 20 then,

Sr.No.	Operator & Description
1	**== (equal to)** Checks if the value of two operands are equal or not, if yes then condition becomes true. **Example** − ($a == $b) is not true.
2	**!= (not equal to)** Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. **Example** − ($a != $b) is true.
3	**<=>** Checks if the value of two operands are equal or not, and returns -1, 0, or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument. **Example** − ($a <=> $b) returns -1.
4	**> (greater than)** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. **Example** − ($a > $b) is not true.
5	**< (less than)** Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example** − ($a < $b) is true.
6	**>= (greater than or equal to)** Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. **Example** − ($a >= $b) is not true.
7	**<= (less than or equal to)** Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. **Example** − ($a <= $b) is true.

```
#!/usr/local/bin/perl
 $a = 21;
$b = 10;
print "Value of \$a = $a and value of \$b = $b\n";
if( $a == $b ) {
```

```
   print "$a == \$b is true\n";
} else {
   print "\$a == \$b is not true\n";
}
if( $a != $b ) {
   print "\$a != \$b is true\n";
} else {
   print "\$a != \$b is not true\n";
}
$c = $a <=> $b;
print "\$a <=> \$b returns $c\n";
if( $a > $b ) {
   print "\$a > \$b is true\n";
} else {
   print "\$a > \$b is not true\n";
}
if( $a >= $b ) {
   print "\$a >= \$b is true\n";
} else {
   print "\$a >= \$b is not true\n";
}
if( $a < $b ) {
   print "\$a < \$b is true\n";
} else {
   print "\$a < \$b is not true\n";
}
if( $a <= $b ) {
   print "\$a <= \$b is true\n";
} else {
   print "\$a <= \$b is not true\n"; }
```

Below is a list of equity operators. Assume variable $a holds "abc" and variable $b holds "xyz" then,

Sr.No.	Operator & Description
1	**lt** Returns true if the left argument is stringwise less than the right argument. **Example** − ($a lt $b) is true.
2	**gt** Returns true if the left argument is stringwise greater than the right argument. **Example** − ($a gt $b) is false.
3	**le** Returns true if the left argument is stringwise less than or equal to the right argument. **Example** − ($a le $b) is true.
4	**ge** Returns true if the left argument is stringwise greater than or equal to the right argument. **Example** − ($a ge $b) is false.
5	**eq** Returns true if the left argument is stringwise equal to the right argument. **Example** − ($a eq $b) is false.
6	**ne** Returns true if the left argument is stringwise not equal to the right argument. **Example** − ($a ne $b) is true.

7	**cmp** Returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument. **Example** − ($a cmp $b) is -1.

```
#!/usr/local/bin/perl
 $a = "abc";
$b = "xyz";
print "Value of \$a = $a and value of \$b = $b\n";
if( $a lt $b ) {
   print "$a lt \$b is true\n";
} else {
   print "\$a lt \$b is not true\n";
}
if( $a gt $b ) {
   print "\$a gt \$b is true\n";
} else {
   print "\$a gt \$b is not true\n";
}
if( $a le $b ) {
   print "\$a le \$b is true\n";
} else {
   print "\$a le \$b is not true\n";
}
if( $a ge $b ) {
   print "\$a ge \$b is true\n";
} else {
   print "\$a ge \$b is not true\n";
}
if( $a ne $b ) {
   print "\$a ne \$b is true\n";
} else {
   print "\$a ne \$b is not true\n";
}
$c = $a cmp $b;
print "\$a cmp \$b returns $c\n";
```

Perl Assignment Operators

Assume variable $a holds 10 and variable $b holds 20, then below are the assignment operators available in Perl and their usage −

Sr.No.	Operator & Description
1	**=** Simple assignment operator, Assigns values from right side operands to left side operand **Example** − $c = $a + $b will assigned value of $a + $b into $c
2	**+=** Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand **Example** − $c += $a is equivalent to $c = $c + $a
3	**-=** Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand **Example** − $c -= $a is equivalent to $c = $c - $a
4	***=** Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand **Example** − $c *= $a is equivalent to $c = $c * $a

5	**/=** Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand **Example** − $c /= $a is equivalent to $c = $c / $a
6	**%=** Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand **Example** − $c %= $a is equivalent to $c = $c % a
7	****=** Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand **Example** − $c **= $a is equivalent to $c = $c ** $a

```
#!/usr/local/bin/perl
 $a = 10;
$b = 20;
print "Value of \$a = $a and value of \$b = $b\n";
$c = $a + $b;
print "After assignment value of \$c = $c\n";
$c += $a;
print "Value of \$c = $c after statement \$c += \$a\n";
$c -= $a;
print "Value of \$c = $c after statement \$c -= \$a\n";
$c *= $a;
print "Value of \$c = $c after statement \$c *= \$a\n";
$c /= $a;
print "Value of \$c = $c after statement \$c /= \$a\n";
$c %= $a;
print "Value of \$c = $c after statement \$c %= \$a\n";
$c = 2;
$a = 4;
print "Value of \$a = $a and value of \$c = $c\n";
$c **= $a;
print "Value of \$c = $c after statement \$c **= \$a\n";
```

Perl Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. Assume if $a = 60; and $b = 13; Now in binary format they will be as follows −

$a = 0011 1100
$b = 0000 1101

$a&$b = 0000 1100
$a|$b = 0011 1101
$a^$b = 0011 0001
~$a = 1100 0011

There are following Bitwise operators supported by Perl language, assume if $a = 60; and $b = 13

Sr.No.	Operator & Description
1	**&** Binary AND Operator copies a bit to the result if it exists in both operands. **Example** − ($a & $b) will give 12 which is 0000 1100
2	**|** Binary OR Operator copies a bit if it exists in eather operand. **Example** − ($a | $b) will give 61 which is 0011 1101
3	**^** Binary XOR Operator copies the bit if it is set in one operand but not both. **Example** − ($a ^ $b) will give 49 which is 0011 0001

4	~ Binary Ones Complement Operator is unary and has the efect of 'flipping' bits. **Example** − (~$a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
5	<< Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. **Example** − $a << 2 will give 240 which is 1111 0000
6	>> Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. **Example** − $a >> 2 will give 15 which is 0000 1111

```
#!/usr/local/bin/perl
use integer;
 $a = 60;
$b = 13;
print "Value of \$a = $a and value of \$b = $b\n";
$c = $a & $b;
print "Value of \$a & \$b = $c\n";
$c = $a | $b;
print "Value of \$a | \$b = $c\n";
$c = $a ^ $b;
print "Value of \$a ^ \$b = $c\n";
$c = ~$a;
print "Value of ~\$a = $c\n";
$c = $a << 2;
print "Value of \$a << 2 = $c\n";
$c = $a >> 2;
print "Value of \$a >> 2 = $c\n";
```

Perl Logical Operators

There are following logical operators supported by Perl language. Assume variable $a holds true and variable $b holds false then −

Sr.No.	Operator & Description
1	**and** Called Logical AND operator. If both the operands are true then then condition becomes true. **Example** − ($a and $b) is false.
2	**&&** C-style Logical AND operator copies a bit to the result if it exists in both operands. **Example** − ($a && $b) is false.
3	**or** Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true. **Example** − ($a or $b) is true.
4	|| C-style Logical OR operator copies a bit if it exists in eather operand. **Example** − ($a || $b) is true.
5	**not** Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. **Example** − not($a and $b) is true.

```
#!/usr/local/bin/perl
 $a = true;
$b = false;
print "Value of \$a = $a and value of \$b = $b\n";
$c = ($a and $b);
print "Value of \$a and \$b = $c\n";
$c = ($a  && $b);
print "Value of \$a && \$b = $c\n";
$c = ($a or $b);
print "Value of \$a or \$b = $c\n";
$c = ($a || $b);
print "Value of \$a || \$b = $c\n";
$a = 0;
$c = not($a);
print "Value of not(\$a)= $c\n";
```

Quote-like Operators

There are following Quote-like operators supported by Perl language. In the following table, a {} represents any pair of delimiters you choose.

Sr.No.	Operator & Description
1	**q{ }** Encloses a string with-in single quotes **Example** − q{abcd} gives 'abcd'
2	**qq{ }** Encloses a string with-in double quotes **Example** − qq{abcd} gives "abcd"
3	**qx{ }** Encloses a string with-in invert quotes **Example** − qx{abcd} gives `abcd`

```
#!/usr/local/bin/perl
$a = 10;
 $b = q{a = $a};
print "Value of q{a = \$a} = $b\n";
$b = qq{a = $a};
print "Value of qq{a = \$a} = $b\n";
# unix command execution
$t = qx{date};
print "Value of qx{date} = $t\n";
```

Miscellaneous Operators

There are following miscellaneous operators supported by Perl language. Assume variable a holds 10 and variable b holds 20 then −

Sr.No.	Operator & Description
1	**.** Binary operator dot (.) concatenates two strings. **Example** − If $a = "abc", $b = "def" then $a.$b will give "abcdef"
2	**x** The repetition operator x returns a string consisting of the left operand repeated the number of times specified by the right operand. **Example** − ('-' x 3) will give ---.
3	**..** The range operator .. returns a list of values counting (up by ones) from the left value to the right value **Example** − (2..5) will give (2, 3, 4, 5)

4	++ Auto Increment operator increases integer value by one **Example** − $a++ will give 11
5	-- Auto Decrement operator decreases integer value by one **Example** − $a-- will give 9
6	->The arrow operator is mostly used in dereferencing a method or variable from an object or a class name **Example** − $obj->$a is an example to access variable $a from object $obj.

```
#!/usr/local/bin/perl
$a = "abc";
$b = "def";
print "Value of \$a  = $a and value of \$b = $b\n";
 $c = $a . $b;
print "Value of \$a . \$b = $c\n";
$c = "-" x 3;
print "Value of \"-\" x 3 = $c\n";
@c = (2..5);
print "Value of (2..5) = @c\n";
$a = 10;
$b = 15;
print "Value of \$a  = $a and value of \$b = $b\n";
$a++;
$c = $a ;
print "Value of \$a after \$a++ = $c\n";

$b--;
$c = $b ;
print "Value of \$b after \$b-- = $c\n";
```

Perl Operators Precedence

The following table lists all operators from highest precedence to lowest.

left	terms and list operators (leftward)
left	->
nonassoc	++ --
right	**
right	! ~ \ and unary + and -
left	=~ !~
left	* / % x
left	+ - .
left	<< >>
nonassoc	named unary operators
nonassoc	< > <= >= lt gt le ge
nonassoc	== != <=> eq ne cmp ~~
left	&
left	| ^
left	&&
left	|| //
nonassoc	.. ...
right	?:
right	= += -= *= etc.
left	, =>
nonassoc	list operators (rightward)
right	not
left	and
left	or xor

```
#!/usr/local/bin/perl
```

```
$a = 20;
$b = 10;
$c = 15;
$d = 5;
$e;
print "Value of \$a  = $a, \$b = $b, \$c = $c and \$d = $d\n";
 $e = ($a + $b) * $c / $d;
print "Value of (\$a + \$b) * \$c / \$d is  = $e\n";
$e = (($a + $b) * $c )/ $d;
print "Value of ((\$a + \$b) * \$c) / \$d is  = $e\n";
$e = ($a + $b) * ($c / $d);
print "Value of (\$a + \$b) * (\$c / \$d ) is  = $e\n";
$e = $a + ($b * $c ) / $d;
print "Value of \$a + (\$b * \$c )/ \$d is  = $e\n";
```

Perl – Subroutines

- A Perl subroutine or function is a group of statements that together performs a task.
- You can divide up your code into separate subroutines.
- How you divide up your code among different subroutines is up to you, but logically the division usually is so each function performs a specific task.

Perl uses the terms subroutine, method and function interchangeably.

Define and Call a Subroutine

The general form of a subroutine definition in Perl programming language is as follows −

```
sub subroutine_name {
   body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows −

```
subroutine_name( list of arguments );
```

In versions of Perl before 5.0, the syntax for calling subroutines was slightly different as shown below. This still works in the newest versions of Perl, but it is not recommended since it bypasses the subroutine prototypes.

```
&subroutine_name( list of arguments );
```

```
#!/usr/bin/perl
# Function definition
sub Hello {
   print "Hello, World!\n";
}
# Function call
Hello();
```

Passing Arguments to a Subroutine

You can pass various arguments to a subroutine like you do in any other programming language and they can be accessed inside the function using the special array @_. Thus the first argument to the function is in $_[0], the second is in $_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next chapter) to pass any array or hash.

```
#!/usr/bin/perl
# Function definition
sub Average {
   # get total number of arguments passed.
   $n = scalar(@_);
   $sum = 0;
   foreach $item (@_) {
      $sum += $item;
   }
   $average = $sum / $n;
   print "Average for the given numbers : $average\n";
}
```

```
# Function call
Average(10, 20, 30);
```

Passing Lists to Subroutines

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_.

```
#!/usr/bin/perl
# Function definition
sub PrintList {
   my @list = @_;
   print "Given list is @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);
# Function call with list parameter
PrintList($a, @b);
```

Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs. For example −

```
#!/usr/bin/perl
# Function definition
sub PrintHash {
   my (%hash) = @_;
   foreach my $key ( keys %hash ) {
      my $value = $hash{$key};
      print "$key : $value\n";
   }
}
%hash = ('name' => 'Tom', 'age' => 19);
# Function call with hash parameter
PrintHash(%hash);
```

Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

```
#!/usr/bin/perl
# Function definition
sub Average {
   # get total number of arguments passed.
   $n = scalar(@_);
   $sum = 0;
   foreach $item (@_) {
      $sum += $item;
   }
   $average = $sum / $n;
   return $average;
}
# Function call
$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

Private Variables in a Subroutine

By default, all variables in Perl are global variables, which means they can be accessed from anywhere in the program. But you can create **private** variables called **lexical variables** at any time with the **my** operator.

The **my** operator confines a variable to a particular region of code in which it can be used and accessed. Outside that region, this variable cannot be used or accessed. This region is called its scope. A lexical scope is usually a block of code with a set of braces around it, such as those defining the body of the subroutine or those marking the code blocks of *if*, *while*, *for*, *foreach*, and *eval* statements.

Following is an example showing you how to define a single or multiple private variables using **my** operator −

```
sub somefunc {
  my $variable; # $variable is invisible outside somefunc()
  my ($another, @an_array, %a_hash); # declaring many variables at once
}
```

Let's check the following example to distinguish between global and private variables −

```
#!/usr/bin/perl
# Global variable
$string = "Hello, World!";
# Function definition
sub PrintHello {
  # Private variable for PrintHello function
  my $string;
  $string = "Hello, Perl!";
  print "Inside the function $string\n";
}
# Function call
PrintHello();
print "Outside the function $string\n";
```

Temporary Values via local()

The **local** is mostly used when the current value of a variable must be visible to called subroutines. A local just gives temporary values to global (meaning package) variables. This is known as *dynamic scoping*. Lexical scoping is done with my, which works more like C's auto declarations.

If more than one variable or expression is given to local, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval.

```
#!/usr/bin/perl
# Global variable
$string = "Hello, World!";
sub PrintHello {
  # Private variable for PrintHello function
  local $string;
  $string = "Hello, Perl!";
  PrintMe();
  print "Inside the function PrintHello $string\n";
}
sub PrintMe {
  print "Inside the function PrintMe $string\n";
}
# Function call
PrintHello();
print "Outside the function $string\n";
```

State Variables via state()

There are another type of lexical variables, which are similar to private variables but they maintain their state and they do not get reinitialized upon multiple calls of the subroutines. These variables are defined using the **state** operator and available starting from Perl 5.9.4.

```
#!/usr/bin/perl
use feature 'state';
sub PrintCount {
  state $count = 0; # initial value
  print "Value of counter is $count\n";
  $count++;
}
```

```
for (1..5) {
  PrintCount();
}
```

Prior to Perl 5.10, you would have to write it like this −

```
#!/usr/bin/perl
{
  my $count = 0; # initial value
  sub PrintCount {
    print "Value of counter is $count\n";
    $count++;
  }
}
for (1..5) {
  PrintCount();
}
```

Subroutine Call Context

The context of a subroutine or statement is defined as the type of return value that is expected. This allows you to use a single function that returns different values based on what the user is expecting to receive. For example, the following localtime() returns a string when it is called in scalar context, but it returns a list when it is called in list context.

```
my $datestring = localtime( time );
```

In this example, the value of $timestr is now a string made up of the current date and time, for example, Thu Nov 30 15:21:33 2000. Conversely −

```
($sec,$min,$hour,$mday,$mon, $year,$wday,$yday,$isdst) = localtime(time);
```

Now the individual variables contain the corresponding values returned by localtime() subroutine.

Perl – References

A Perl reference is a scalar data type that holds the location of another value which could be scalar, arrays, or hashes. Because of its scalar nature, a reference can be used anywhere, a scalar can be used. You can construct lists containing references to other lists, which can contain references to hashes, and so on. This is how the nested data structures are built in Perl.

Create References

It is easy to create a reference for any variable, subroutine or value by prefixing it with a backslash as follows −

```
$scalarref = \$foo;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*foo;
```

You cannot create a reference on an I/O handle (filehandle or dirhandle) using the backslash operator but a reference to an anonymous array can be created using the square brackets as follows −

```
$arrayref = [1, 2, ['a', 'b', 'c']];
```

Similar way you can create a reference to an anonymous hash using the curly brackets as follows −

```
$hashref = {
  'Adam'  => 'Eve',
  'Clyde' => 'Bonnie',
};
```

A reference to an anonymous subroutine can be created by using sub without a subname as follows −

```
$coderef = sub { print "Boink!\n" };
```

Dereferencing

Dereferencing returns the value from a reference point to the location. To dereference a reference simply use $, @ or % as prefix of the reference variable depending on whether the reference is pointing to a scalar, array, or hash. Following is the example to explain the concept −

```
#!/usr/bin/perl
$var = 10;
```

```
# Now $r has reference to $var scalar.
$r = \$var;
# Print value available at the location stored in $r.
print "Value of $var is : ", $$r, "\n";
@var = (1, 2, 3);
# Now $r has reference to @var array.
$r = \@var;
# Print values available at the location stored in $r.
print "Value of @var is : ",  @$r, "\n";
%var = ('key1' => 10, 'key2' => 20);
# Now $r has reference to %var hash.
$r = \%var;
# Print values available at the location stored in $r.
print "Value of %var is : ", %$r, "\n";
```

If you are not sure about a variable type, then its easy to know its type using **ref**, which returns one of the following strings if its argument is a reference. Otherwise, it returns false −

SCALAR
ARRAY
HASH
CODE
GLOB
REF

```
#!/usr/bin/perl
$var = 10;
$r = \$var;
print "Reference type in r : ", ref($r), "\n";
@var = (1, 2, 3);
$r = \@var;
print "Reference type in r : ", ref($r), "\n";
%var = ('key1' => 10, 'key2' => 20);
$r = \%var;
print "Reference type in r : ", ref($r), "\n";
```

Circular References

A circular reference occurs when two references contain a reference to each other. You have to be careful while creating references otherwise a circular reference can lead to memory leaks. Following is an example −

```
#!/usr/bin/perl
 my $foo = 100;
 $foo = \$foo;
  print "Value of foo is : ", $$foo, "\n";
```

References to Functions

This might happen if you need to create a signal handler so you can produce a reference to a function by preceding that function name with \& and to dereference that reference you simply need to prefix reference variable using ampersand &. Following is an example −

```
#!/usr/bin/perl
# Function definition
sub PrintHash {
   my (%hash) = @_;
      foreach $item (%hash) {
      print "Item : $item\n";
   }
}
%hash = ('name' => 'Tom', 'age' => 19);
# Create a reference to above function.
$cref = \&PrintHash;
# Function call using reference.
&$cref(%hash);
```

Regular Expressions

The basic method for applying a regular expression is to use the pattern binding operators =~ and !~. The first operator is a test and assignment operator.

There are three regular expression operators within Perl.

Match Regular Expression - m//
Substitute Regular Expression - s///
Translate Regular Expression - tr///

The forward slashes in each case act as delimiters for the regular expression (regex) that you are specifying. If you are comfortable with any other delimiter, then you can use in place of forward slash.

The Match Operator

The match operator, m//, is used to match a string or statement to a regular expression. For example, to match the character sequence "foo" against the scalar $bar, you might use a statement like this −

```
#!/usr/bin/perl
$bar = "This is foo and again foo";
if ($bar =~ /foo/) {
  print "First time is matching\n";
} else {
  print "First time is not matching\n";
}
$bar = "foo";
if ($bar =~ /foo/) {
  print "Second time is matching\n";
} else {
  print "Second time is not matching\n";
}
```

The match operator supports its own set of modifiers. The /g modifier allows for global matching. The /i modifier will make the match case insensitive. Here is the complete list of modifiers

Sr.No.	Modifier & Description
1	**i** Makes the match case insensitive.
2	**m** Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary.
3	**o** Evaluates the expression only once.
4	**s** Allows use of . to match a newline character.
5	**x** Allows you to use white space in the expression for clarity.
6	**g** Globally finds all matches.
7	**cg** Allows the search to continue even after a global match fails.

Regular Expression Variables:

Regular expression variables include $, which contains whatever the last grouping match matched; $&, which contains the entire matched string; $`, which contains everything before the matched string; and $', which contains everything after the matched string. Following code demonstrates the result −

```
#!/usr/bin/perl
$string = "The food is in the salad bar";
$string =~ m/foo/;
```

```
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";
```

When above program is executed, it produces the following result −

```
Before: The
Matched: foo
After: d is in the salad bar
```

The Substitution Operator:

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is −

```
s/PATTERN/REPLACEMENT/;
```

The PATTERN is the regular expression for the text that we are looking for. The REPLACEMENT is a specification for the text or regular expression that we want to use to replace the found text with. For example, we can replace all occurrences of dog with cat using the following regular expression −

```
#/user/bin/perl
$string = "The cat sat on the mat";
$string =~ s/cat/dog/;
print "$string\n";
```

When above program is executed, it produces the following result −

```
The dog sat on the mat
```

Substitution Operator Modifiers:

Here is the list of all the modifiers used with substitution operator.

Sr.No.	Modifier & Description
1	**i** Makes the match case insensitive.
2	**m** Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary.
3	**o** Evaluates the expression only once.
4	**s** Allows use of . to match a newline character.
5	**x** Allows you to use white space in the expression for clarity.
6	**g** Replaces all occurrences of the found expression with the replacement text.
7	**e** Evaluates the replacement as if it were a Perl statement, and uses its return value as the replacement text.

Translation Operator Modifiers

Following is the list of operators related to translation.

Sr.No.	Modifier & Description
1	**c** Complements SEARCHLIST.

2	**d** Deletes found but unreplaced characters.
3	**s** Squashes duplicate replaced characters.

The /d modifier deletes the characters matching SEARCHLIST that do not have a corresponding entry in REPLACEMENTLIST.

Example

```
#!/usr/bin/perl
$string = 'the cat sat on the mat.';
$string =~ tr/a-z/b/d;
print "$string\n";
```

When above program is executed, it produces the following result −

```
b b   b.
```

More Complex Regular Expressions

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. Here's a quick cheat sheet −

Sr.No.	Pattern & Description
1	^ Matches beginning of line.
2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.
4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets.
6	*** Matches 0 or more occurrences of preceding expression.
7	+ Matches 1 or more occurrence of preceding expression.
8	**?** Matches 0 or 1 occurrence of preceding expression.
9	**{ n}** Matches exactly n number of occurrences of preceding expression.
10	**{ n,}** Matches n or more occurrences of preceding expression.
11	**{ n, m}** Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.

13	**\w** Matches word characters.
14	**\W** Matches nonword characters.
15	**\s** Matches whitespace. Equivalent to [\t\n\r\f].
16	**\S** Matches nonwhitespace.
17	**\d** Matches digits. Equivalent to [0-9].
18	**\D** Matches nondigits.
19	**\A** Matches beginning of string.
20	**\Z** Matches end of string. If a newline exists, it matches just before newline.
21	**\z** Matches end of string.
22	**\G** Matches point where last match finished.
23	**\b** Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
24	**\B** Matches nonword boundaries.
25	**\n, \t, etc.** Matches newlines, carriage returns, tabs, etc.
26	**\1...\9** Matches nth grouped subexpression.
27	**\10** Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.
28	**[aeiou]** Matches a single character in the given set
29	**[^aeiou]** Matches a single character outside the given set

File Handling

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - STDIN, STDOUT, and STDERR, which represent standard input, standard output and standard error devices respectively.

Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any new or existing file in Perl.

open FILEHANDLE, EXPR
open FILEHANDLE

sysopen FILEHANDLE, FILENAME, MODE, PERMS
sysopen FILEHANDLE, FILENAME, MODE

Open Function

Following is the syntax to open file.txt in read-only mode. Here less than < sign indicates that file has to be opend in read-only mode.

```
open(DATA, "<file.txt");
```

Here DATA is the file handle, which will be used to read the file. Here is the example, which will open a file and will print its content over the screen.

```
#!/usr/bin/perl
open(DATA, "<file.txt") or die "Couldn't open file file.txt, $!";
while(<DATA>) {
   print "$_";
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opend in the writing mode.

```
open(DATA, ">file.txt") or die "Couldn't open file file.txt, $!";
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it −

```
open(DATA, "+<file.txt"); or die "Couldn't open file file.txt, $!";
```

To truncate the file first −

```
open DATA, "+>file.txt" or die "Couldn't open file file.txt, $!";
```

You can open a file in the append mode. In this mode, writing point will be set to the end of the file.

```
open(DATA,">>file.txt") || die "Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it −

```
open(DATA,"+>>file.txt") || die "Couldn't open file file.txt, $!";
```

Following is the table, which gives the possible values of different modes

Sr.No.	Entities & Definition
1	**< or r** Read Only Access
2	**> or w** Creates, Writes, and Truncates

3	**>> or a** Writes, Appends, and Creates
4	**+< or r+** Reads and Writes
5	**+> or w+** Reads, Writes, Creates, and Truncates
6	**+>> or a+** Reads, Writes, Appends, and Creates

Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the close function. This flushes the filehandle's buffers and closes the system's file descriptor.

close FILEHANDLE
close
If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns true only if it could successfully flush the buffers and close the file.

close(DATA) || die "Couldn't close file properly";

Reading and Writing Files
Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

The <FILEHANDL> Operator
The main method of reading the information from an open filehandle is the <FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle. For example −

```
#!/usr/bin/perl
print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array −

```
#!/usr/bin/perl
open(DATA,"<import.txt") or die "Can't open data";
@lines = <DATA>;
close(DATA);
```
getc Function
The getc function returns a single character from the specified FILEHANDLE, or STDIN if none is specified −

getc FILEHANDLE
getc
If there was an error, or the filehandle is at end of file, then undef is returned instead.

read Function
The read function reads a block of information from the buffered filehandle: This function is used to read binary data from the file.

read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

print Function

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

print FILEHANDLE LIST
print LIST
print

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current output filehandle (STDOUT by default). For example −

```
print "Hello World!\n";
```

Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and generate another copy file file2.txt.

```
#!/usr/bin/perl
# Open file to read
open(DATA1, "<file1.txt");
# Open new file to write
open(DATA2, ">file2.txt");
# Copy data from one file to another.
while(<DATA1>) {
   print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```
#!/usr/bin/perl
rename ("/usr/test/file1.txt", "/usr/test/file2.txt" );
```

This function renames takes two arguments and it just renames the existing file.

Deleting an Existing File

Here is an example, which shows how to delete a file file1.txt using the unlink function.

```
#!/usr/bin/perl
unlink ("/usr/test/file1.txt");
```

Retrieving documents from the web with Perl

There are plenty of choices when you need to fetch a page or two from the Internet. We are going to see several simple examples using wget, curl, LWP::Simple, and HTTP::Tiny.

Wget:

While they are not Perl solutions, they can actually provide a quick solution for you. I think there are virtually no Linux distributions that don't come with either wget or curl. They are both command line tool that can download files via various protocols, including HTTP and HTTPS.

You can use the system function of Perl to execute external program so you can write the following:

```
my $url = 'https://perlmaven.com/';
system "wget $url";
```

However there is another, more straight-forward way to get the remote file in a variable. You can use the qx operator (what you might have seen as back-tick ``) instead of the system function, and you can

ask wget to print the downloaded file to the standard output instead of saving to a file. As qx will capture and return the standard output of the external command, this can provide a convenient way to download a page directly into a variable:

```
my $url = 'https://perlmaven.com/';
my $html = qx{wget --quiet --output-document=- $url};
```

--output-document can tell wget where to save the downloaded file. As a special case, if you pass a dash - to it, wget will print the downloaded file to the standard output.

Curl:

For curl the default behavior is to print to the standard output, and the --silent flag can tell it to avoid any other output.

This is the solution with curl:

```
my $url = 'https://perlmaven.com/';
my $html = qx{curl --silent $url};
```

The drawback in both cases it that you rely on external tools and you probably have less control over those than over perl-based solutions.

Get one page using LWP::Simple

Probably the most well know perl module implementing a web client is LWP and its sub-modules. LWP::Simple is a, not surprisingly, simple interface to the library.

The code to use it is very simple. It exports a function called get that fetch the content of a single URL:

```
use LWP::Simple qw(get);
my $url = 'https://perlmaven.com/';
my $html = get $url;
```

Get one page using HTTP::Tiny
For that HTTP::Tiny is much better even if the code is slightly longer:

```
use HTTP::Tiny;
my $url = 'https://perlmaven.com/';
my $response = HTTP::Tiny->new->get($url);
if ($response->{success}) {
   my $html = $response->{content};
}
```

HTTP::Tiny is object oriented, hence you first call the constructor new. It returns an object and on that object you can immediately call the get method.

Introduction to Ruby

Ruby is "A Programmer's Best Friend".

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan. Matsumoto is also known as –Matz‖ in the Ruby community.

Features of ruby:

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.
- Ruby can be installed in Windows and POSIX environments.
- Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
- Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.
- Ruby has a rich set of built-in functions, which can be used directly into Ruby scripts.

Interactive Ruby (IRb):

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line.

This tool comes along with Ruby installation so you have nothing to do extra to have IRb working.

Just type **irb** at your command prompt and an Interactive Ruby Session will start as given below:

```
$irb
irb 0.6.1(99/09/16)
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out irb(main):004:1> end
nil
irb(main):005:0> hello Hello
World
nil irb(main):006:0>
```

Ruby Syntax:

Let us write a simple program in ruby. All ruby files will have extension **.rb**. So, put the following source code in a **test.rb file.**

(**#!/usr/bin/ruby**. This is a path to the **Ruby** interpreter, which will execute the script.)

```
#!/usr/bin/ruby –w
puts "Hello, Ruby!";
```

Here, I assumed that you have Ruby interpreter available in /usr/bin directory. Now, try to run this program as follows:

```
$ ruby test.rb
```

This will produce the following result:

```
Hello, Ruby!
```

Whitespace in Ruby Program:

Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings. Sometimes, however, they are used to interpret ambiguous statements. Interpretations of this sort produce warnings when the -w option is enabled.

Example:

```
a + b is interpreted as a+b ( Here a is a local variable)
a  +b is interpreted as a(+b) ( Here a is a method call)
```

Line Endings in Ruby Program:

Ruby interprets semicolons and newline characters as the ending of a statement. However, if Ruby encounters operators, such as +, -, or backslash at the end of a line, they indicate the continuation of a statement.

Ruby Identifiers:
Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive. It mean Ram and RAM are two different idendifiers in Ruby. Ruby identifier names may consist of alphanumeric characters and the underscore character (_).

Reserved Words:
The following list shows the reserved words in Ruby. These reserved words may not be used as constant or variable names. They can, however, be used as method names.

BEGIN	do	next	then
END	else	nil	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	r e d o	until
break	false	rescue	when
case	for	retry	while
class	if	return	yield
def	in	self	_FILE_____
defined?	module	super	__LINE_____

Here Document in Ruby:
"Here Document" refers to build strings from multiple lines. Following a << you can specify a string or an identifier to terminate the string literal, and all lines following the current line up to the terminator are the value of the string.
If the terminator is quoted, the type of quotes determines the type of the line-oriented string literal. Notice there must be no space between << and the terminator.
Here are different examples:

```
#!/usr/bin/ruby -w
print <<EOF
   This is the first way of creating
   here document ie. multiple line string.
EOF
print <<"EOF";                # same as above
   This is the second way of creating
   here document ie. multiple line string.
EOF
print <<`EOC`                 # execute commands
              echo hi there
              EOC
print <<"foo", <<"bar"  # you can stack them
              I said foo.
foo
              I said bar.
bar
```

Ruby *BEGIN* Statement Syntax:
Declares *code* to be called before the program is run

```
BEGIN {
      code
}
```

Example:

```
#!/usr/bin/ruby
puts "This is main Ruby Program"
 BEGIN {
      puts "Initializing Ruby Program"
}
```

This will produce the following result:

```
Initializing Ruby Program This is
main Ruby Program
```

Ruby *END* Statement Syntax:
Declares *code* to be called at the end of the program.

```
 END {
```

```
    code
}
```

Example:

```
#!/usr/bin/ruby
puts "This is main Ruby Program"
END {
    puts "Terminating Ruby Program"
}
BEGIN {
    puts "Initializing Ruby Program"
}
```

This will produce the following result:

```
Initializing Ruby Program This is
main Ruby Program Terminating
Ruby Program
```

Ruby Comments:

- A comment hides a line, part of a line, or several lines from the Ruby interpreter. You can use the hash character (#) at the beginning of a line:Or, a comment may be on the same line after a statement or expression
- You can comment multiple lines as follows:
  # This is a comment.
  # This is a comment, too. # This is a comment, too. # I said that already.
- Here is another form. This block comment conceals several lines from the interpreter with =begin/=end:

```
=begin
This is a  comment. This is
a comment, too. This is a
comment, too. I said that
already.
=end
```

Scalar Types and Their Operations

Ruby has three categories of data types:

1. Scalars
2. Arrays, and
3. Hashes.

The most commonly used types: scalars. There are two categories of scalar types: numeric‘s and character strings.

Integer Numbers:

- Ruby supports integer numbers. An integer number can range from -2^{30} to 2^{30-1} or -2^{62} to 2^{62-}
- Integers with-in this range are objects of class *Fixnum* and integers outside this range are stored in objects of class *Bignum*.
- You write integers using an optional leading sign, an optional base indicator (0 for octal, 0x for hex, or 0b for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.
- You can also get the integer value corresponding to an ASCII character or escape sequence by preceding it with a question mark.

Example:

```
123                     # Fixnum decimal
1_234                   # Fixnum decimal with underline
-500                    # Negative Fixnum
0377                    # octal
0xff                    # hexadecimal
0b1011                  # binary
?a                      # character code for 'a'
?\n                     # code for a newline (0x0a)
12345678901234567890    # Bignum
```

Floating Numbers:

Ruby supports integer numbers. They are also numbers but with decimals. Floating-point

numbers are objects of class *Float* and can be any of the following:

Example:

```
123.4                   # floating point value
1.0e6                   # scientific notation
4E20                    # dot not required
4e+20                   # sign before exponential
```

String Literals:

- Ruby strings are simply sequences of 8-bit bytes and they are objects of class String.
- Double- quoted strings allow substitution and backslash notation but single-quoted strings don't allow substitution and allow backslash notation only for \\ and \'

Example:

```
#!/usr/bin/ruby -w
puts 'escape using "\\"'; puts
'That\'s right';
```

This will produce the following result:

```
escape using "\"
That's right
```

You can substitute the value of any Ruby expression into a string using the sequence **#{ expr}**. Here, expr could be any ruby expression.

```
#!/usr/bin/ruby -w
puts "Multiplication Value : #{24*60*60}"; This will produce the following result: Multiplication Value : 86400
```

Backslash Notations:

Following is the list of Backslash notations supported by Ruby:

Notation	*Character represented*
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\a	Bell (0x07)
\e	Escape (0x1b)
\s	Space (0x20)
\nnn	Octal notation (n being 0-7)
\xnn	Hexadecimal notation (n being 0-9, a-f, or A-F)
\x	Character x

Ruby Ranges:

- A Range represents an interval a set of values with a start and an end. Ranges may be constructed using the s..e and s...e literals, or with Range.new.
- Ranges constructed using .. run from the start to the end inclusively. Those created using ... exclude the end value. When used as an iterator, ranges return each value in the sequence.
- A range (1..5) means it includes 1, 2, 3, 4, 5 values and a range (1...5) means it includes 1, 2, 3, 4 values.

Example:

```ruby
#!/usr/bin/ruby

(10..15).each do |n|
   print n, ' '
end
```

This will produce the following result:

```
10 11 12 13 14 15
```

Variables and Assignment Statements

- The form of variable names is a lowercase letter or an underscore, followed by any number of uppercase or lowercase letters, digits, or underscores.
- The letters in a variable name are case sensitive, meaning that fRIZZY, frizzy, frIzZy, and friZZy are all distinct names.
- Programmer-defined variable names do not include uppercase letters.
- double-quoted string literals can include the values of variables specified by placing the code in braces and preceding the left brace with a pound sign (#).

For example, if the value of *tue_high* is *83*, then the string *"Tuesday's high temperature was*

#{tue_high}" has the following value: *"Tuesday's high temperature was 83"*

A scalar variable that has not been assigned a value by the program has the value nil.

Variables in a Ruby Class:
Ruby provides four types of variables:

- **Local Variables:** Local variables are the variables that are defined in a method. Local variables are not available outside the method. Local variables begin with a lowercase letter or _.
- **Instance Variables:** Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
- **Class Variables:** Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
- **Global Variables:** Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

Ruby Global Variables: Global variables begin with **@.**

- Uninitialized global variables have the value *nil* and produce warnings with the -w option.
- Assignment to global variables alters global status. It is not recommended to use global variables. They make programs cryptic.

Here is an example showing usage of global variable.

```
#!/usr/bin/ruby
$global_variable = 10 class
Class1
   def print_global
        puts "Global variable in Class1 is #$global_variable" end
end
class Class2
   def print_global
        puts "Global variable in Class2 is #$global_variable" end
end
class1obj     =     Class1.new
class1obj.print_global
class2obj     =     Class2.new
class2obj.print_global
```

Here $global_variable is a global variable. This will produce the following result:
NOTE: In Ruby you can access value of any variable or constant by putting a hash (#) character just before that variable or constant.

```
Global variable in Class1 is 10 Global  variable in Class2 is 10
```

Ruby Instance Variables: Instance variables begin with **@.**
Uninitialized instance variables have the value *nil* and produce warnings with the -w option.
Here is an example showing usage of Instance Variables.

```
#!/usr/bin/ruby
class Customer
     def initialize(id, name, addr)
          @cust_id=id
      @cust_name=name
     @cust_addr=addr
     end
     def display_details()
          puts "Customer id #@cust_id"
          puts "Customer name #@cust_name"
          puts "Customer address #@cust_addr" end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
# Call Methods
```

```
cust1.display_details()
cust2.display_details()
```

Here, @cust_id, @cust_name and @cust_addr are instance variables.

Ruby Class Variables: Class variables begin with **@@**

- It must be initialized before they can be used in method definitions.
- Referencing an uninitialized class variable produces an error.
- Class variables are shared among descendants of the class or module in which the class variables are defined.
- Overriding class variables produce warnings with the -w option

Here is an example showing usage of class variable:

```
#!/usr/bin/ruby
class Customer
    @@no_of_customers=0
    def initialize(id, name, addr)
        @cust_id=id  @cust_name=name
        @cust_addr=addr
        @@no_of_customers += 1
    end
    def display_details()
        puts "Customer id #@cust_id" puts
        "Customer name #@cust_name"
        puts "Customer address #@cust_addr" end

        def  total_no_of_customers()
    puts "Total number of customers: #@@no_of_customers" end
end
# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
# Call Methods cust1.total_no_of_customers()
cust2.total_no_of_customers()
```

Here @@no_of_customers is a class variable.

Ruby Local Variables: Local variables begin with a lowercase letter or _.

- The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.
- When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.
- Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached.
- The lifetime of local variables is determined when Ruby parses the program.

Ruby Constants

- Ruby has constants, which are distinguished from variables by their names, which always begin with uppercase letters.
- A constant is created when it is assigned a value, which can be any constant expression.
- In Ruby, a constant can be assigned a new value, although it causes a warning message to the user.
- Ruby includes some predefined, or implicit, variables.
- The name of an implicit scalar variable begins with a dollar sign. The rest of the name is often just one more special character, such as an underscore (_), a circumflex (^), or a backslash (\).

Example:

```
#!/usr/bin/ruby
class Example
    VAR1 = 100
    VAR2 = 200
    def show
        puts "Value of first Constant is #{VAR1}"
         puts "Value of second Constant is #{VAR2}"
    end
end
```

```
# Create Objects
object=Example.new()
object.show
```

Ruby Pseudo-Variables

They are special variables that have the appearance of local variables but behave like constants. You cannot assign any value to these variables.

- **self:** The receiver object of the current method.
- **true:** Value representing true.
- **false:** Value representing false.
- **nil:** Value representing undefined.
- **__FILE :** The name of the current source file.
- **__LINE :** The current line number in the source file.

Numeric Operators

Most of Ruby's numeric operators are similar to those in other common programming languages, so they should be familiar to most readers. There are the binary operators: + for addition, - for subtraction, * for multiplication, / for division, ** for exponentiation, and % for modulus. The modulus operator is defined as follows: x % y produces the remainder of the value of x after division by y. If an integer is divided by an integer, integer division is done. Therefore, 3 / 2 produces 1.

Operator	Associativity
**	Right
unary +, –	Right
*, /, %	Left
binary +, –	Left

The operators listed first have the highest precedence.

- Ruby does not include the increment (++) and decrement (--) operators.

Ruby includes the Math module, which has methods for basic trigonometric and transcendental functions. Among these methods are cos (cosine), sin (sine), log (logarithm), sqrt (square root), and tan (tangent). The methods of the Math module are referenced by prefixing their names with Math., as in Math.sin(x). All of these take any numeric type as a parameter and return a Float value.

Included with the Ruby implementation is an interactive interpreter, which is very useful to the student of Ruby. It allows one to type any Ruby expression and get an immediate response from the interpreter. The interactive interpreter's name is Interactive Ruby, whose acronym, IRB, is the name of the program that supports it. For example, if the command prompt is a percent sign (%), one can type % irb

after which irb will respond with its own prompt, which is *irb(main):001:0>*

At this prompt, any Ruby expression or statement can be typed, whereupon irb interprets the expression or statement and returns the value after an implication symbol (=>), as in the following example:

```
irb(main):001:0> 17 * 3
=> 51
irb(main):002:0>
```

The lengthy default prompt can be easily changed. We prefer the simple –>>– prompt. The default prompt can be changed to this with the following command:

```
irb(main):002:0> conf.prompt_i = ">>"
```

From here on, we will use this simple prompt.

String Methods

The Ruby String class has more than 75 methods. Many of these methods can be used as if they were operators. In fact, we sometimes call them operators, even though underneath they are all methods.

The String method for catenation is specified by plus (+), which can be used as a binary operator. This method creates a new string from its operands:

```
>>"Happy"+" "+"Holidays!"
=> "Happy Holidays!"
```

The << method appends a string to the right end of another string, which, of course, makes sense only if the left operand is a variable. Like +, the << method can be used as a binary operator. For example, in the interactions

```
>>mystr="G'day, "
=> "G'day, "
>>mystr << "Friend"
=> "G'day, Friend"
```

The other most commonly used methods of Ruby are similar to those of other programming languages. Among these are the ones shown in Table below; all of them create new strings.

Method	Action
capitalize	Converts the first letter to uppercase and the rest of the letters to lowercase
chop	Removes the last character
chomp	Removes a newline from the right end if there is one
upcase	Converts all of the lowercase letters in the object to uppercase
downcase	Converts all of the uppercase letters in the object to lowercase
strip	Removes the spaces on both ends
lstrip	Removes the spaces on the left end
rstrip	Removes the spaces on the right end
reverse	Reverses the characters of the string
swapcase	Converts all uppercase letters to lowercase and all lowercase letters to uppercase

As stated previously, all of these methods produce new strings, rather than modify the given string in place. However, all of the methods also have versions that modify their objects in place. These methods are called bang or mutator methods and are specified by following their names with an exclamation point (!). To illustrate the difference between a string method and its bang counterpart, consider the following interactions:

```
>>str="Welcome"
=> "Welcome"
>>str.upcase
=> "WELCOME"
>>str
=> "Welcome"
>>str.upcase!
=> "WELCOME"
>>str
=> "WELCOME"
```

Ruby strings can be indexed, somewhat as if they were arrays. To get the character

```
=> "Welcome"
>>str[0]
=> "W"
>>str[-1]
=> "e"
```

:

Note: If a negative subscript is used as an index, the position is counted from the right.

A multi-character substring of a string can be accessed by including two numbers in the brackets, in which case the first is the position of the first character of the substring and the second is the number of characters in the substring:

```
>>str="Welcome"
=> "Welcome"
>>str[3,4]
=> "come"
```

The usual way to compare strings for equality is to use the == method as an operator:

```
>>"abc" == "abc"
=> true
>>"abc" == "abd"
=> false
```

To facilitate ordering, Ruby includes the –spaceship‖ operator, <=>, which returns -1 if the second operand is greater than the first, 0 if the two operands are equal, and 1 if the first operand is greater than the second.

Greater‖ in this case means that the text in question belongs later alphabetically. The following interactions illustrate all three cases:

```
>>"apple" <=> "banana"
=> -1
>>"banana" <=> "apple"
=> 1
>>"grape" <=> "grape"
=> 0
```

The repetition operator is specified with an asterisk (*). It takes a string as its left operand and an expression that evaluates to a number as its right operand. The left operand is replicated the number of times equal to the value of the right operand:

```
>>"cse! "*3
=> "cse! cse! cse! "
```

Simple Input and Output

Screen Output (*puts*)

Output is directed to the screen with the *puts* method (or operator). The operand for puts is a string. A newline character is implicitly appended to the string. If the value of a variable is to be part of a line

```
>>name="Ruby"
=> "Ruby"
>>puts "My name is #{name}"
My name is Ruby
=> nil
```

of output, the #{...} notation can be used to insert it into a double-quoted string, as in the following interactions:

The value returned by puts is nil, and that is the value returned after the string has been displayed. Use *print*
method if you do not want to append a newline at the end of your string.

Keyboard Input (*gets*)

The *gets* method gets a line of input from the keyboard. The retrieved line includes the newline character. If the newline is not needed, it can be discarded with *chomp*:

```
>>name= gets
apples
=> "apples\n"
```

This code could be done by applying chomp directly to the value returned by gets:

```
>>name= gets.chomp
apples
=> "apples"
```

If a number is to be input from the keyboard, the string from gets must be converted to an integer with the *to_i*
method, as in the following interactions:

```
>>age=gets.to_i
27
=> 27
```

If the number is a floating-point value, the conversion method is *to_f*:

```
>>age=gets.to_f
27
=> 27.0
```

In this same way, *to_s* which converts the value of the object to a string.

```
>>age=gets.to_s
27
=> "27\n"
```

The following program created with a text editor and stored in a file with *.rb* extension:

```
## quadeqn.rb - A simple Ruby
program ## Get input
print "Enter the value of
a:" a = gets.to_i
print "Enter the value of
b:" b = gets.to_i
print "Enter the value of
c:" c = gets.to_i
print "Enter the value of
x:" x = gets.to_i
res = a * x ** 2 + b * x + c
puts "The value of the expression is: #{res}
```

A program stored in a file can be run by the command

>ruby –w filename

So, our example program can be run (interpreted) with

```
>ruby -w quadeqn.rb
Enter the value of a:1
Enter the value of b:2
Enter the value of c:3
Enter the value of x:1
The value of the expression is: 6
```

If the program is found to be syntactically correct, the response to the following command is:

```
>ruby -cw quadeqn.rb
Syntax OK
```

Arrays

Ruby includes two structured classes or types: arrays and hashes.

Arrays in Ruby are more flexible than those of most of the other common languages. This flexibility is a result of two fundamental differences between Ruby arrays and those of other common languages such as C, C++, and Java. First, the length of a Ruby array is dynamic: It can grow or shrink anytime during program execution. Second, a Ruby array can store different types of data. For example, an array may have some numeric elements, some string elements, and even some array elements.

Ruby arrays can be created in two different ways. First, an array can be created by sending the new message to the predefined Array class, including a parameter for the size of the array. The second way is simply to assign a list literal to a variable, where a list, literal is a list of literals delimited by brackets. For example, in the following interactions, the first array is created with new and the second is created by assignment:

```
>>list1 = Array.new(5)
=> [nil, nil, nil, nil, nil]
>>list2 = [2,2.145,"Vijay",[]]
=> [2, 2.145, "Vijay", []]
```

An array created with the new method can also be initialized by including a second parameter, but every element is given the same value. Thus, we may have the following interactions:

```
>>list = Array.new(5,"Hi")
=> ["Hi", "Hi", "Hi", "Hi", "Hi"]
```

All Ruby array elements use integers as subscripts, and the lower bound subscript of every array is zero. Array elements are referenced through subscripts delimited by brackets ([]).

The length of an array can be retrieved with the *length* method, as illustrated in the following interactions:

```
=> [2, 2.145, "Vijay", []]
>>list2.length
=> 4
```

Built-In Methods for Arrays and Lists

Ruby has four methods for adding elements at the beginning or at end of the array:

1. ***unshift( )*:** The unshift method takes a scalar or an array literal as a parameter and appends it to the beginning
2. ***shift( )*:** Removes and returns the first element of the array
3. ***push( ):*** The push method takes a scalar or an array literal and adds it to the end of the array:
4. ***pop( ):*** Removes and returns the last element of the array

```
>>list = [2,3,4]
=> [2, 3, 4]
>>list.push(5,6)
=> [2, 3, 4, 5, 6]
>>list.pop()
=> 6
>>list.unshift(10,20)
=> [10, 20, 2, 3, 4, 5]
>>list.shift()
=> 10
>>list
=> [20, 2, 3, 4, 5]
```

5. **max and min:** return the smallest or largest element in an array respectively

```
>>list
=> [20, 2, 3, 4, 5]
>>list.max
=> 20
>>list.min
=> 2
```

6. **uniq:** returns an array with no duplicate elements

```
>>list = [1,2,3,4,3,2,1]
=> [1, 2, 3, 4, 3, 2, 1]
>>list.uniq
=> [1, 2, 3, 4]
```

7. **compact** – return an array with no nil elements

```
>>list = Array.new(5)
=> [nil, nil, nil, nil, nil]
>>list[1]=10
=> 10
>>list[4]=20
=> 20
>>list[2]=30
=> 30
>>list
=> [nil, 10, 30, nil, 20]
>>list.compact
=> [10, 30, 20]
```

8. **Set operations:** There are three methods that perform set operations on two arrays: **&** for set intersection;

- for set difference, and **|** for set union.

```
>>list1 = [1,2,3,2,1,4]
=> [1, 2, 3, 2, 1, 4]
>>list2 = [1,2,3,5,4,2]
=> [1, 2, 3, 5, 4, 2]
>>list1 & list2
=> [1, 2, 3, 4]
>>list1 | list2
=> [1, 2, 3, 4, 5]
>>list1 - list2
=> []
```

Hashes

Associative arrays are arrays in which each data element is paired with a key, which is used to identify the data element; associative arrays often are called ***hashes***. There are two fundamental differences between arrays and hashes: First, arrays use numeric subscripts to

address specific elements, whereas hashes use string values (the keys) to address elements; second, the elements in arrays are ordered by subscript, but the elements in hashes are not.

Like arrays, hashes can be created in two ways, with the new method or by assigning a value to a variable. In the latter case, the value is a hash value, in which each element is specified by a key– value pair, separated by the symbol =>. Hash literals are delimited by braces, as in the following manner:

```
>>ages = {"raju"=>20,"king"=>21,"anand"=>19}
=> {"raju"=>20, "king"=>21, "anand"=>19}
```

If the *new* method is sent to the Hash class without a parameter, it creates an empty hash, denoted by {}:

```
>>my_hash = Hash.new
=> {}
```

An individual element of a hash can be referenced by –subscripting‖ the hash name with a key.

```
>>ages['raju']
=> 20
```

A new value is added to a hash by assigning the value of the new element to a reference to

```
>>ages['gopi'] = 39
=> 39
>>ages
=> {"raju"=>20, "king"=>21, "anand"=>19, "gopi"=>39}
```

the key of the new element, as in the following example:

An element is removed from a hash with the delete method, which takes an element key as a parameter:

```
>>ages.delete('raju')
=> 20
>>ages
=> {"king"=>21, "anand"=>19, "gopi"=>39}
```

The keys and values of a hash can be extracted into arrays with the methods keys and values,

```
>>ages.keys
=> ["king", "anand", "gopi"]
>>ages.values
=> [21, 19, 39]
```

respectively:

The has_key? predicate method is used to determine whether an element with a specific key is in a hash.

```
>>ages
=> {"king"=>21, "anand"=>19, "gopi"=>39}
>>ages.has_key?("gopi")
=> true
>>ages.has_key?("henry")
=> false
```

A hash can be set to empty in one of two ways: either an empty hash value can be assigned to the hash, or the clear method can be used on the hash. These two approaches are illustrated with the following statements:

```
>>ages = {"raju"=>20,"king"=>21,"anand"=>19}
=> {"raju"=>20, "king"=>21, "anand"=>19}
>>ages = {}
=> {}
>>temp = {"mon"=>45,"tue"=>42,"wed"=>39}
=> {"mon"=>45, "tue"=>42, "wed"=>39}
>>temp.clear
=> {}
```

Control Statements

Selection Statements

Ruby offers conditional structures that are pretty common to modern languages. Here, we will explain all the conditional statements and modifiers available in Ruby.

Ruby's *if* statement is similar to that of other languages. One syntactic difference is that there are no parentheses around the control expression, as is the case with most of the languages based directly or even loosely on C. The following construct is illustrative:

if...else Statement Syntax

```
if conditional [then]
        code...
[else
        code...]
end
```

Executes *code* if the *conditional* is true otherwise *code* specified in the *else* clause is executed.

```
#!/usr/bin/ruby

x = 1
if x > 2
   puts "x is greater than 2"
elsif x <= 2 and x!=0
   puts "x is 1"
else
   puts "I can't guess the number"
end
```

if elsif else Statement

```
if expr1 === _tmp || expr2 === _tmp
   stmt1
elsif expr3 === _tmp || expr4 === _tmp
   stmt2
else
   stmt3
end
```

unless Statement

Ruby has an *unless* statement, which is the same as its if statement, except that the inverse of the value of the control expression is used. The following construct illustrates an *unless* statement:

Syntax

unless conditional [then] code

else[code]
end

case statement

Ruby includes two kinds of multiple selection constructs, both named case. One Ruby case construct, which is similar to a switch, has the following form:

```
case expression
when value then
 - statement sequence
...
when value then
 - statement sequence
[else
 - statement sequence]
end
```

The value of the case expression is compared with the values of the when clauses, one at a time, from top to bottom, until a match is found, at which time the sequence of statements that follow is interpreted. The comparison is done with the === relational operator, which is defined for all built-in classes. Consider the following example:

```
print "Enter the value of
in_val:" in_val = gets.to_i
case in_val
when -1
then
        neg_count += 1
when 0 then
        zero_count += 1
when 1 then
        pos_count += 1
else
    print "Error - in_val is out of range"
end
```

Second form is:

```
case
when Boolean expression then expression
...
when Boolean expression then expression
else expression
end
```

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

Looping Statements

Loops in Ruby are used to execute the same block of code a specified number of times.

The while Statement:

Executes *code* while *condition* is true. A *while* loop's *condition* is separated from *code* by the reserved word *do*.

Syntax:

```
while condition
    [do] code
end
```

Example

```
i = 1
while i < 5 do
    puts "Inside the loop i =
    #{$i}" i = i + 1

end
```

This will produce the following

```
result: Inside the loop i = 1
Inside the loop i =
2 Inside the loop i
= 3 Inside the
loop i = 4
```

while modifier Syntax

```
begin
    code
end while condition
```

Executes *code* while *condition* is true. Irrespective of the condition, *code* is executed once before condition is evaluated.

Example

```
i = 1
begi
n
    puts "Inside the loop i =
    #{$i}" i= i + 1
end while i < 5
```

This will produce the following

```
result: Inside the loop i =
1
Inside the loop i =
2 Inside the loop i
= 3 Inside the
```

```
loop i = 4
```

Until Statement Syntax:

```
until condition [do]
        code
end
```

Executes *code* while *condition* is false. An *until* statement's condition is separated from *code* by the reserved word *do*.

Example

```
i = 1
until i > 5 do
        puts "Inside the loop i =
        #{i}" i = i+1
end
```

This will produce the following

```
result: Inside the loop i = 1
Inside the loop i =
2 Inside the loop i
= 3 Inside the
loop i = 4 Inside
the loop i = 5
```

until modifier Syntax:

```
begin
        code
end until condition
```

Executes *code* while *condition* is false. If an *until* modifier follows a *begin* statement with no clauses, *code* is executed once before *conditional* is evaluated.

Example

```
i = 0
begi
n
        puts "Inside the loop i =
        #{i}" i = i+1
end until i > 5
```

This will produce the following

```
result: Inside the loop i =
1
Inside the loop i =
2 Inside the loop i
= 3 Inside the
loop i = 4
Inside the loop i = 5
```

For Statement Syntax

```
for variable [, variable ...] in expression
        [do] code
end
```

Executes *code* once for each element in *expression*.

Example

```
for i in 1..5
        puts "Value of local variable is #{i}"
end
```

Here, we have defined the range 1..5. The statement for *i* in 0..5 will allow *i* to take values in the range from 1 to 5 (including 5). This will produce the following result:

```
Value of local variable
is 1 Value of local
variable is 2 Value of
local variable is 3 Value
of local variable is 4
Value of local variable
```

```
is 5
```

A ***for...in*** loop is almost exactly equivalent to the following: (expression).each do |variable[, variable...]|

```
        code
end
```

Example

```
(1..5).each do |i|
        puts "Value of local variable is #{i}"
end
```

This will produce the following result: Value of local variable is 1 Value of local variable is 2 Value of local variable is 3 Value of local variable is 4 Value of local variable is 5

Break Statement

Syntax

```
break
```

Terminates the loop. Terminates a method with an associated block if called within the block

Example

```
for i in 1..5
        if i > 2 then
                break
        end
        puts "Value of local variable is #{i}"
end
```

This will produce the following

```
result: Value of local
variable is 1 Value of local
variable is 2
```

next Statement Syntax

```
next
```

breaks the current iteration and place the loop in next iteration. Terminates execution of a block if called within a block

Example:

```
for i in 1..5
        if i > 2 then
                next
        end
        puts "Value of local variable is #{i}"
end
```

This will produce the following

```
result: Value of local
variable is 1 Value of local
variable is 2
```

BLOCKS: Similarly, Ruby has a concept of Block.

- A block consists of chunks of code.
- You assign a name to a block.
- The code in the block is always enclosed within braces ({}).

- A block is always invoked from a function with the same name as that of the block. This means that if you have a block with the name *test*, then you use the function *test* to invoke this block.
- You invoke a block by using the *yield* statement.

Syntax

```
block_name {
   statement1
   statement2
   .....
}
```

Here, you will learn to invoke a block by using a simple *yield* statement. You will also learn to use a *yield* statement with parameters for invoking a block. You will check the sample code with both types of *yield* statements.

The yield Statement

Let's look at an example of the yield statement −

```
#!/usr/bin/ruby

def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end
test {puts "You are in the block"}
```

This will produce the following result −

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

You also can pass parameters with the yield statement. Here is an example −

```
#!/usr/bin/ruby

def test
   yield 5
   puts "You are in the method test"
   yield 100
end
test {|i| puts "You are in the block #{i}"}
```

This will produce the following result −

```
You are in the block 5
You are in the method test
You are in the block 100
```

Here, the *yield* statement is written followed by parameters. You can even pass more than one parameter. In the block, you place a variable between two vertical lines (||) to accept the parameters. Therefore, in the preceding code, the yield 5 statement passes the value 5 as a parameter to the test block.

Now, look at the following statement −

```
test {|i| puts "You are in the block #{i}"}
```

Here, the value 5 is received in the variable *i*. Now, observe the following *puts* statement −

```
puts "You are in the block #{i}"
```

The output of this *puts* statement is −

You are in the block 5

If you want to pass more than one parameters, then the *yield* statement becomes −

```
yield a, b
```

and the block is −

```
test {|a, b| statement}
```

The parameters will be separated by commas.

Iterators

Iterators are nothing but methods supported by *collections*. Objects that store a group of data members are called collections. In Ruby, arrays and hashes can be termed collections. *Iterators* return all the elements of a collection, one after the other. The syntax is

object.iterator { | ***value*** |
statement } or
object.iterator do |***value***|
statements
end

The object is typically an array, a range, or a hash

The times iterator

The *times* iterator method provides a way to build simple counting loops. Typically, *times* is added to

```
>>5.times {puts "Hi..!"}
Hi..!
Hi..!
Hi..!
Hi..!
Hi..!
=> 5
```

a number object, which repeats the attached block that number of times. Consider the following example:

The each iterator

The most common iterator is *each*, which is often used to go through arrays and apply a block to each element.

```
>>list = [2,4,6,8]
=> [2, 4, 6, 8]
>>list.each do |value| puts "#{value}" end
2
4
6
8
=> [2, 4, 6, 8]
```

The upto iterator

Iterates through successive values, starting at *start* and ending at *last* inclusive, passing each value in turn to the block.

```
>>5.upto(8){ |v| puts v }
5
6
7
8
=> 5
```

The downto iterator

This decrements a number. It reduces the number by 1 after each pass through the iterator body. If the argument is not lower, the iterator body is not executed.

```
>>5.downto(3){ |v| puts v }
5
4
3
=> 5
```

The step iterator

The step iterator method takes a terminal value and a step size as parameters and generates the values from that of the object to which it is sent and the terminal value:

```
>>10.step(20,3){ |v| puts v }
10
13
16
19
=> 10
```

The each_char iterator

get each character (as an integer) from a string

```
>>str="CSE"
=> "CSE"
>>str.each_char {|c| puts c }
C
S
E
=> "CSE"
```

The collect iterator

the *collect* iterator method takes the elements from an array, one at a time, and puts the values generated by the given block into a new array:

```
>>list=[5,10,15,20]
=> [5, 10, 15, 20]
>>list.collect {|v| v = v-5 }
=> [0, 5, 10, 15]
>>list
=> [5, 10, 15, 20]
>>list.collect! {|v| v = v+5 }
=> [10, 15, 20, 25]
>>list
=> [10, 15, 20, 25]
```

The each_key iterator

Operates on hashes and returns all the keys of hash

```
>>ages = {"raju"=>20,"ravi"=>21,"kiran"=>19}
=> {"raju"=>20, "ravi"=>21, "kiran"=>19}
>>ages.each_key do |key| puts key end
raju
ravi
kiran
```

The each_value iterator

Operates on hashes and returns all the values of hash

```
>>ages = {"raju"=>20,"ravi"=>21,"kiran"=>19}
=> {"raju"=>20, "ravi"=>21, "kiran"=>19}
>>ages.each_value do |value| puts value end
20
21
19
```

The each_pair iterator
Operates on hashes and returns both keys and values of hash

```
>>ages = {"raju"=>20,"ravi"=>21,"kiran"=>19}^Z
=> {"raju"=>20, "ravi"=>21, "kiran"=>19}
>>ages.each_pair do |key,value| puts "#{key} = #{value}" end
raju = 20
ravi = 21
kiran = 19
```

Classes

A class is defined as a template for objects, of which any number can be created. An object has a state, which is maintained in its collection of instance variables, and a behavior, which is defined by its methods. An object can also have constants and a constructor.

Defining a Class in Ruby:
A class in Ruby always starts with the keyword *class* followed by the name of the class. The name should always be in initial capitals. The class *Customer* can be displayed as:class Customer end
You terminate a class by using the keyword *end*. All the data members in the *class* are between the class definition and the *end* keyword.

The Basics of Classes
The methods and variables of a class are defined in the syntactic container that has the following form:

class class_name
... end

- Class names, like constant names, must begin with uppercase letters.
- Instance variables are used to store the state of an object. They are defined in the class definition, and every object of the class gets its own copy of the instance variables.
- The name of an instance variable must begin with an at sign (@), which distinguishes instance variables from other variables.
- A class can have a single constructor, which in Ruby is a method with the name *initialize*, which is used to initialize instance variables to values.
- A constructor can take any number of parameters, which are treated as local variables; therefore, their names begin with lowercase letters or underscores.
- Classes in Ruby are dynamic in the sense that members can be added at any time
- Methods can also be removed from a class, by providing another class definition in which themethod to be removed is sent to the method *remove_method* as a parameter.

Creating Objects in Ruby using *new* Method:
You can create objects in Ruby by using the method *new* of the class.
The method *new* is a unique type of method, which is predefined in the Ruby library. The new method belongs to the *class* methods.
Here is the example to create two objects cust1 and cust2 of the class Customer:
cust1 = Customer. new
cust2 = Customer. new
Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword *new* will follow.

Custom Method to create Ruby Objects :
You can pass parameters to method *new* and those parameters can be used to initialize class variables.
When you plan to declare the *new* method with parameters, you need to declare the method *initialize* at the time of the class creation.
The *initialize* method is a special type of method, which will be executed when the *new* method of the class is called with parameters.
Here is the example to create initialize method:

```
class Customer
    @@no_of_customers=0
    def initialize(id, name, addr)
        @cust_id=id
        @cust_name=name
        @cust_addr=addr
    end
end
```

In the *initialize* method, you pass on the values of these local variables to the instance variables @cust_id, @cust_name, and @cust_addr. Here local variables hold the values that are passed along with the new method.
Now, you can create objects as follows:

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

Member Functions in Ruby Class:
In Ruby, functions are called methods. Each method in a *class* starts with the keyword *def* followed by the method name.
The method name always preferred in **lowercase letters**. You end a method in Ruby by using the keyword *end*.

Now in the following example, create one object of Sample class and call *hello* method and see the result:

```
#!/usr/bin/ruby

class Sample
    def hello
        puts "Hello Ruby!" end
end

# Now using above class to create objects object =
Sample. new
object.hello
```

This will produce the following result: Hello Ruby!

Ruby Methods

Ruby methods are very similar to functions in any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit.
Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.
Methods should be defined before calling them, otherwise Ruby will raise an exception for undefined method invoking.
Syntax:

```
def method_name
    expr..
end
```

You can represent a method that accepts parameters like this:

```
def method_name (var1, var2) expr..
end
```

You can set default values for the parameters which will be used if method is called without passing

required parameters:

```
def method_name (var1=value1, var2=value2) expr..
end
```

Whenever you call the simple method, you write only the method name as follows:

```
method_name
```

However, when you call a method with parameters, you write the method name along with the parameters, such as:

```
method_name 25, 30
```

Return Values from Methods:

Every method in Ruby returns a value by default. This returned value will be the value of the last statement. For example:

```
def test
    i = 100
    j = 10
    k = 0 end
```

This method, when called, will return the last declared variable k.

Ruby *return* Statement:

The *return* statement in ruby is used to return one or more values from a Ruby Method.

Syntax:

```
return [expr[`,' expr...]]
```

If more than two expressions are given, the array containing these values will be the return value.
If no expression given, nil will be the return value.

Example:

```
return
OR
return 12
OR
return 1,2,3
```

Have a look at this example:

```
#!/usr/bin/ruby
def test
    i = 100
    j = 200
    k = 300
return i, j, k end
var = test
puts var
```

Class Methods:

When a method is defined outside of the class definition, the method is marked as *private* by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the *private* mark of the methods can be changed by *public* or *private* of the Module.

Whenever you want to access a method of a class, you first need to instantiate the class. Then, using the object, you can access any member of the class.

Ruby gives you a way to access a method without instantiating a class. Let us see how a class method is declared and accessed:

```
class Accounts
    def reading_charge
    end
    def Accounts.return_date end
end
```

See how the method return_date is declared. It is declared with the class name followed by a period, which is followed by the name of the method. You can access this class method directly as follows:

```
Accounts.return_date
```

To access this method, you need not create objects of the class Accounts.

Ruby *undef* Statement:

This cancels the method definition. An *undef* can not appear in the method body.

By using *undef* and *alias*, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

Syntax:

```
undef method-name
```

Example:

To undefine a method called *bar* do the following:

```
undef bar
```

Ruby File I/O, Directories

Ruby provides a whole set of I/O-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.

The class *IO* provides all the basic methods, such as *read, write, gets, puts, readline, getc,* and *printf*.

The *puts* Statement:

The *puts* statement instructs the program to display the value stored in the variable. This will add a new line at the end of each line it writes.

Example:

```
#!/usr/bin/ruby

val1 = "This is variable one"
val2 = "This is variable two" puts val1
puts val2
```

This will produce the following result: This is variable one This is variable two

The *gets* Statement:

The *gets* statement can be used to take any input from the user from standard screen called STDIN.

Example:

This code will prompt the user to enter a value, which will be stored in a variable val and finally will be printed on STDOUT.

```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
puts val
```

This will produce the following result:

```
Enter a value : This is entered value
```

The *putc* Statement:

Unlike the *puts* statement, which outputs the entire string onto the screen, the *putc* statement can be used to output one character at a time.

Example:

The output of the following code is just the character H:

```
#!/usr/bin/ruby

str="Hello Ruby!"
putc str
```

This will produce the following result:Hello Ruby!

The *print* Statement:

The *print* statement is similar to the *puts* statement. The only difference is that the *puts* statement goes to the next line after printing the contents, whereas with the *print* statement the cursor is positioned on the same line.

Example:

```
#!/usr/bin/ruby

print "Hello World" print
"Good Morning"
```

This will produce the following result: Hello World Good Morning

Opening and Closing Files:

Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.

The *File.new* Method:

You can create a *File* object using *File.new* method for reading, writing, or both, according to the mode string. Finally, you can use *File.close* method to close that file.

Syntax:

```
aFile = File.new("filename", "mode") # ... process the file
aFile.close
```

The *File.open* Method:

You can use *File.open* method to create a new file object and assign that file object to a file. However, there is one difference in between *File.open* and *File.new* methods. The difference is that the *File.open* method can be associated with a block, whereas you cannot do the same using the *File.new* method.

```
File.open("filename", "mode") do |aFile| # ... process the file
end
```

Here is a list of The Different Modes of Opening a File:

Modes	Description
r	Read-only mode. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Read-write mode. The file pointer will be at the beginning of the file.
w	Write-only mode. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Read-write mode. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Write-only mode. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Read and write mode. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Reading and Writing Files:

The same methods that we've been using for 'simple' I/O are available for all file objects. So, gets reads a line from standard input, and *aFile.gets* reads a line from the file object aFile.

The *sysread* Method:

You can use the method *sysread* to read the contents of a file. You can open the file in any of the modes when using the method sysread.

For example :

```
#!/usr/bin/ruby
aFile = File.new("input.txt", "r")
if aFile
    content = aFile.sysread(20)
    puts content
else
    puts "Unable to open file!" end
```

This statement will output the first 20 characters of the file. The file pointer will now be placed at the 21st character in the file.

The *syswrite* Method:

You can use the method syswrite to write the contents into a file. You need to open the file in write mode when using the method syswrite.

For example:

```
#!/usr/bin/ruby
aFile = File.new("input.txt", "r+")
if aFile
    aFile.syswrite("ABCDEF") else
    puts "Unable to open file!" end
```

This statement will write "ABCDEF" into the file.

The *each_byte* Method:

This method belongs to the class *File*. The method *each_byte* is always associated with a block. Consider the following code sample:

```
#!/usr/bin/ruby
aFile = File.new("input.txt", "r+") if aFile
    aFile.syswrite("ABCDEF")
    aFile.each_byte {|ch| putc ch; putc ?. } else
    puts "Unable to open file!" end
```

Characters are passed one by one to the variable ch and then displayed on the screen as follows:

```
s. .a. .s.i.m.p.l.e. .t.e.x.t. .f.i.l.e. .f.o.r. .t.e.s.t.i.n.g.
.p.u.r.p.o.s.e...
.
.
```

The *IO.readlines* Method:

The class *File* is a subclass of the class IO. The class IO also has some methods, which can be used to manipulate files.

One of the IO class methods is *IO.readlines*. This method returns the contents of the file line by line. The following code displays the use of the method *IO.readlines*:

```
#!/usr/bin/ruby
arr = IO.readlines("input.txt")
puts arr[0]
puts arr[1]
```

In this code, the variable arr is an array. Each line of the file *input.txt* will be an element in the array arr. Therefore, arr[0] will contain the first line, whereas arr[1] will contain the second line of the file.

The *IO.foreach* Method:

This method also returns output line by line. The difference between the method *foreach* and the method *readlines* is that the method *foreach* is associated with a block. However, unlike the method *readlines*, the method *foreach* does not return an array.

For example:

```
#!/usr/bin/ruby
IO.foreach("input.txt"){|block| puts block}
```

This code will pass the contents of the file *test* line by line to the variable block, and then the output will be displayed on the screen.

Renaming and Deleting Files:

You can rename and delete files programmatically with Ruby with the *rename* and *delete* methods.

Following is the example to rename an existing file *test1.txt*:

```
#!/usr/bin/ruby
# Rename a file from test1.txt to test2.txt
File.rename( "test1.txt", "test2.txt" )
```

Following is the example to delete an existing file *test2.txt*:

```
#!/usr/bin/ruby
# Delete file test2.txt
File.delete("test2.txt")
```

File Modes and Ownership:

Use the *chmod* method with a mask to change the mode or permissions/access list of a file:

Following is the example to change mode of an existing file *test.txt* to a mask value:

```
#!/usr/bin/ruby
 file = File.new( "test.txt", "w" ) file.chmod( 0755 )
```

Following is the table, which can help you to choose different mask for *chmod* method:

Mask	Description
0700	rwx mask for owner
0400	r for owner
0200	w for owner
0100	x for owner
0070	rwx mask for group
0040	r for group
0020	w for group
0010	x for group
0007	rwx mask for other
0004	r for other
0002	w for other
0001	x for other
4000	Set user ID on execution
2000	Set group ID on execution
1000	Save swapped text, even after use

File Inquiries:

The following command tests whether a file exists before opening it:

```
#!/usr/bin/ruby
File.open("file.rb") if File::exists?( "file.rb" )
```

The following command inquire whether the file is really a file:

```
#!/usr/bin/ruby
# This returns either true or false
File.file?( "text.txt" )
```

The following command finds out if it given file name is a directory:

```
#!/usr/bin/ruby
# a directory
File::directory?( "/usr/local/bin" ) # => true
# a file
File::directory?( "file.rb" ) # => false
```

The following command finds whether the file is readable, writable or executable:

```
#!/usr/bin/ruby
File.readable?( "test.txt" )      # => true
File.writable?( "test.txt" )      # => true
File.executable?( "test.txt" ) # => false
```

The following command finds whether the file has zero size or not:

```
#!/usr/bin/ruby
File.zero?( "test.txt" )          # => true
```

The following command returns size of the file :

```
#!/usr/bin/ruby
File.size?( "text.txt" )          # => 1002
```

The following command can be used to find out a type of file :

```
#!/usr/bin/ruby
File::ftype( "test.txt" )         # => file
```

The ftype method identifies the type of the file by returning one of the following: *file, directory, characterSpecial, blockSpecial, fifo, link, socket, or unknown.*

The following command can be used to find when a file was created, modified, or last accessed :

```
#!/usr/bin/ruby
```

File::ctime("test.txt") # => Fri May 09 10:06:37 -0700 2008
File::mtime("text.txt") # => Fri May 09 10:44:44 -0700 2008
File::atime("text.txt") # => Fri May 09 10:45:01 -0700 2008

Regular-Expression Modifiers

Regular expression literals may include an optional modifier to control various aspects of matching. The modifier is specified after the second slash character, as shown previously and may be represented by one of these characters −

Sr.No.	Modifier & Description
1	**i** Ignores case when matching text.
2	**o** Performs #{} interpolations only once, the first time the regexp literal is evaluated.
3	**x** Ignores whitespace and allows comments in regular expressions.
4	**m** Matches multiple lines, recognizing newlines as normal characters.
5	**u,e,s,n** Interprets the regexp as Unicode (UTF-8), EUC, SJIS, or ASCII. If none of these modifiers is specified, the regular expression is assumed to use the source encoding.

Ruby Regular Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Ruby.

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.

(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses. (?#...) Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular-expression Examples:

Literal characters:

Example	Description
/ruby/	Match "ruby".
¥	Matches Yen sign. Multibyte characters are supported in Ruby 1.9 and Ruby 1.8.

Character classes:

Example	Description
/[Rr]uby/	Match "Ruby" or "ruby"
/rub[ye]/	Match "ruby" or "rube"
/[aeiou]/	Match any one lowercase vowel
/[0-9]/	Match any digit; same as /[0123456789]/
/[a-z]/	Match any lowercase ASCII letter
/[A-Z]/	Match any uppercase ASCII letter
/[a-zA-Z0-9]/	Match any of the above
/[^aeiou]/	Match anything other than a lowercase vowel
/[^0-9]/	Match anything other than a digit

Special Character Classes:

Example	Description
/./	Match any character except newline
/./m	In multiline mode . matches newline, too
/\d/	Match a digit: /[0-9]/
/\D/	Match a nondigit: /[^0-9]/
/\s/	Match a whitespace character: /[\t\r\n\f]/
/\S/	Match nonwhitespace: /[^ \t\r\n\f]/
/\w/	Match a single word character: /[A-Za-z0-9_]/

/\W/	Match a nonword character: /[^A-Za-z0-9_]/

Repetition Cases:

Example	Description
/ruby?/	Match "rub" or "ruby": the y is optional
/ruby*/	Match "rub" plus 0 or more ys
/ruby+/	Match "rub" plus 1 or more ys
/\d{3}/	Match exactly 3 digits
/\d{3,}/	Match 3 or more digits
/\d{3,5}/	Match 3, 4, or 5 digits

Nongreedy repetition:

This matches the smallest number of repetitions:

Example	Description
/<.*>/	Greedy repetition: matches "<ruby>perl>"
/<.*?>/	Nongreedy: matches "<ruby>" in "<ruby>perl>"

Grouping with parentheses:

Example	Description
/\D\d+/	No group: + repeats \d
/(\D\d)+/	Grouped: + repeats \D\d pair
/([Rr]uby(,)?)+/	Match "Ruby", "Ruby, ruby, ruby", etc.

.

Alternatives:

Example	Description
/ruby|rube/	Match "ruby" or "rube"
/rub(y|le))/	Match "ruby" or "ruble"
/ruby(!+|\?)/	"ruby" followed by one or more ! or one ?

Anchors:

This need to specify match position

Example	Description
/^Ruby/	Match "Ruby" at the start of a string or internal line
/Ruby$/	Match "Ruby" at the end of a string or line
/\ARuby/	Match "Ruby" at the start of a string
/Ruby\Z/	Match "Ruby" at the end of a string
/\bRuby\b/	Match "Ruby" at a word boundary
/\brub\B/\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone	
/Ruby(?=!)/	Match "Ruby", if followed by an exclamation point
/Ruby(?!!)/	Match "Ruby", if not followed by an exclamation point

Special syntax with parentheses:

Example	Description
/R(?#comment)/	Matches "R". All the rest is a comment
/R(?i)uby/	Case-insensitive while matching "uby"
/R(?i:uby)/	Same as above
/rub(?:y|le))/	Group only without creating \1 backreference

Search and Replace:

Some of the most important String methods that use regular expressions are **sub** and **gsub**, and their in-place variants **sub!** and **gsub!**.

All of these methods perform a search-and-replace operation using a Regexp pattern. The **sub** & **sub!** replaces the first occurrence of the pattern and **gsub** & **gsub!** replaces all occurrences.

The **sub** and **gsub** returns a new string, leaving the original unmodified where as **sub!** and **gsub!** modify the string on which they are called.

```
#!/usr/bin/ruby
phone = "2004-959-559 #This is Phone Number" # Delete
Ruby-style comments
phone = phone.sub!(/#.*$/, "")
```

```
 puts "Phone Num : #{phone}"
# Remove anything other than digits phone = phone.gsub!(/\D/, "")
 puts "Phone Num : #{phone}"
```

This will produce the following result:

```
 Phone Num : 2004-959-559
 Phone Num : 2004959559
```

Following is another example:

```
 #!/usr/bin/ruby

 text = "rails are rails, really good Ruby on Rails"

 # Change "rails" to "Rails" throughout
 text.gsub!("rails", "Rails")

 # Capitalize the word "Rails" throughout
 text.gsub!(/\brails\b/, "Rails")

 puts "#{text}"
```

This will produce the following result:

```
 Rails are Rails, really good Ruby on Rails
```

Access Control

The Ruby supports three levels of access control for methods are defined as follows:

- **"Public access"** means that the method can be called by any code.
- **"Protected access"** means that only objects of the defining class and its subclasses may call the method.
- **"Private access"** means that the method can only be used by object that defines itself. So, no code can ever call the private methods of another object.

All instance variables has private access by default, and you can not change its access control.

```
class My_class
  def meth1
  ...
  end
  ...
private
  def meth7
  ...
  end
  ...
protected
  def meth11
  ...
  end
  ...
end  # of class My_class
```

Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

Inheritance also provides an opportunity to reuse the code functionality and fast implementation time but unfortunately Ruby does not support multiple levels of inheritances but Ruby supports **mixins**. A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class or superclass**, and the new class is referred to as the **derived class or sub-class**.

Ruby also supports the concept of subclassing, i.e., inheritance and following example explains the concept. The syntax for extending a class is simple. Just add a < character and the name

of the superclass to your class statement. For example, following define a class *BigBox* as a subclass of *Box*:

```
class Box                       # definition of Base
        class def initialize(w,h)  ## constructor method
                @width, @height = w, h
        end
        def getArea             ## instance
                method @width * @height
        end
end
class BigBox < Box              ## definition of subclass
        def printArea           ## add a new instance
                method @area = @width * @height
                puts "Big box area is : #@area"
        end
end


box = BigBox.new(10, 20)        ## create an
object box.printArea()          # print the area
```

When the above code is executed, it produces the following result: Big box area is : 200

RUBY on Rails

What is Rails?

- An extremely productive web-application framework.
- Written in Ruby by David Heinemeier Hansson.
- You could develop a web application at least ten times faster with Rails than you could with a typical Java framework.
- An open source Ruby framework for developing database-backed web applications.
- Configure your code with Database Schema.
- No compilation phase required.

Full Stack Framework

- Includes everything needed to create a database-driven web application, using the Model-View-Controller pattern.
- Being a full-stack framework means all the layers are built to work seamlessly together with less code.
- Requires fewer lines of code than other frameworks.

Convention over Configuration

- Rails shuns configuration files in favor of conventions, reflection, and dynamic runtime extensions.
- Your application code and your running database already contain everything that Rails needs to know!

Rails Strengths

Metaprogramming
Active Record
Convention over configuration
Scaffolding
Built-in testing
Three environments

To develop a web application using Ruby on Rails Framework, you need to install the following software −

- Ruby
- The Rails Framework
- A Web Server
- A Database System

We assume that you already have installed a Web Server and a Database System on your computer. You can use the WEBrick Web Server, which comes with Ruby. Most websites however use Apache or lightTPD web servers in production.

Rails works with many database systems, including MySQL, PostgreSQL, SQLite, Oracle, DB2 and SQL Server. Please refer to a corresponding Database System Setup manual to set up your database.

Rails Installation on Windows

Follow the steps given below for installing Ruby on Rails.

Step 1: Check Ruby Version

First, check if you already have Ruby installed. Open the command prompt and type **ruby -v**. If Ruby responds, and if it shows a version number at or above 2.2.2, then type **gem --version**. If you don't get an error, skip **Install Ruby** step. Otherwise, we'll install a fresh Ruby.

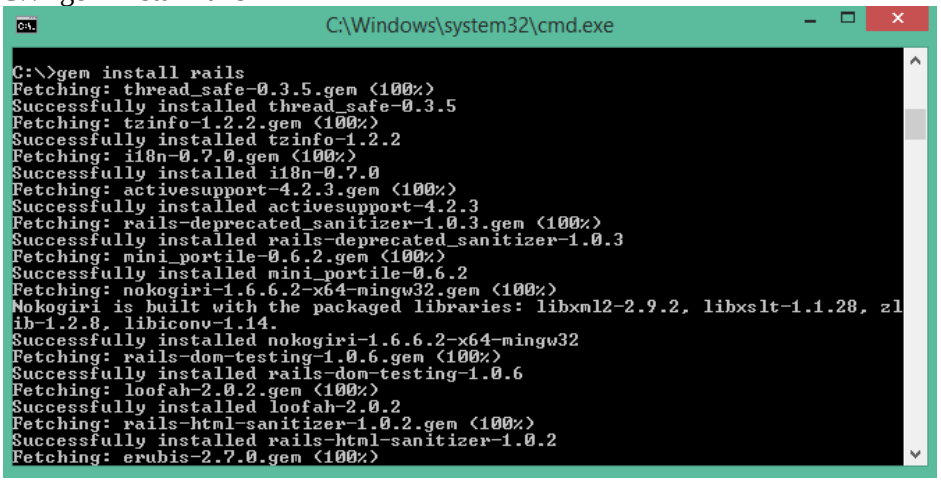
Step 2: Install Ruby

If Ruby is not installed, then download an installation package from rubyinstaller.org. Follow the **download** link, and run the resulting installer. This is an exe file **rubyinstaller-2.2.2.x.exe** and will be installed in a single click. It's a very small package, and you'll get RubyGems as well along with this package. Please check the **Release Notes** for more detail.

Step 3: Install Rails

Install Rails − With Rubygems loaded, you can install all of Rails and its dependencies using the following command through the command line −

C:\> gem install rails



Note − The above command may take some time to install all dependencies. Make sure you are connected to the internet while installing gems dependencies.

Step 4: Check Rails Version

Use the following command to check the rails version.

C:\> rails -v

Output

Rails 4.2.4

Congratulations! You are now on Rails over Windows.

Workflow for Creating Rails Applications

A recommended work flow for creating Rails Application is as follows −

- Use the rails command to create the basic skeleton of the application.
- Create a database on the PostgreSQL server to hold your data.
- Configure the application to know where your database is located and the login credentials for it.
- Create Rails Active Records (Models), because they are the business objects you'll be working with in your controllers.
- Generate Migrations that simplify the creating and maintaining of database tables and columns.
- Write Controller Code to put a life in your application.
- Create Views to present your data through User Interface.

So, let us start with creating our library application.

Creating an Empty Rails Web Application

Rails is both a runtime web application framework and a set of helper scripts that automate many of the things you do when developing a web application. In this step, we will use one such helper script to create the entire directory structure and the initial set of files to start our Library System application.

- Go into ruby installation directory to create your application.
- Run the following command to create a skeleton for library application. It will create the directory structure in the current directory.

tp> rails new library

This will create a subdirectory for the library application containing a complete directory tree of folders and files for an empty Rails application. Check a complete directory structure of the application. Check Rails Directory Structure for more detail.

Most of our development work will be creating and editing files in the **library/app** subdirectories. Here's a quick run down of how to use them −

- The *controllers* subdirectory is where Rails looks to find controller classes. A controller handles a web request from the user.
- The *views* subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user's browser.
- The *models* subdirectory holds the classes that model and wrap the data stored in our application's database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it dead simple.
- The *helpers* subdirectory holds any helper classes used to assist the model, view, and controller classes. This helps to keep the model, view, and controller code small, focused, and uncluttered.

Starting Web Server

Rails web application can run under virtually any web server, but the most convenient way to develop a Rails web application is to use the built-in WEBrick web server. Let's start this web server and then browse to our empty library application −

This server will be started from the application directory as follows. It runs on port number 3000.

tp> cd ruby\library

tp\ruby\library\> Rails server

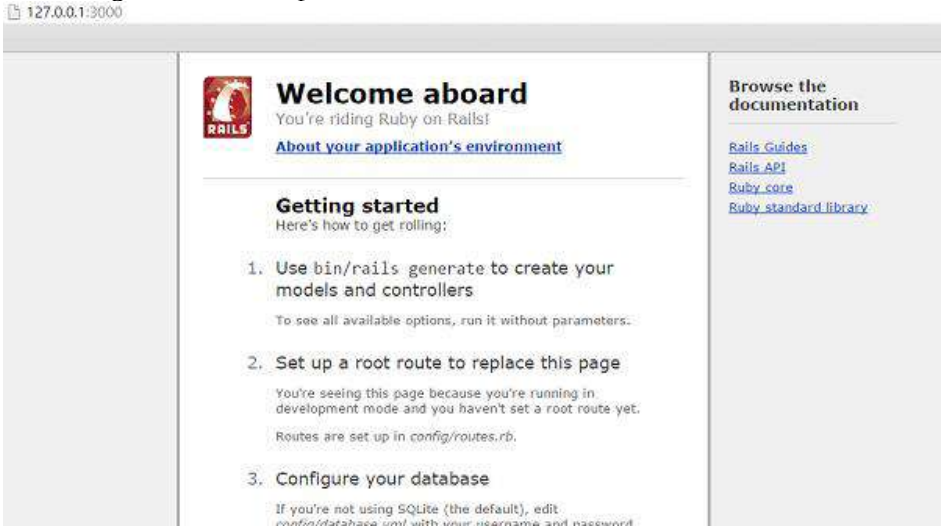
It generates the auto code to start the server as shown below −

```
$ rails server
DL is deprecated, please use Fiddle
=> Booting WEBrick
=> Rails 4.2.3 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
```

This will start your WEBrick web server.

Now open your browser and browse to **http://127.0.0.1:3000**. If everything is gone fine, then you should see a greeting message from WEBrick, otherwise there is something wrong with your setting. If

everything goes well it will generate the output as follows.



Ruby on Rails 5 Hello World Example

We will create a simple Ruby on Rails 5 program displaying Hello World. Ruby on Rails 5 program is quite different from Ruby on Rails 4 program.

Prerequisites

Text Editor: You can use any text editor which is suitable for you. We are using Sublime Text editor which features many plugins.

Browser: We are using Ubuntu default browser, Mozilla Firefox.

Hello World Example

Step 1 Create a directory **jtp** in which all the code will be present and will navigate from the command line.

1. mkdir jtp

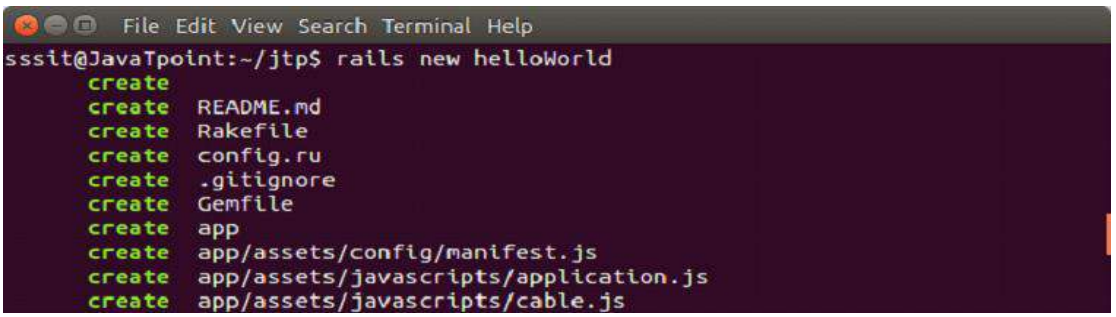
Step 2 Change the directory to jtp

1. cd jtp

Step 3 Create a new application with the name **helloWorld.**

1. rails new helloWorld

You will see something as shown in the below snapshot.



A helloWorld directory will be created in your system. Inside this folder there will be many files and subfolders which is actually the Rails application.

Step 4 Move in to your above created application directory that is helloWorld.

1. cd helloWorld

Step 5 Rails 5 has no longer a static index page in production. There will not be a root page in the production, so we need to create it. First we will create a controller called **hello** for our home page.

1. rails generate controller hello

You will see something as shown in the above snapshot.

Step 6 Now we need to add an index page.

In file app/views/hello/index.html.erb, write

1. <h2>Hello World</h2>
2.
3. Today is 23r March, Thursday.
4.

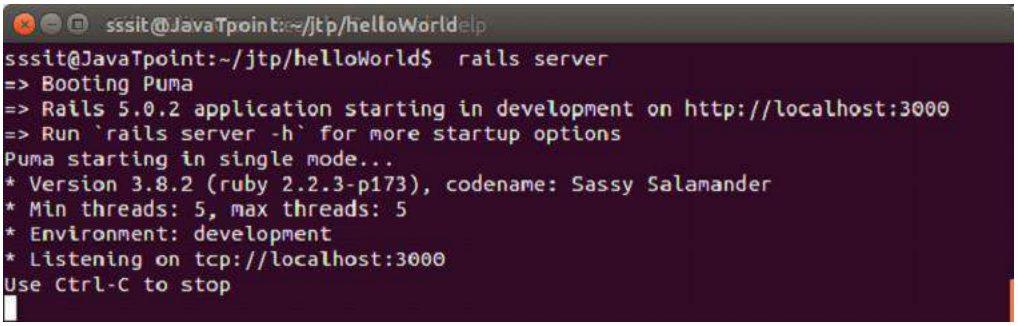
Step 7 Now we need to route the Rails to this action. Edit the config/routes.rb file to set the index page to our new method.

Add the following line in the routes.rb file,

1. root 'hello#index'

Step 8 Now you can verify the page by running your server.

1. rails server



By default, Rails server listens to the port 3000. Although you can change it with the following command.

1. rails server -p portNumber

Step 9 Visit click here in your browser.