

4/4 B.TECH CSE 1ST SEMESTER
WEB TECHNOLOGIES

UNIT-1

S.JAYA PRAKASH
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
SIRCRRCOE, ELURU

UNIT-I: HTML, CSS

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 stands for **Hyper Text Markup Language**, which is the most widely used language on Web to develop web pages. **HTML** was created by Berners-Lee in late 1991 but "HTML 2.0" was the first standard HTML specification which was published in 1995. HTML 4.01 was a major version of HTML and it was published in late 1999. Though HTML 4.01 version is widely used but currently we are having HTML-5 version which is an extension to HTML 4.01, and this version was published in 2012.

Hyper Text Markup Language is the basic structural element that is used to create web pages. HTML is a markup language, which means that it is used to “mark up” the content within a document, in this case a webpage, with structural and semantic information that tells a browser how to display a page. There are three types of code that make up a basic website page. HTML governs the structural elements, CSS styles those elements, and JavaScript enables dynamic interaction between those elements.

HTML structure + CSS style + JS interaction = web page

The key advantages of learning HTML:

- ❖ **Create Web site** - You can create a website or customize an existing web template if you know HTML well.
- ❖ **Become a web designer** - If you want to start a career as a professional web designer, HTML and CSS designing is a must skill.
- ❖ **Learn other languages** - Once you understand the basic of HTML then other related technologies like JAVASCRIPT, PHP are become easier to understand.

BASIC SYNTAX:

The fundamental syntactic units of HTML are called tags. In general, tags are used to specify categories of content. The syntax of a tag is the tag's name surrounded by angle brackets (< and >). Tag names must be written in all lowercase letters. Most tags appear in pairs: an opening tag and a closing tag. The name of a closing tag is the name of its corresponding opening tag with a slash attached to the beginning. For example, if the tag's name is p, the corresponding closing tag is named /p. whatever appears between a tag and its closing tag is the content of the tag.

The opening tag and its closing tag together specify a container for the content they enclose. The container and its content together are called an element.

For example, consider the following element:

```
<p> This is simple stuff. </p>
```

The paragraph tag, <p>, marks the beginning of the content; the </p> tag marks the end of the content of the paragraph element.

STANDARD HTML DOCUMENT STRUCTURE:

A basic HTML page is a document that typically has the file extension .html, though HTML frequently appears in the content of other file types as well. All HTML documents follow the same basic structure so that the browser that renders the file knows what to do. The basic structure on which all web pages are built looks like this:

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>Page Title</title>
```

```
</head>
```

```
<body>
```

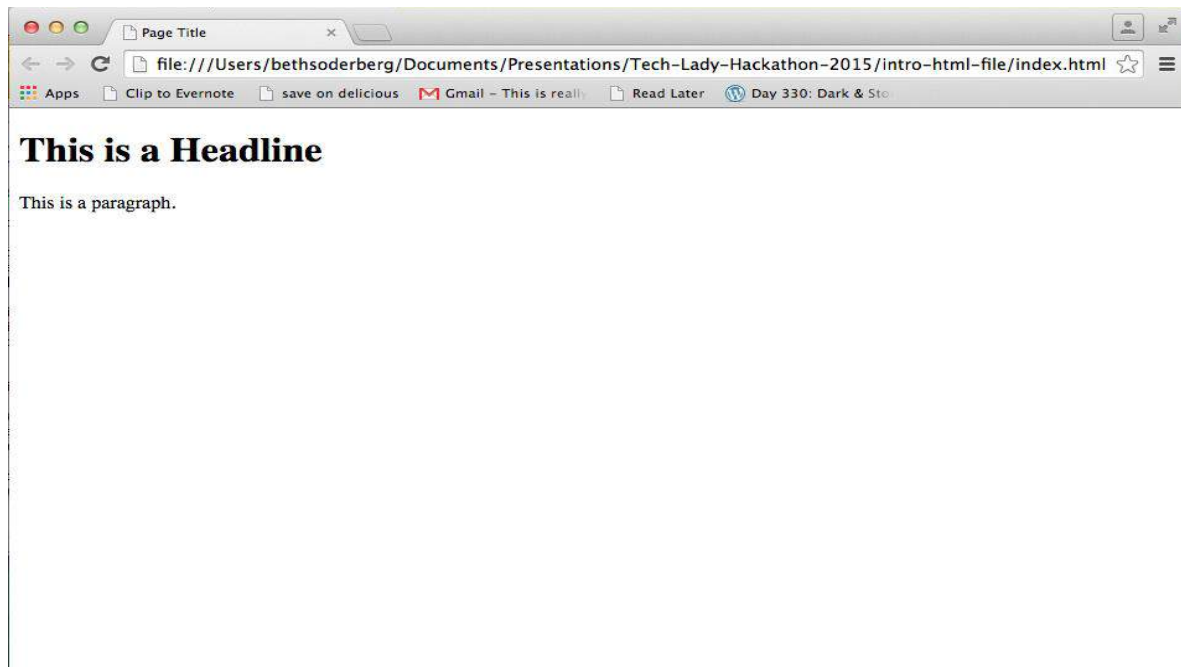
```
<h1>This is a Headline</h1>
```

```
<p>This is a paragraph.</p>
```

```
</body>
```

```
</html>
```

When this code is rendered by a browser, it will look like this:



DOCTYPE:

The first line of code, `<!DOCTYPE html>`, is called a doctype declaration and tells the browser which version of HTML the page is written in. In this case, we're using the doctype that corresponds to HTML5, the most up-to-date version of the HTML language.

HTML ROOT ELEMENT:

Next, the `<html>` element wraps around all of the other code and content in our document. This element, known as the HTML root element, always contains one `<head>` element and one `<body>` element.

HEAD ELEMENT:

The HTML head element is a container that can include a number of HTML elements that are not visible parts of the page rendered by the browser. These elements are either metadata that describe information about the page or are helping pull in external resources like CSS style sheets or JavaScript files.

The `<title>` element is the only element that is required to be contained within the `<head>` tags. The content within this element is displayed as the page title in the tab of the browser and is also what search engines use to identify the title of a page.

All of the HTML elements that can be used inside the `<head>` element are:

- ❖ `<base>`
- ❖ `<link>`
- ❖ `<meta>`
- ❖ `<noscript>`
- ❖ `<script>`
- ❖ `<style>`
- ❖ `<title>`

BODY ELEMENT:

There can only be one <body> element in an HTML document because this element is the container that holds the content of the document. All of the content that you see rendered in the browser is contained within this element. In the example above, the content of the page is a headline and simple paragraph.

BASIC TEXT MARKUP:

The various text markup tags used in the HTML language are listed below.

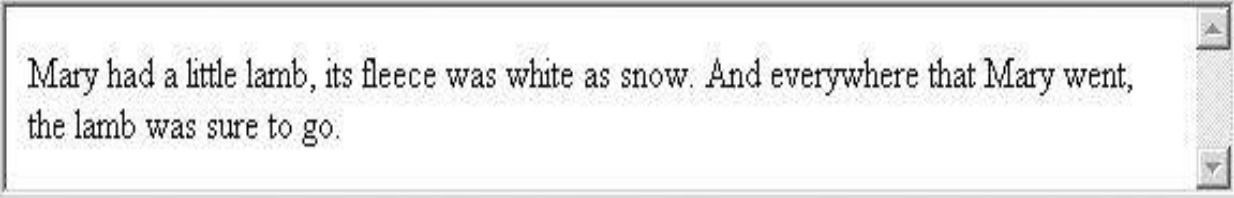
PARAGRAPHS:

The HTML standard does not allow text to be placed directly in a document body. Instead, textual paragraphs appear as the content of a paragraph element, specified with the tag <p>. In displaying the content of a paragraph, the browser puts as many words as will fit on the lines in the browser window. The browser supplies a line break at the end of each line.

Example:

```
<p>
Mary had
a
little lamb, its fleece was white as snow. And
everywhere that
Mary went, the lamb
was sure to go.
</p>
```

Output:



Mary had a little lamb, its fleece was white as snow. And everywhere that Mary went, the lamb was sure to go.


LINE BREAKS:

Sometimes text requires a line break without the preceding blank line. This is exactly what the break tag does. The break tag differs syntactically from the paragraph tag in that it can have no content and therefore has no closing tag (because a closing tag would serve no purpose). The break tag is specified as `
`. The slash indicates that the tag is both an opening and closing tag.

Consider the following markup:

```
<p>  
Mary had a little lamb, <br />  
its fleece was white as snow.  
</p>
```

This markup would be displayed as shown in following figure:



Mary had a little lamb,
its fleece was white as snow.

HEADINGS:

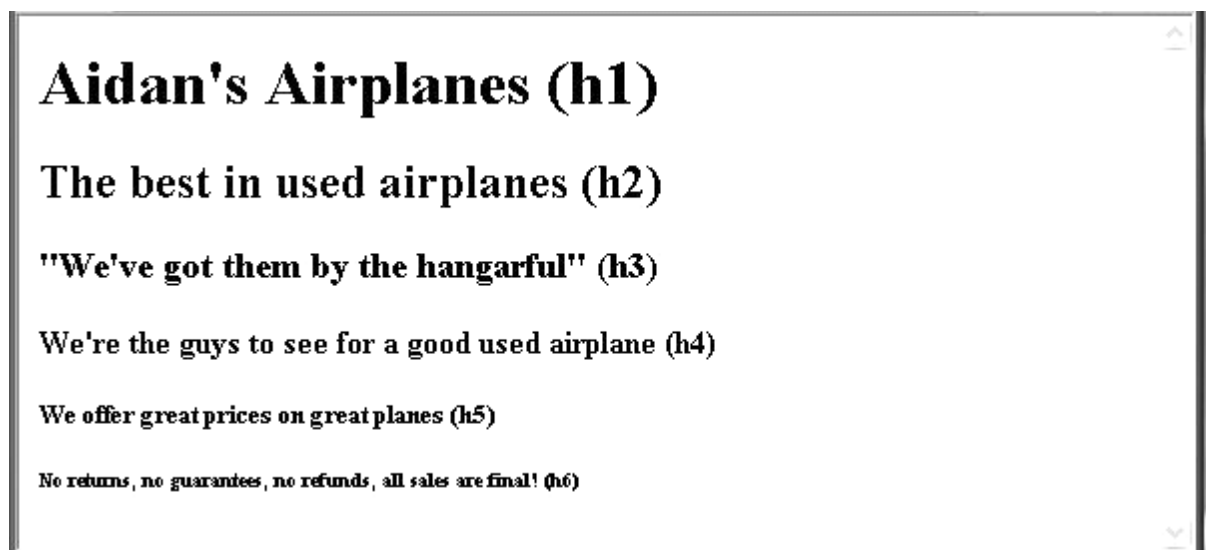
Text is often separated into sections in documents by beginning each section with a heading. Larger sections sometimes have headings that appear more prominent than headings for sections nested inside them. In HTML, there are six levels of headings, specified by the tags `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, and `<h6>`, where `<h1>` specifies the highest-level heading.

Headings are usually displayed in a boldface font whose default size depends on the number in the heading tag. On most browsers, <h1>, <h2>, and <h3> use font sizes that are larger than that of the default size of text, <h4> uses the default size, and <h5> and <h6> use smaller sizes. The heading tags always break the current line, so their content always appears on a new line. Browsers usually insert some vertical space before and after all headings.

Example:

```
<html>
<head> <title> Headings </title>
</head>
<body>
<h1> Aidan's Airplanes (h1) </h1>
<h2> The best in used airplanes (h2) </h2>
<h3> "We've got them by the hangarful" (h3) </h3>
<h4> We're the guys to see for a good used airplane (h4) </h4>
<h5> We offer great prices on great planes (h5) </h5>
<h6> No returns, no guarantees, no refunds,
all sales are final! (h6) </h6>
</body>
</html>
```

OUTPUT:



Aidan's Airplanes (h1)

The best in used airplanes (h2)

"We've got them by the hangarful" (h3)

We're the guys to see for a good used airplane (h4)

We offer great prices on great planes (h5)

No returns, no guarantees, no refunds, all sales are final! (h6)

BLOCK QUOTATIONS:

Sometimes we want a block of text to be set off from the normal flow of text in a document. In many cases, such a block is a long quotation. The <blockquote> tag is designed for this situation. Browser designers determine how the content of <blockquote> can be made to look different from the surrounding text. In many cases, the block of text is indented, either on the left or right side or both. Another possibility is that the block is set in italics.

Example:

```
<html>
```

```
<head> <title> Blockquotes </title>
```

```
</head>
```

```
<body>
```

```
<p>
```

Abraham Lincoln is generally regarded as one of the greatest presidents of the United States. His most famous speech was delivered in Gettysburg, Pennsylvania, during the Civil War.

This speech began with

```
</p>
```

```
<blockquote>
```

```
<p>
```

"Fourscore and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

```
</p>
```

```
<p>
```

Now we are engaged in a great civil war, testing whether

that nation or any nation so conceived and so dedicated,
can long endure."

</p>

</blockquote>

<p>

Whatever one's opinion of Lincoln, no one can deny the
enormous and lasting effect he had on the United States.

</p>

</body>

</html>

OUTPUT:

Abraham Lincoln is generally regarded as one of the greatest presidents of the United States. His most famous speech was delivered in Gettysburg, Pennsylvania, during the Civil War. This speech began with

"Fourscore and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated, can long endure."

Whatever one's opinion of Lincoln, no one can deny the enormous and lasting effect he had on the United States.

FONT STYLES AND SIZES:

Early Web designers used a collection of tags to set font styles and sizes. For example, `<i>` specified italics and `` specified bold. Since the advent of cascading style sheets, the use of these tags has become passé. There are a few tags for fonts that are still in widespread use, called content-based style tags. These tags are called content based because they indicates the particular kind of text that appears in their content.

Three of the most commonly used content-based tags are the emphasis tag, the strong tag, and the code tag. The emphasis tag, ``, specifies that its textual content is special and should be displayed in some way that indicates this distinctiveness. Most browsers use italics for such content. The strong tag, `` is like the emphasis tag, but more so. Browsers often set the content of strong elements in bold. The code tag, `<code>`, is used to specify a monospace font, usually for program code.

For example, consider the following element:

```
<code> cost = quantity * price </code>
```

OUTPUT:



```
cost = quantity * price
```

Subscript and superscript characters can be specified by the `<sub>` and `<sup>` tags, respectively. These are not content-based tags.

For example,

```
X<sub>2</sub><sup>3</sup> + y<sub>1</sub><sup>2</sup>
```

OUTPUT:


$$x_2^3 + y_1^2$$

Character-modifying tags are not affected by `<blockquote>`, except when there is a conflict. For example, if the text content of `<blockquote>` is set in italics and a part of that text is made the content of an `` tag, the `` tag would have no effect.

HTML tags are categorized as being either block or inline. The content of an inline tag appears on the current line. So, an inline tag does not implicitly include a line break. One exception is `br`, which is an inline tag, but its entire purpose is to insert a line break in the content.

A block tag breaks the current line so that its content appears on a new line. The heading and block quote tags are block tags, whereas `` and `` are inline tags. In HTML, block tags cannot appear in the content of inline tags.

Therefore, a block tag can never be nested directly in an inline tag. Also, inline tags and text cannot be directly nested in body or form elements. Only block tags can be so nested. That is why the example `greet.html` has the text content of its body nested in a paragraph element.

CHARACTER ENTITIES:

HTML provides a collection of special characters that are sometimes needed in a document but cannot be typed as themselves. In some cases, these characters are used in HTML in some special way—for example, `>`, `<`, and `&`. In other cases, the characters do not appear on keyboards, such as the small raised circle that represents “degrees” in a reference to temperature.

Finally, there is the non breaking space, which browsers regard as a hard space—they do not squeeze them out, as they do other multiple spaces. These special characters are defined as entities, which are codes for the characters. An entity in a document is replaced by its associated character by the browser.

Some of the most commonly used entities are:

| Character | Entity Meaning | Meaning |
|-----------|----------------|---------------------------|
| & | & | Ampersand |
| < | < | Is less than |
| > | > | Is greater than |
| " | " | Double quote |
| ' | ' | Single quote (apostrophe) |
| ¼ | ¼ | One-quarter |
| ½ | ½ | One-half |
| ¾ | ¾ | Three-quarters |
| ° | ° | Degree |
| (space) | | Non breaking space |

HORIZONTAL RULES:

The parts of a document can be separated from each other, making the document easier to read, by placing horizontal lines between them. Such lines are called *horizontal rules*, and the block tag that creates them is `<hr />`. The `<hr />` tag causes a line break (ending the current line) and places a line across the screen. The browser chooses the thickness, length, and horizontal placement of the line. Typically, browsers display lines that are three pixels thick.

Note again the slash in the `<hr />` tag, indicating that this tag has no content and no closing tag.

The meta ELEMENT:

The meta element is used to provide additional information about a document. The meta tag has no content; rather, all of the information provided is specified with attributes. The two attributes that are used to provide information are name and content.

The user makes up a name as the value of the name attribute and specifies information through the content attribute. One commonly chosen name is keywords; the values of the content attribute associated with the keywords are those which the author of a document believes characterizes his or her document.

An example is

```
<meta name = "keywords" content = "binary trees, linked lists, stacks" />
```

Web search engines use the information provided with the meta element to categorize Web documents in their indices. So, if the author of a document seeks widespread exposure for the document, one or more meta elements are included to ensure that it will be found by Web search engines.

For example, if an entire book were published as a Web document, it might have the following meta elements:

```
<meta name = "Title" content = "Don Quixote" />
```

```
<meta name = "Author" content = "Miguel Cervantes" />
```

```
<meta name = "keywords" content = "novel, Spanish literature, groundbreaking work" />
```

IMAGES:

The inclusion of images in a document can dramatically enhance its appearance. So as a web developer we should have clear understanding on how to use images in the web pages.

The 3 most popular image formats are:

- ❖ GIF(Graphic Interchange Format)
- ❖ JPEG(Joint Photographic Experts Group)
- ❖ PNG(Portable Network Graphics)

Indicated with .gif, .jpg, .png file extensions.

The images are included in the body of the web document using tag.

Syntax:

```

```

| Attributes | Description |
|---------------------------|---|
| src | Provides a path for the picture to be inserted into the web page |
| alt | Displays a text if the picture is not displayed on the web browser |
| align | Specifies the position of the image on the web page |
| border | Specifies the width of the border around the image |
| width | Specifies how wide the picture should appear on the web page |
| height | Specifies how high the picture should appear on the web page |
| hspace [horizontal space] | Specifies the amount of white space to the left and right side of a image |
| vspace [vertical space] | Specifies the amount of white space above and below the image |

HYPER LINKS:

- ❖ A link is a pointer to some resource.
- ❖ Web pages can contain links that take us directly to other pages and even specific parts of a given web page.
- ❖ These links are known as hyperlinks.

Hyperlinks allow visitors to navigate between web sites by clicking on words, phrases and images.

Web links have 2 basic components:

- ❖ A link: The tag in the main document (source) that refers to another document.
- ❖ A target: The document (or particular location in the document) to which the link points to.

The tags used to produce links are the <a> and , called as anchor tag. The anchor tag consists of two pieces of information:

1. The URL of the target document
2. The text needed to activate the hyperlink

Syntax:

```
<a href="URL"> click here </a>
```

The href (Hypertext reference) attribute is used to specify the target of the link.

Types of Links:

There are 3 different kinds of links we can have in the web page:

1. INTERNAL: Link to a section on the current (same) page or document.
2. LOCAL: Link to a page on the same server or directory.
3. EXTERNAL: Link to a page or website on a different server or directory.

Link targets:

How link to another website to open in a new window?

Usually links will open in the current window. We can link to another website to open in a new window by using the “target” attribute and setting the value to “_blank” with the anchor tag.

Example:

```
<a href="http://www.google.co.in" target="_blank" > Click here </a>
```

Other predefined values for target attribute are:

- _self: Loads the page into the current window.
- _parent: Loads the page into the frame that is superior to the window the hyperlink is in.
- _top: cancels all windows and loads in full browser window.

The default value is _self.

LINK WITHIN A PAGE (NAMED ANCHORS / INTERNAL LINKS)

How to create a hyperlink to a different part of the same page?

To create a link to a different part of the same page the name attribute of the <a> tag is used.

It is a two step process:

1. Create a link pointing to the anchor.
click here to read the UNIT
2. Create the anchor itself.

Note: # (pound sign) indicates that the link is pointing to anchor on a page. Anchor meaning a specific place in the middle of the page.

LISTS:

We frequently make and use lists in daily life—for example, to-do lists and grocery lists. Likewise, both printed and displayed information is littered with lists. HTML provides simple and effective ways to specify lists in documents. The primary list types supported are those with which most people are already familiar: unordered lists such as grocery lists and ordered lists such as the assembly instructions for a new bookshelf. Definition lists can also be defined.

Unordered Lists:

- ❖ An unordered list is used to list items where ordering is not required.
- ❖ Therefore unordered lists are bulleted.
- ❖ Unordered Lists are specified with `` **and** `` tags.
- ❖ The bullets used for indentation can be a filled circle / square /empty circle.
- ❖ By default the bullet style is **disc** (filled circle).
- ❖ The actual contents of the lists are specified with `` **and** `` tags.
- ❖ Both `` and `` tags can use the type attribute to specify Bullet style.

Syntax:

```
<ul type="circle /disc /square" >
```

```
<li>list item </li>
```

```
</ul>
```

Ordered lists:

1. Ordered list is used to list items where ordering is very important.
2. Ordered lists are preceded by numbers or letters.
3. Numbering can be done using:
 - Arabic numbers (1,2,3....)
 - Letters (A-Z and a-z)
 - Roman Numerals (i,ii,iii,iv,..... and I ,II,III,IV,V....)
4. The tags used to specify an ordered lists are and tags.
5. Ordered lists also uses and to list the items.

Syntax:

```
<ol type="1 /a/A/i/I " start="x" >
```

```
<li>list item</li>
```

```
</ol>
```

| | |
|-----------|----------------------------------|
| type="1" | Arabic Numbers |
| type="a" | Lower alpha |
| type="A" | Upper alpha |
| type="i" | lower roman |
| type="I" | upper roman |
| start="x" | The beginning of the list number |

Definition lists:

- ✓ Definition lists consists of terms followed by its definitions.
- ✓ Definition lists are specified with <dl> and </dl> tags. And requires two more special tags first is for data term <dt> and </dt> and the other is for data definition <dd> and </dd> tags.

Syntax:

```
<dl>
```

```
<dt>B.Tech </dt>
```

```
<dd> Bachelor of Technology</dd>
```

```
</dl>
```

TABLES:

Tables are common fixtures in printed documents, books, and, of course, Web documents. Tables provide a highly effective way of presenting many kinds of information.

A table is a matrix of cells. The cells in the top row often contain column labels, those in the leftmost column often contain row labels, and most of the rest of the cells contain the data of the table. The content of a cell can be almost any document element, including text, a heading, a horizontal rule, an image, and a nested table.

Tables are defined with the <table> and </table> tag.

Syntax:

```
<table border="n" bgcolor="xxx" bordercolor="yy" align="left / right /center /top  
/bottom" width="x / x%" cellpadding="x" cellspacing="x" >  
.....  
</table>
```

| Attributes | Description |
|-------------------|--|
| cellpadding | Specifies the amount of space held between the contents of the cell and the cell border ,specified in pixels |
| cellspacing | Specifies the amount of space held between cells, specified in pixels |
| align | Specifies the position of the table on the web page |
| border | Specifies the width of the border around the table |
| width | Specifies the width for the entire table |
| bgcolor | Specifies the background color for the table |
| bordercolor | Specifies the color for the table border |

- ❖ To add rows to the table we use <tr> **and** </tr> tags.
- ❖ To divide the rows into columns we use <td> **and** </td> tags (i.e., The contents of the cell is specified with <td> and </td> tag).

Syntax:

```
<td colspan='n' rowspan='n' align="left /right /center" valign="top/bottom/middle  
/center/baseline" width="x" height="x" >  
</td>
```

| Attributes | Description |
|------------|---|
| colspan | It is used to stretch the cell to the right to span over the subsequent columns |
| rowspan | It is used to stretch the cell to span downwards over several rows |
| align | Specifies the horizontal alignment of the text within the table cell |
| valign | Specifies the vertical alignment of the text within the table cell |
| width | Specifies the width for the table cell |
| height | Specifies the height of the table cell |

- ❖ The column headings are specified with **<th>** and **</th>** tags.
- ❖ Normally **<th>** tag is used in the top row of the table to replace the **<td>** tag.
- ❖ All the attributes used the **<td>** cell can also be used in the **<th>** tag also.
- ❖ We can specify the title to the table using **<caption>** and **</caption>** tags.

Syntax:

`<caption align="top/bottom/left/right">Table title</caption>`

The align attribute specifies the position of the table title on the web page.

Example:

```
<html>

<head> <title> A simple table </title>

</head>

<body>

<table border = "border">

<caption> Fruit Juice Drinks </caption>

<tr>

<th> </th>

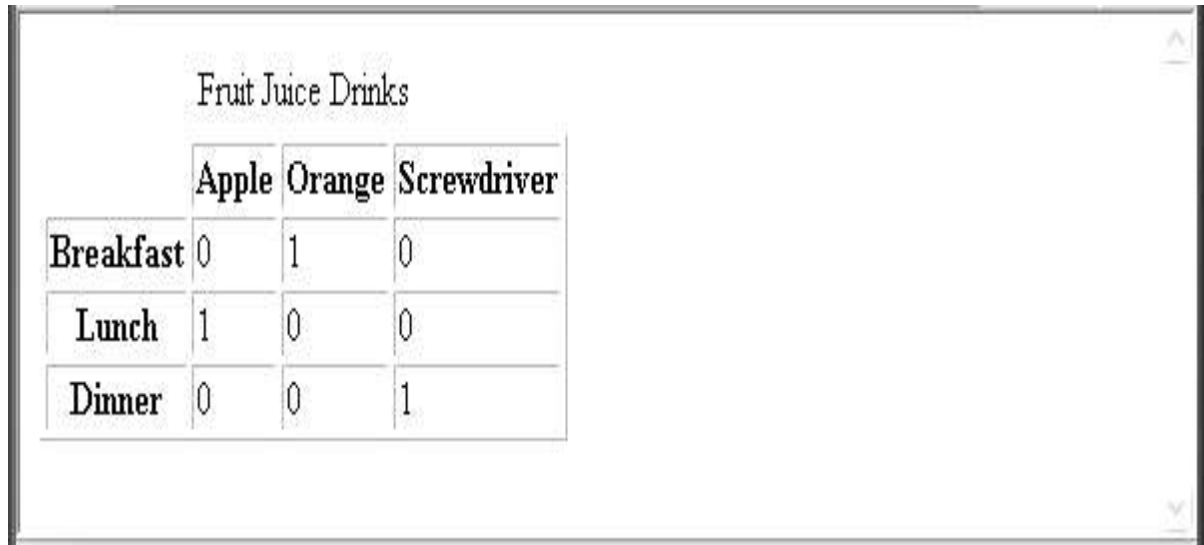
<th> Apple </th>

<th> Orange </th>

<th> Screwdriver </th>
```

```
</tr>
<tr>
<th> Breakfast </th>
<td> 0 </td>
<td> 1 </td>
<td> 0 </td>
</tr>
<tr>
<th> Lunch </th>
<td> 1 </td>
<td> 0 </td>
<td> 0 </td>
</tr>
<tr>
<th> Dinner </th>
<td> 0 </td>
<td> 0 </td>
<td> 1 </td>
</tr>
</table>
</body>
</html>
```

OUTPUT:



The screenshot shows a web browser window with a table titled "Fruit Juice Drinks". The table has three columns: "Apple", "Orange", and "Screwdriver". The rows represent different meals: "Breakfast", "Lunch", and "Dinner". The values in the cells are 0 or 1, indicating the presence or absence of each drink.

| | Apple | Orange | Screwdriver |
|-----------|-------|--------|-------------|
| Breakfast | 0 | 1 | 0 |
| Lunch | 1 | 0 | 0 |
| Dinner | 0 | 0 | 1 |

FORMS:

The most common way for a user to communicate information from a Web browser to the server is through a form. Modeled on the paper forms that people frequently are required to fill out, forms can be described in HTML and displayed by the browser. HTML provides tags to generate the commonly used objects on a screen form. These objects are called *controls* or *widgets*.

There are controls for single-line and multiple-line text collection, checkboxes, radio buttons, and menus, among others. All control tags are inline tags. Most controls are used to gather information from the user in the form of either text or button selections.

Each control can have a value, usually given through user input. Together, the values of all of the controls (that have values) in a form are called the *form data*. Every form requires a *Submit* button. When the user clicks the *Submit* button, the form data is encoded and sent to the Web server for processing.

Syntax:

```
<form name="form1" action="form_action.asp" method="get" >  
...  
...  
...  
</form>
```

| Attributes | Description |
|------------|--|
| name | Used to give name to the form. |
| action | Tells the browser where to send the form content when we submit the forms. |
| method | Tells the browser how to send the form content when we submit the form. |

Form data can be sent to the server in two ways, each corresponding to HTTP methods.

- ❖ The **GET** method, which sends data as part of the URL.
- ❖ The **POST method**, which hides data in something known as the HTTP header.

What the difference is between get and post methods?

- ❖ The get method is not appropriate for forms with personal or financial information because it sends data as part of the URL.
- ❖ When the method attribute is set to POST, the browser sends a separate server request containing some special headers followed by the data. So that only the server sees the content of this request.
- ❖ Thus it is the best method form sending secure information such as credit card or other personal information.

The <input> Tag:

Many of the commonly used controls are specified with the inline tag <input>, including those for text, passwords, checkboxes, radio buttons, and the action buttons *Reset*, *Submit*, and *plain*.

Syntax:

```
<input type="text/password/radio/checkbox/submit/reset" name="XXX" size="n"  
value="nnn" checked="checked"/>
```

| Attributes | Description |
|------------|--|
| type | Specifies the particular kind of control |
| name | Specifies the name of an <input> element |
| value | Specifies the value of the <input> element |
| size | Specifies default size of the control (no of characters to be entered). |
| checked | Specifies that an <input> element should be preselected when the page loads (for type="checkbox" or type="radio") |

Textboxes: Used to collect online input from the user.

Example: `<input type="text" name="fname" " value="txt " size="30" />`

Passwords: It is a text box whose content is never displayed by the browser.

Example: `<input type=" password " name="fname" " value="pwd " size="30" />`

Checkboxes: One or more buttons used by the user to select one or more elements from a list.

Example: `<input type=" checkbox" name="fname" size="30" " value="check" checked="checked"/>`

Radio buttons:

One or more buttons used by the user to select only one or more elements from a list. But only one button can be on at a time.

Example: `<input type=" radio" name="fname" size="30" " value="radio" checked="checked" />`

File Upload:

The File Upload object represents a single-line text input control and a browse button, in an HTML forms.

This object allows file uploading to a server. Clicking on the browse button opens the file dialog box that allows a user to select a file to upload.

Example: `<input type="file" value="fileupload"/>`

Button:

This allows buttons to be included in the form that perform task other than submitting or resetting.

Example: `<input type="button" name="fname" value="button"/>`

Image Button:

Image buttons have the same effect as submit buttons. When a visitor clicks an image button the form is sent to the address specified in the action setting of the `<form>` tag.

Example: `<input type="image" src="rainbow.gif" name="image" width="60" height="60"/>`

Hidden control:

Hidden fields are similar to text fields, but it does not show on the page.

Example: `<input type="hidden" />`

Submit: This creates a button that automatically submits a form.

Example: `<input type="submit" name="fname" value="Submit"/>`

Reset: This creates a button that automatically resets a form.

Example: `<input type="reset" name="fname" value="Reset"/>`

The `<textarea>` Control:

A text area control created with the `<textarea>` tag creates a multiple line text gathering box, with implicit scrolling in both the directions.

Syntax:

`<textarea rows="n" cols="n">`

.....

`</textarea>`

- ❖ rows: specifies the number of lines of text the area should display.
- ❖ cols: specifies the width of the text area measured in no. of characters.

The <select> tag:

- ❖ Menus/ drop down lists are used to allow the user to select items from a list when the list is too long to use check boxes or radio buttons.
- ❖ Drop down lists are created using <select> tag.
- ❖ Each of the items in a menu is specified with the <option> tag within the <select> element.

Example:

```
<select name="s1" size="10">  
<option selected="selected"> INDIAN </option>  
<option> AMERICAN </option>  
<option> AUSTRALIAN </option>  
<option> TIBETIAN </option>  
<option> JAPANESE </option>  
</select>
```

- ❖ **Selected:** specifies that the item is pre-selected.
- ❖ **Size:** specifies the number of items that are to be displayed.
- ❖ **Name:** specifies the name of the menu.

The <label> tag :

It is used to specify label to Form controls.

Example:

```
<label>  
FILE UPLOAD :< input type="file" value="fileupload"/>  
</label>
```

Cascading Style Sheets

What is CSS?

CSS is a language that applies styles to a HTML document and its elements to change the look and feel and is usually stored in separate .css style which can be re-used for all the web pages. A website is made up of HTML for content plus CSS for appearance.

| | | | | |
|-------------------|---|-----------------------|---|----------|
| HTML (Content) | + | CSS (Presentation) | = | WEB PAGE |
|-------------------|---|-----------------------|---|----------|

Advantages:

- **CSS saves time** - You can write CSS once and then reuse same sheet in multiple HTML pages. You can define a style for each HTML element and apply it to as many Web pages as you want.
- **Pages load faster** - If you are using CSS, you do not need to write HTML tag attributes every time. Just write one CSS rule of a tag and apply to all the occurrences of that tag. So less code means faster download times.
- **Easy maintenance** - To make a global change, simply change the style, and all elements in all the web pages will be updated automatically.
- **Superior styles to HTML** - CSS has a much wider array of attributes than HTML so you can give far better look to your HTML page in comparison of HTML attributes.
- **Multiple Device Compatibility** - Style sheets allow content to be optimized for more than one type of device. By using the same HTML document, different versions of a website can be presented for handheld devices such as PDAs and cell phones or for printing.
- **Global web standards** - Now HTML attributes are being deprecated and it is being recommended to use CSS. So its a good idea to start using CSS in all the HTML pages to make them compatible to future browsers.

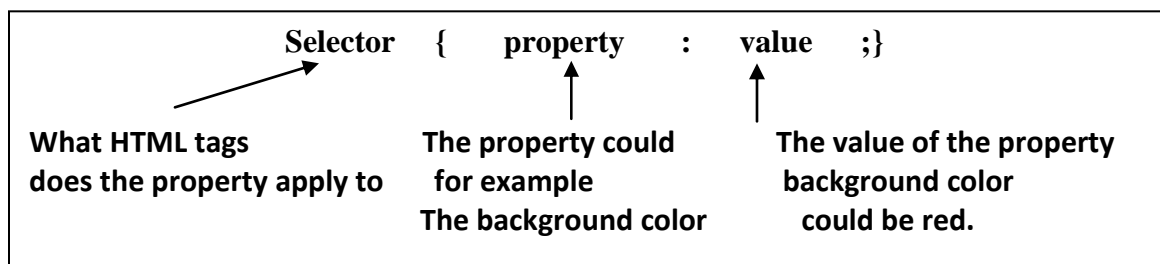
Disadvantages:

- **Browser Compatibility:** Some style sheet features are supported and some are not by the browsers.

Basic CSS Syntax

The CSS syntax consists of a set of rules.

These rules have a 3 parts:



- **Selector:** A selector is an HTML tag at which style will be applied. This could be any tag like <h1> or <table> etc.
- **Property:** A property is a type of attribute of HTML tag. Put simply, all the HTML attributes are converted into CSS properties. They could be color or border etc.
- **Value:** Values are assigned to properties. For example color property can have value either red.

Example: You can define a table border as follows:

```
table{ border :1px; }
```

Here **table** is a selector and **border** is a property and given value **1px** is the value of that property.

LEVELS OF STYLESHEET (CSS Inclusion)

There are 3 levels of style sheet used associate css styles with your HTML document.

- 1) Inline
- 2)Document
- 3)External

1) Inline CSS - The style Attribute:

- ✓ Inline style sheet rules will be applied to the content of a single element.
- ✓ Inline style sheet rules are specified as the values of the style attribute.
- ✓ Here is the generic syntax:

```
<element style="...style rules...">
```

Style rule: is a list of property: value pairs

Example:

Following is the example of inline CSS based on above syntax:

```
<h1 style ="color: red;"> This is inline CSS </h1>
```

2) Document Level CSS - The <style> Element:

- ✓ Document level style sheet rules will be applied to all the elements available in the document.
- ✓ Document level style sheet rules are specified as the content of <style> element.
- ✓ This tag is placed inside <head>...</head> tags.
- ✓ Here is the generic syntax:

```
<head>  
<style type="text/css" media="projection/screen/tv/print">  
Style Rules  
.....  
</style>  
</head>
```

Where

type="text/css" specifies the style sheet language as a content type. This is required attribute.
media=" screen/tty/tv/projection/handheld/print/all"

Specifies the device the document will be displayed on. Default value is all. This is optional attribute.

3) External CSS - The <link> Element:

- ✓ An external style sheet is a separate text file with .css extension.
- ✓ You define all the Style rules within this text file and then you can include this file in any HTML document using <link> element.
- ✓ External style sheets can apply to the bodies of multiple html documents.
- ✓ Here is the generic syntax of including external CSS file:

```
<head>
<link rel="stylesheet" type="text/css" href="..." />
</head>
```

Where

rel="stylesheet" specifies the relationship of the linking document to the current document.
type="text/css" Specifies the style sheet language as a content-type (MIME type).

This attribute is required.

href="filename.css" Specifies the style sheet file having Style rules.

This attribute is a required.

Example:

Consider a simple style sheet file with a name mystyle.css having the following rules:

```
h1 {color: blue; font-size: 34pt; }
```

Now you can include this file mystyle.css in any HTML document as follows:

```
<head>
<link rel="stylesheet" type="text/css" href="mystyle.css" />
</head>
```

TYPES OF SELECTORS

In CSS, a selector is the target element to which each CSS rule is applied.

The different types of selectors are:

- Type selectors
- Class selectors
- ID selectors
- Descendant selectors
- Child selectors
- Universal selectors
- Adjacent sibling selectors
- Attribute selectors
- Pseudo-classes and Pseudo-elements

1) The Type Selectors:

- ✓ It is a single element name.
- ✓ In this case, the property values in the rule apply to all occurrences of the named elements.

Ex:

```
h1 {font-size:24pt;}
```

2) The Universal Selectors:

The universal selector, denoted by an asterisk (*), applies its style to all elements in the document.

Ex:

```
*{color: red}
```

Makes all elements in the document red.

3) The Descendant Selectors:

Descendant selectors are used to select elements that are descendants of another element in the document tree.

Ex:

```
p em {color: blue;}
```

4) The Class Selectors:

- ✓ Class selectors are used to allow different occurrences of the same tag to use different style specifications.
- ✓ Class selectors are used to select any XHTML element that has a class attribute.
- ✓ A class selector is indicated by a period(.).

Ex:

```
<head>  
  p.normal{color:blue;}  
</head>  
.....  
<body>  
<p class="normal">text</p>  
</body>
```

5) The ID Selectors:

- ✓ ID selector is an individually identified (named) selector to which a specific style is declared.
- ✓ ID selectors are created by a character # followed by the selector's name.

Ex:

```
#mypara{color:green;}  
.....  
<p id="mypara">This is Id selector</p>
```

6) The Child Selectors:

A child selector is used to select an element that is a direct child of another element.

Ex:

```
P>em{font-size:50pt;}
```

7) The Attribute Selectors:

Attribute selectors are used to select elements based on their attribute or attribute values.

Ex:

```
img[src="small.gif"]{border-width:1px; border-style:dotted;}
```

8) Adjacent sibling selectors:

Adjacent sibling selectors will select the siblings immediately following an element.

Ex:

```
em+strong{font-style:italic;}
```

9) Pseudo classes:

- ✓ Pseudo classes are used to add special effects to some selectors.
- ✓ Pseudo class styles apply when something happens rather than because the target element simply exists.

Syntax:

```
Selector: pseudo-class {property: value ;}
```

The commonly used pseudo classes are:

: first-child =add special style to the first child of another element

:link =add special style to the normal link

:visited=add special style to the visited link

:active=add special style to the active link

hover: =apply styles when mouse points over the element

focus: =apply styles when element accept input from the keyboard

Ex:

```
a:visited{color:red;}
```

```
a:active{color:green;}
```

```
a:link{color:blue;}
```

```
input:hover{color:red;}
```

```
input:focus{color:green;}
```

CSS UNITS OF MEASUREMENTS (PROPERTY VALUE FORMS)

- Every CSS property has a value and value can include some measurements like %,cm,mm etc.,
- **Absolute size:**
Sets the text to a specified size
Ex: inches, centimeters, points, and so on,
- **Relative size:**
Sets the size relative to surrounding elements
Ex: percentages and em units.

| Unit | Description | Example |
|------|---|----------------------|
| em | 1em is equal to the current font size , 2em means 2 times the size of the current font Ex: if an element is displayed with a font size 12pt, then 2em is 24pt. | P{font-size:1.5em;} |
| ex | x-height (height of the lower case letter) | P{font-size:2ex;} |
| px | pixel(a dot on computer screen) | P{font-size:12px;} |
| in | Define measurements in inches | P{font-size:0.15in;} |
| cm | Define measurements in centimeters | P{font-size:0.8cm;} |
| mm | Define measurements in millimeters | P{font-size:4mm;} |
| pc | Picas(1pica=12points ,1inch=6picas) | P{font-size:1pc;} |
| pt | Points(1point=1/72 inches) | P{font-size:12pt;} |
| % | Percentage units are relative to another values 100% takes the current value, 200% doubles it and so on. | P{font-size:200%;} |

Color values:

| | | |
|-----------------------------|---|------------------------------------|
| #rrggbb | Standard XHTML red-green-blue 6-digit hexadecimal number. Each hexadecimal code will be preceded by pound sign(#) | P{color:#FFFF00;} [yellow] |
| #rgb | This is a shorter form of the six-digit notation. In this format ,each digit is replicated to arrive at an equivalent six-digit value; Ex: #F00 becomes FF0000 | P{color:#F00;} [red] |
| rgb(rrr%, ggg%,bbb %) | Percentage representation ,each value being a percentage between 0.0% and 100.0% | P{color:rgb(40%,70%,4%;)} |
| Rgb(rrr,ggg,bbb) | Decimal representation ,each value being an integer between 0 and 255 | P{ color:rgb (0,0,255);} [blue] |
| Aqua,red,black | Using color names | P{color:orange;} |

FONT PROPERTIES:

CSS font properties define the font family, boldness, size, and the style of a text.

Font Family


Font family is the type of font used for rendering text similar to the fonts used we select in MS-WORD.

To change the fonts we can use the font-family property with font names as comma separated values.

```
EX: h1 {font-family:Arial;}  
    h2 {font-family:Arial,Verdana,"Times New Roman";}
```

In CSS, there are two types of font family names:

- **generic family** - a group of font families with a similar look (like "Serif" or "Monospace")
- **font family** - a specific font family (like "Times New Roman" or "Arial")

| Generic family | Description | Examples |
|----------------|--|---|
| Serif | Serif fonts have small lines at the ends on some characters | Times New Roman,Georgia |
| Sans-serif | "Sans" means without - these fonts do not have the lines at the ends of characters | Arial,Arial Black,Trebuchet MS,Helvetica,Geneva,Verdana |
| Monospace | All monospace characters have the same width | Courier New,Andale Mono, Lucida Console |
| Cursive | Cursive scripts emulates handwritten appearance | Apple chancery,,comic Sans,and Zapf chancery, ex:  |
| fantasy | Fantasy fonts are purely decorative and would be appropriate for headings. | Impact, Western |

Font Style

The font-style property is mostly used to specify whether the letters shape are vertical or slanted.

| Value | Description |
|----------------|---|
| normal | The text is shown normally |
| italic | The text is shown in italics(curved letter forms) |
| oblique | The text is "leaning" (oblique is very similar to italic, it takes the normal font design and just slants it) |

Ex: p {font-style:italic;}

Font Size

The font-size property specifies the size of the text.

| Value | Description |
|----------|---|
| xx-small | Sets the font-size to an xx-small size |
| x-small | Sets the font-size to an extra small size |
| small | Sets the font-size to a small size |
| medium | Sets the font-size to a medium size. This is default |
| large | Sets the font-size to a large size |
| x-large | Sets the font-size to an extra large size |
| xx-large | Sets the font-size to an xx-large size |
| smaller | Sets the font-size to a smaller size than the parent element |
| larger | Sets the font-size to a larger size than the parent element |
| length | Sets the font-size to a fixed size in px, cm, etc. |
| % | Sets the font-size to a percent of the parent element's font size |

Ex: h1 {font-size:40px;}

h2 {font-size:150%;}

Font variant:

Specifies whether or not a text should be displayed in a small-caps font.

| Value | Description |
|------------|---|
| normal | The browser displays a normal font. This is default |
| small-caps | The browser displays a small-caps font |

EX: p {font-variant: small-caps ;}

Font weights

Font-weight property specifies the thickness of the font.

| Value | Description |
|---------|--|
| normal | Defines normal characters. This is default |
| bold | Defines thick characters |
| bolder | Defines thicker characters |
| lighter | Defines lighter characters |

EX:p{font-weight: lighter;}

Text properties:

text-align:

The text-align property specifies the horizontal alignment of text in an element.

| Value | Description |
|-----------|---------------------------------------|
| left | Left-justified |
| right | Right-justified |
| center | Text is centered |
| justified | Text is both right and left justified |

EX: h2 {text-align: left}

text-decoration:

The text-decoration property specifies the decoration added to text.

| Value | Description |
|--------------|--|
| underline | Adds an underline to the text |
| overline | Adds a line on top of the text |
| line-through | Adds a line through the middle of the text |
| blink | Causes the text to blink |

EX:h2 {text-decoration: line-through;}

text-indent:

The text-indent property specifies the indentation of the first line in a text-block.

EX: p {text-indent:50px;}

text-transform:

The text-transform property controls the capitalization of text.

| Value | Description |
|------------|--|
| capitalize | Capitalizes the first letter in a word |
| uppercase | Makes the entire word uppercase |
| lowercase | Makes the entire word lower case |
| none | No transform is performed |

EX:p{text-transform:capitalize;}

List Properties

list-style-type

The list-style-type specifies the type of list-item marker in a list.

| Value | Description |
|----------------------|--|
| none | No marker is shown |
| disc | The marker is a filled circle. This is default for |
| circle | The marker is a circle |
| square | The marker is a square |
| decimal-leading-zero | The marker is a number with leading zeros (01, 02, 03, etc.) |
| lower-alpha | The marker is lower-alpha (a, b, c, d, e, etc.) |
| upper-alpha | The marker is upper-alpha (A, B, C, D, E, etc.) |
| lower-roman | The marker is lower-roman (i, ii, iii, iv, v, etc.) |
| upper-roman | The marker is upper-roman (I, II, III, IV, V, etc.) |

Ex: `ul{list-style-type:square;}`
`ol{list-style-type:upper-roman;}`

list-style-image

The list-style-image property replaces the list-item marker with an image.

| Value | Description |
|-------|--|
| url | The path to the image to be used as a list-item marker |
| none | No image will be displayed. Instead, the list-style-type property will define what type of list marker will be rendered. This is default |

Ex: `ul {list-style-image:url('image.jpg');}`

list-style-position

The list-style-position property specifies if the list-item markers should appear inside or outside the content flow.

| Value | Description |
|---------|---|
| inside | Indents the marker and the text. The bullets appear inside the content flow |
| outside | Keeps the marker to the left of the text. The bullets appears outside the content flow. This is default |

Ex: `ul{list-style-position:inside;}`
`ul{list-style-position:outside;}`

Color properties

Color:

The color property specifies the color of text (foreground color).

We can specify the color values in various formats:

| Format | Syntax | example |
|----------------|---------------------|----------------------------|
| Hex code | #rrggbb | P{color:#fff00;} |
| Short Hex Code | #rgb | P{color:#6b7;} |
| rgb% | rgb(rrr%,ggg%,bbb%) | P{color:rgb(50%,50%,50%);} |
| Rgb absolute | Rgb(rrr,ggg,bbb) | p{color:rgb(0,0,255);} |
| keyword | Aqua,red,black | P{color:orange;} |

```
Ex: body {color: red ;}
    h1 {color: #00ff00 ;}
    p>em {color: rgb (0, 0,255) ;}
```

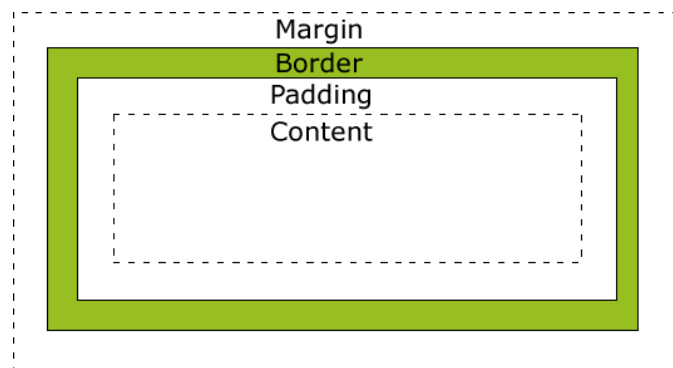
background-color:

The background-color property is used to set the background color.

```
Ex: body {background-color: red ;}
    h1 {background-color: rgb (0, 0,255) ;}
```

CSS Box Model

- The term “box model” is about CSS based layouts and design.
- The box model is a term used when referring to the rectangular boxes placed around every element in the web page.
- An XHTML element can be considered a box, and so the box model applies to all XHTML elements.
- The box model is the specification that defines how a box and its attributes relate to each other.



The 4 different parts of the Box model are:

1. Content:

The innermost part of the box is the content, such as “<h1>”, “”, “<p>” ...etc.
The width and height property defines the width and height of the element.

Ex:

```
P {  
    width: 100px;  
    height: 50px ;  
}
```

2. Padding:

The padding defines the space between the content and the border.
The top, right, bottom, and left padding can be changed independently using separate properties.

Ex:

```
p{  
    padding-top:25px;  
    padding-bottom:25px;  
    padding-right:50px;  
    padding-left:50px;  
}
```

A shorthand padding property can also be used, to change all puddings at once.

Ex: P {padding: 10px 5px 15px 20px ;}

3. Border:

The middle layer in the box model is the element's border.
The space used by the border in the box model is the thickness of the border.

CSS border has following 4 properties:

- border-style
- border-Width
- border -Color
- border - Individual sides

The border-style property sets the style of an element's four borders.

Ex:

```
1) P  
  {  
    border-top-style:dotted;  
    border-right-style:solid;  
    border-bottom-style:dotted;  
    border-left-style:solid;  
  }
```

2)

```
P {  
  border-style:solid;  
  border-color:red;  
  border-width:5px;  
}
```

4. Margin:

The space just outside the border is margin.

The margin is completely invisible.

The top, right, bottom, and left margin can be changed independently using separate properties.

Ex: P

```
{  
  margin-top:100px;  
  margin-bottom:100px;  
  margin-right:50px;  
  margin-left:50px;  
}
```

A shorthand margin property can also be used, to change all margins at once.

Ex: P {margin: 25px 50px 75px 100px ;}

CSS and <div> tags

<div> Tag

<div>(Division) tag divides the content into individual sections .

Div is a block-level container, meaning that there is a line feed after the </div> tag.

Ex: <div style="color:#00FF00">
 <h3>This is a header</h3>
 <p>This is a paragraph.</p>
</div>

 Tag

 tag is used for grouping and applying styles to inline elements.

The span is an inline-level container, meaning that there is no line feed after the tag.

They can only contain text or data in the document like STRONG and EM tags.

Ex :<p>My mother has

```
<span style="color: lightblue; font-weight: bold">light blue</span> eyes  
</p>
```

Background –images

The background-image property is used to add a background image to an element.

Ex: `body {background-image: url ('siley.gif');}`

Background-repeat

By default, the image is repeated both in x- and y- directions so it covers the entire element.

The background-repeat property specifies whether a background image repeats itself.

We can specify no-repeat, repeat-x, and repeat-y.

Ex: `body (background-repeat: repeat-y; }`

Background-position

The position of the image is specified by the background-position property.

The values can be:

A combination of [top, center, bottom] and [left, center, right].

Ex: `body {background-position: center ;}`

4/4 B.TECH CSE 1ST SEMESTER
WEB TECHNOLOGIES

UNIT-2

S.JAYA PRAKASH
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
SIRCRRCOE, ELURU

UNIT-II: Java script

The Basic of Java script: Objects, Primitives Operations and Expressions, Screen Output and Keyboard Input, Control Statements, Object Creation and Modification, Arrays, Functions, Constructors, Pattern Matching using Regular Expressions

DHTML: Positioning Moving and Changing Elements

Java script:

“JavaScript is a cross-platform, object-based scripting language invented specifically for use in web browsers to make websites more dynamic and attractive.”

- ❖ A script is a program or sequence of instructions that is interpreted or carried out by another program.
- ❖ Some languages: VBScript, JavaScript, Jscript and ECMA Script.
- ❖ Scripts can be executed on client or the server.

| Client Side Scripting | Server Side Scripting |
|--|------------------------------|
| Runs on the user's computer i.e. Browser interprets the script | Runs on the Web server |
| Visible to users if they view the HTML source | Not visible to user |

- ❖ JavaScript is mainly used as a client side scripting language.
- ❖ JavaScript was originated with Netscape and it is invented by Brendan Eich.
- ❖ JavaScript was initially called as Live Script.
- ❖ Later in 1995, after an agreement with Sun microsystems, Live Script was re-named JavaScript, to Leverage the popularity of Java.

Uses of JavaScript:

JavaScript as a programming language has many uses. A non-exhaustive list of its uses is given below:

1. JavaScript is used for validating user input on client-side.
2. JavaScript is used for file handling and database connectivity on the server-side.
3. JavaScript can be used as an alternative to applets.
4. JavaScript is used to implement programming on client-side web documents.
5. JavaScript can be used to implement event driven programming.
6. Along with DOM, JavaScript can be used to develop DHTML (Dynamic HTML).

Syntax of JavaScript:

JavaScript is often written in another file which is saved with .js extension. But it can be also embedded with HTML code in a web document. JavaScript is embedded in a web document using the <script> tags as shown below:

```
<script type="text/javascript">  
    ---JavaScript code here---  
</script>
```

The <script> tags can be written in the *head* section and as well in the *body* section for multiple times. They will be interpreted by the JavaScript interpreter in the order in which they are written in the web document. The script which has to be executed once is written in the *body* section and other scripts are written in the *head* section.

Another way of specifying JavaScript is to write it in a separate file. Below example demonstrates external JavaScript:

```
script.js      (JavaScript      file)  
document.write("Welcome to JavaScript");
```

hello.html (Web document)

```
<html>  
    <head><title>First JavaScript</title></head>  
    <body>  
        <script type="text/javascript" src="script.js"></script>  
    </body>  
</html>
```

While specifying JavaScript in another file (like script.js), it should be remembered that there is a need to specify that file name as a value to the *src* attribute of <script> tag. The *type* attribute specifies the MIME type. The closing script tag </script> is mandatory even though there is no content for the script element.

JavaScript Fundamentals:

Primitives:

JavaScript provides five scalar primitive types: Number, Boolean, String, Undefined, Null and two compound primitive types Array and Object. Even though JavaScript supports object-orientation features, it still supports these scalar primitive data types simply because of performance reasons. Maintenance of primitive values is much faster than objects.

JavaScript also provides object versions known as *wrapper objects* for the primitive types Number, Boolean and String namely, *Number*, *Boolean* and *String*. All primitive values are stored on stack whereas objects are stored in heap.

Literals:

A literal is a primitive constant value. In JavaScript all numeric values are of type Number. Internally all numerical values are stored as double-precision floating point values. This is why numeric values are often called as *numbers*.

In JavaScript integer values are a sequence of digits with no decimal points and floating point numbers contains digits and a decimal point or digits and an exponent or both. The exponent can be represented using *e* or *E*. Hexadecimal numbers can be represented by preceding a integer value with *0x* or *0X*.

A string literal is a sequence of characters enclosed in single quotes (') or double quotes ("). A null string can be denoted with a pair of single quotes (") or a pair of double quotes ("").

The only value of type Null is the reserved word *null*, which indicates no value and the only value of type Undefined is *undefined* which is not a reserved word like *null*. If a variable is not explicitly declared or is not assigned, it is said to be *null*. If a variable is explicitly declared and is not assigned, it is said to be *undefined*.

The only possible values for Boolean type are *true* and *false*.

Variables:

A variable is named memory location to store data. JavaScript is a loosely typed (or dynamically typed) language i.e., there is no need to declare the data type of a variable. The type of a variable is dynamically decided at runtime based on the value assigned. A variable can be assigned any of the five primitive type values or it can refer an object.

A variable can be declared implicitly as shown below:

```
pi = 3.14; //The data type of pi is Number
```

A variable can be declared explicitly as shown below:

```
var pi = 3.14; //The data type of pi is Number
```

```
var name = "John"; //The data type of name is String
```

Comments:

Every programming language supports comments through which a programmer can provide details like author name, creation date and purpose of the script. Apart from these details, comments allow programmers to write explanations or reviews about their script or elements in the script.

A single line comment in JavaScript starts with `//`. Everything followed by `//` in that line is treated as a comment.

A single line or multi-line comment starts with `/*` and ends with `*/`. Everything in between these two markers is treated as a comment and will be ignored by the JavaScript interpreter.

Operators:

Numeric Operators:

As any typical programming language, JavaScript provides the numeric operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo), `++` (increment) and `--` (decrement).

If the increment appears before a variable, it is known as pre increment. Otherwise, if it appears after a variable, it is known as post increment. Similarly for decrement operator we have pre decrement and post decrement.

The *precedence rules* of a language specify which operator should be evaluated when there are multiple operators in an expression which belong to different precedence levels. The *associativity rules* specify which operator should be evaluated when an expression contains multiple operations of the same precedence level. The precedence and associativity rules for numeric operators are shown below:

| Operators | Associativity |
|---|----------------------|
| <code>++</code> , <code>--</code> , unary <code>-</code> , unary <code>+</code> | Right-to-Left |
| <code>*</code> , <code>/</code> , <code>%</code> | Left-to-Right |
| <code>+</code> , <code>-</code> | Left-to-Right |

String Concatenation Operator:

Unlike C, strings in JavaScript are not stored or treated as arrays. Strings in JavaScript are unit scalar values. Two strings can be concatenated with each other using the + operator. For example, if a variable *first* holds the string value "Java" then the result of the following expression will be JavaScript:

first + "Script"

The typeof Operator:

The *typeof* operator accepts a single operand either a value or a variable and returns its type as a string value. It returns "number", "string" and "boolean" for values or variables of the type Number, String and Boolean respectively. It returns "object" for values or variables of the type object or null. For a variable which is not assigned a value, the *typeof* operator returns "undefined".

The *typeof* operator can be used in situations where there is a need to be absolute about the type of a value of variable before proceeding further. The *typeof* operator either may include parentheses for the operand or may not. So, both *typeof variable* and *typeof(variable)* are same.

Assignment Statements:

Like any other typical programming language, JavaScript supports assignment statements. An assignment statement contains the assignment operator or any one of the compound assignment operators as shown below:

$x += 1$; is same as:

$x = x + 1$;

Type Conversion:

The mechanism of converting a value or variable from one type to another type is known as type conversion. There two types of type conversion: implicit type conversion and explicit type conversion.

Implicit Type Conversion:

Type conversion which is performed automatically by the JavaScript interpreter is known as implicit type conversion or *coercion*. Implicit type conversion may take place when the expected type is different from the actual type of the value given from the user or read from a file etc. For example, consider the following expression:

$"John" + 123$

It can be observed that in the above expression, + is a concatenation operator. So, even though the second operand is a number, it will be converted (implicit conversion) to a string and will be concatenated with "John". Now, consider the following example:

$8 * "2"$

The * operator is only applicable on numbers. So, the second operand will be converted from a string to a number and the resultant value will be 16.

Boolean *false* will be equal to 0 and *true* will be equal to 1 when converted to number. Both *null* and *undefined* will become *false* when converted to Boolean. A number 0 will be converted to *false* and any other number will be converted to *true*, when it is converted to a Boolean.

Explicit Type Conversion:

Sometimes there is a need for the programmer to force type conversion. Such conversion specified by the programmer is known as explicit type conversion or *type casting*. For example, a number can be casted to a string using the String constructor as shown below:

```
var str = String(5);
```

We can also cast a number to String using the toString() method which provides an advantage of specifying the *base* of the number. Consider the following expressions:

```
var x = 5;  
var y = x.toString( );  
var z = x.toString(2); //2 specifies the base of the result. 5 in base 2 is 101
```

In the above expressions, y will have the string "5" and z will have the string "101". A string can be casted to a number using the Number constructor as shown below:

```
var x = Number("10");
```

The above code works only when the supplied string parameter does not contain any other characters after the number. To convert a string which contains a number at any position, we can use parseInt() and parseFloat() methods.

Strings:

A string is a collection of characters. Most of the times in scripts, there is a need to work with strings. JavaScript provides various properties and methods to work with String objects. Whenever a String property or method is used on a string value, it will be coerced to a String object.

One most frequently used property on String objects is *length*. The *length* property gives the number of characters in the given string. Consider the following example:

```
var str = "Hello World";  
var len = str.length;
```

The value stored in the *len* variable will be 11 which is the number of characters in the string. Frequently used methods on String objects are given below:

| Method | Description |
|-----------------------------|--|
| charAt(pos) | Returns the character at specified position |
| indexOf(char) | Returns the position of the given character in the string |
| substring(startpos, endpos) | Returns the substring in between the specified positions |
| toLowerCase() | Returns the string with all characters converted to lower case |
| toUpperCase() | Returns the string with all characters converted to upper case |

Math Object:

The Math object provides a number of properties and methods to work with Number values. Among the methods there are *sin* and *cos* for trigonometric functions, *floor* and *round* for truncating and rounding the given numbers and *max* for returning the maximum of given numbers.

Number Object:

The number object contains a collection of useful properties and methods to work with Numbers. The properties include: `MAX_VALUE` (represents largest number that is available), `MIN_VALUE` (represents smallest number that is available), `NaN` (represents Not a Number), `POSITIVE_INFINITY` (special value to represent infinity), `NEGATIVE_INFINITY` (special value to represent negative infinity) and `PI` (represents value of π).

To test whether the value in a variable is NaN or not, there is a method `isNaN(var)`, which returns *true* if the value in the specified variable is NaN and *false* otherwise. To convert a Number to a String, use the method `toString()` as shown below:

```
var cost = 250;
var output = cost.toString( );
```

Date Object:

At times you there will be a need to access the current date and time and also past and future date and times. JavaScript provides support for working with dates and time through the *Date* object.

Date object is created using the *new* operator and one of the *Date's* constructors. Current date and time can be retrieved as shown below:

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

Below are some of the frequently used methods available on a Date object:

| Method | Description |
|-----------------------------|----------------------------------|
| <code>toLocaleString</code> | A string of the date information |
| <code>getDate</code> | The day of the month |

| | |
|-----------------|--|
| getMonth | The month of the year, as a number in the range of 0 to 11 |
| getDay | The day of a week, as a number in the range 0 to 6 |
| getFullYear | The year |
| getTime | Number of milliseconds since Jan 1, 1970 |
| getHours | Number of the hour, as a number in the range 0 to 23 |
| getMinutes | Number of the minute, as a number in the range 0 to 59 |
| getSeconds | Number of the second, as a number in the range 0 to 59 |
| getMilliseconds | Number of milliseconds, as a number in the range of 0 to 999 |

Input and Output:

JavaScript models the HTML/XHTML document as the *document* object and the browser window which displays the HTML/XHTML document as the *window* object. The *document* object contains a method named *write* to **output** text and data on to the document as shown below:

```
document.write("Welcome to JavaScript");
```

To display multiple lines of text using the *write* method, include the HTML tag `
` as a part of the string parameter as shown below:

```
document.write("Welcome<br />to<br />JavaScript");
```

The output of the above code on the document will be as shown below:

```
Welcome  
to  
JavaScript
```

Although the *document* object has another method named *writeln* which appends a new line (`\n`), the output of both *write* and *writeln* will be same as the browser ignores the new line characters.

The ***window*** object is the default object in JavaScript. So, whenever a property or method on *window* object is to be referenced, there is no need to mention *window* object explicitly.

There are three methods available on the *window* object to perform input/output namely *alert* and *confirm* for output and *prompt* for input. The *alert* method is used to display a dialog box which shows the provided text or data as output along with a *OK* button. The *alert* method can be used as shown below:

```
alert("Sum of 10 and 20 is: " + (10 + 20));
```

The *alert* method allows the new line character (\n) for displaying text in multiple lines and does not support HTML tags like
 etc.

The *confirm* method displays a dialog box which displays the string provided as output along with two buttons labelled *OK* and *Cancel*. This method returns a Boolean value *true* when the user clicks *OK* button and *false* when user clicks *Cancel* button. The *confirm* method can be used as shown below:

```
var status = confirm("Do you want to proceed?");
```

The *prompt* method is used to accept string input from the user. This accepts two parameters. First parameter is a string that will be displayed on the dialog box and the second string is the default input string in case the user doesn't provide any input. The *prompt* method displays a string along with a textbox and two buttons labeled *OK* and *Cancel*. Prompt box can be created as shown below:

```
var a = prompt("Enter a value: ", "");
```

Control Statements:

Statements that are used to control the flow of execution in a script are known as control statements. Control statements are used along with compound statements (a set of statements enclosed in braces). Local variables are not allowed in a *control construct* (control statement + single/compound statement). Even though a variable is declared within a control construct, it is treated as a global variable.

Control Expressions:

The expressions upon which the flow of control depends are known as control expressions. These include primitive values, relational expressions and compound expressions. The result of evaluating a control expression is always a Boolean value either *true* or *false*.

A relational expression has two operands and one relational operator which may be one of the following:

| Operator | Operation |
|-----------------|--------------------------|
| == | Equal to |
| != | Not equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| === | Is strictly equal to |
| !== | Is strictly not equal to |

The === and !== relational operators considers the type of the operands while evaluating the expression.

JavaScript also supports the logical operators && (AND), || (OR) and ! (NOT).

Operator precedence (highest to lowest) and associativity rules which are followed in expression evaluation are given below:

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

Selection Statements:

Statements which are used to execute a set of statements based on a condition are known as selection statements. In JavaScript selection statements are: if, if-else and switch.

Syntax for an **if** statement is shown below:

```
if(expression)  
{  
    statement(s);  
}
```

Syntax for an **if-else** statement is shown below:

```
if(expression)  
{  
    statement(s);  
}  
else  
{  
    statement(s);  
}
```

Syntax of a **switch** statement is given below:

```
switch(expression)
{
    case label1:
        statement(s);
        break;
    case label2:
        statement(s);
        break;
    ...
    default:
        statement(s);
}
```

The *expression* and *labels* in a switch statement can be either numbers, strings or Boolean values.

Loop Statements:

Statements which are used to execute a set of statements repeatedly based on a condition are known as loop statements or iteration statements. Loop statements supported by JavaScript are while, do-while, for and for-in.

Syntax of **while** loop is given below:

```
while(condition)
{
    statement(s);
}
```

Syntax for **do-while** loop is given below:

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

In a do-while statement condition is evaluated after the body of the loop is executed. So, the difference between while and do-while is, unlike in while loop, the body of the loop is guaranteed to be executed at least once in a do-while loop.

Syntax of **for** loop is given below:

```
for(initialization; condition; increment/decrement expression)
{
    statement(s);
}
```

JavaScript provides another loop statement to work with objects which is known as a for-in loop. Its syntax is given below:

```
for(identifier in object)
{
    statement(s);
}
```

Creation and Manipulation of Objects:

An object is a real world entity that contains properties and behavior. Properties are implemented as identifiers and behavior is implemented using a set of methods. An object in JavaScript doesn't contain any predefined type. In JavaScript the *new* operator is used to create a blank object with no properties. A constructor is used to create and initialize properties in JavaScript. In Java *new* operator is used to create the object and its properties and a constructor is used to initialize the properties of the created object.

An object can be created as shown below:

```
var obj = new Object( );
```

The properties of an object can be accessed using the dot (.) operator. In JavaScript, the properties are not fixed as in Java. Number of properties can vary at runtime i.e., new properties can be added as and when needed. In JavaScript properties are not variables, they are just names which are used to access values and hence are never declared. Creating and accessing properties is shown below:

```
var person = new Object( );  
person.name = "jps";  
person.branch = "cse";  
document.write(person.name);
```

Another way for creating the above object is shown below:

```
var person = {name: "jps", branch: "cse" };  
Nested objects can be created as shown below:  
person.address = new Object( );  
person.address.dno = "Door Number";  
person.address.street = "Street Name";
```

The properties can be accessed with the dot operator or using the property name as an array subscript. For example let's consider printing the name of the person object created above:

```
document.write(person.name);
```

or

```
document.write(person["name"]);
```

To step through all the properties of an object we can use the for-in loop as shown below:

```
for(var p in person)  
{  
    document.write("Property Name: ", p, "; Value: ", person[p], "<br />");  
}
```

Arrays:

An array is a collection of elements. Unlike in Java, arrays in JavaScript can contain elements of different types. A data element can be a primitive value or a reference to other objects or arrays.

An Array object can be created in two ways. First way is by using the *new* operator as shown below:

```
var array1 = new Array(15, 25, "hello");  
var array2 = new Array(15);
```

In the first declaration, the size of array1 is 3 and the values are initialized to 15, 25 and "hello". In the second declaration, the size of array2 is 15 and the elements are not initialized.

Second way to create an Array object is by specifying a literal array in square brackets as shown below:

```
var array3 = [1, 2, "hello"];
```

In the above declaration, size of array3 is 3 and values are initialized to 1, 2 and "hello" respectively.

An array element can be accessed using the index or subscript which is included in square brackets. In JavaScript the index of every array starts with zero. An array element can be accessed as shown below:

```
document.write("Second array element is: ", array3[1]);
```

Arrays in JavaScript are dynamic and are always allocated memory from heap. The size of an array can be accessed or changed by using the *length* property of an Array object as shown below:

```
var array1 = new Array(15); //Length is 15  
array1.length = 25; //Now length is 25
```

Array Methods:

Several methods are available on an Array object to perform various manipulations on the array elements. Most frequently used methods are mentioned here:

The method *join* is used to convert the array elements into string and join them into a single string separated by the specified string. If no separator is given, all the elements are joined using commas. Below example demonstrates *join* method:

```
var names_array = ["Ken", "John", "Mary"];  
var names_string = names_array.join("-");
```

Now *names_string* contains the string "Ken-John-Mary".

The method *reverse* is used to reverse the order of array elements and the method *sort* converts the array elements into strings and then sorts them in alphabetical order.

The method *concat* appends the specified elements to the end of the Array object as shown below:

```
var names = ["Ken", "John", "Mary"];  
var names = names.concat("Mike", "James");
```

Now, *names* array contains, "Ken", "John", "Mary", "Mike" and "James". Length of *names* array will be 5.

The method *slice* is used to retrieve subset of elements in an Array object. This method accepts two parameters, starting index and ending index. All the elements from starting index excluding the ending index are retrieved. Below example demonstrates *slice* method:

```
var nums1 = [10, 20, 35, 45, 55];  
var nums2 = nums1.slice(2,4);
```

Now *nums2* contains [35, 45]. If no ending index is specified, all the elements from starting index to last element in the array are returned.

The method *toString* is used to convert the array elements into strings and concatenate those using commas. This method behaves same as *join* method when applied on array of strings.

The methods *push* and *pop* behave as stack operations. The method *push* is used to append an element at the last position in an array and the method *pop* is used to return the last element of the array.

The methods *shift* and *unshift* are used to remove and add elements at the starting index of the array respectively.

Two-Dimensional Arrays:

A two-dimensional array in JavaScript can be thought of as an array of arrays. It can be created using the *new* operator or with nested array literals as shown below:

```
var array1 = new Array(1, 2, 3);  
var array2 = new Array(2, 3, 4);  
var array3 = new Array(1, 3, 4);
```



```
var twod = new Array(array1, array2, array3); //Two-dimensional array twod
```

```
var twod_array = [ [1, 2, 3], [2, 3, 4], [4, 5, 6] ]; //Two-dimensional array twod_array
```

Functions:

A function is a block of code which can be executed again and again. Functions make the code modular which improves maintenance. Functions in JavaScript are similar to functions in C language. A function definition is as shown below:

```
function functionName(param1, param2, ... , paramN)
{
    //Body of the function
    return; //This is optional
}
```

The function definition consists of function header and the body of the function (a compound statement). The function header consists of *function* keyword, function name followed by parameters (if any) enclosed in round brackets. The function body contains a set of statements and an optional *return* statement that are executed when the function is called. A function call is as shown below:

```
functionName(param1, param2, ... , paramN);
```

In JavaScript functions are objects. So, variables that reference them can be treated as object references. Example for function definition and function call is given below:

```
//Function Definition
function add(x, y)
{
    return x+y;
}
```

//Function Call

addref = add; //Here addref is a reference to the add(function) object

add(10, 20);

addref(20, 30); //This is also valid

In JavaScript, a function definition must occur before a function call. So, it is wise to define a function in the *head* section of the page.

Local Variables:

The scope of a variable denotes the range of statements over which the variable is visible. In JavaScript we have two kinds of variables: global variables and local variables.

Variables that are declared implicitly (without *var*) outside a function or inside a function definition have *global* scope. Variables that are declared explicitly (with *var*) outside (including compound statements) a function also have *global* scope. Variables that are declared explicitly inside a function definition have *local* scope.

When a global variable and a local variable have the same name, the local variable hides the global variable.

Parameters:

The parameters available in the function call are known as *actual parameters* while the parameters in the function definition are known as *formal parameters*. When the formal parameters are more than the actual parameters, remaining formal parameters are set to *undefined*. When the formal parameters are less than the actual parameters, remaining actual parameters are truncated.

The actual parameters passed to a function can be accessed through the property array, *arguments*. So, even when the formal parameters are less than the actual parameters, all the actual parameters are directly accessible through *arguments* property.

We can use the arguments property as shown below:

```
arguments.length //Gives the number of actual parameters  
arguments[n] //Access the nth actual parameter
```

In JavaScript, parameters are passed by value. Even though object references are being passed, references itself are passed as a value.

Constructors:

A constructor is a special function which is used to create and initialize the properties of an object. A constructor has the same name as the object. A constructor can be called by using the *new* keyword. A constructor can reference the properties of its object by using *this* keyword. A constructor can be defined as shown below:

```
function person(p_id, p_name)  
{  
    this.id    =    p_id;  
    this.name = p_name;  
}
```

Now, the above constructor *person* can be called by *new* keyword as shown below:

```
person1 = new person (10, "ramu");
```

If the object should also contain a method, it is initialized in the same way as a property is initialized. Let's consider a method for displaying the details of the *person* object as shown below:

```
function display_person( )  
{  
    document.write("Person id is: ", this.id, "<br />");  
    document.write("Person name is: ", this.name);  
}
```

Add a property named *display* to the constructor as shown below:

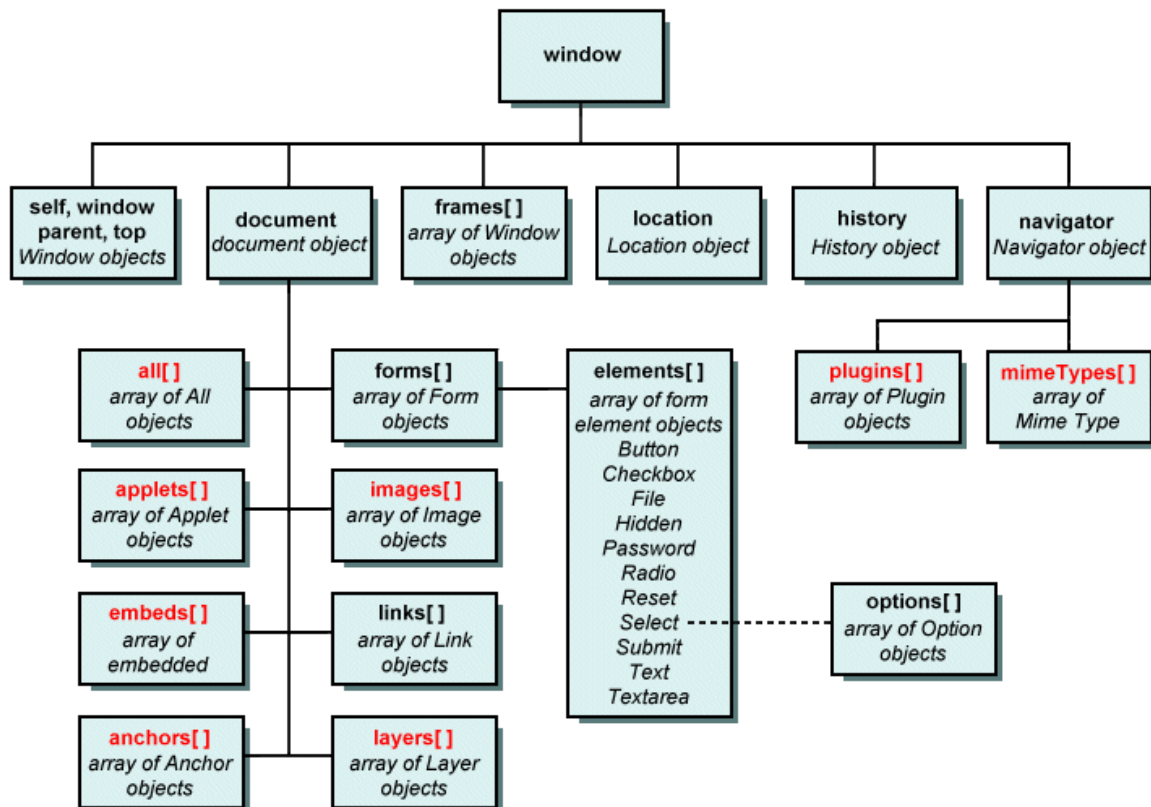
```
this.display = display_person;
```

Now, we can access the method to print the details of a *person* object as shown below:

```
person1.display( );
```

Document Object Model (DOM):

DOM is an Application Programming Interface (API) for application programs to interact with the web documents. Using DOM along with JavaScript, we can create documents, navigate through the document structure, add or delete elements and their content. Documents in the DOM have a tree like structure as shown below:



Accessing Elements in a Document:

Different elements in a web document are treated as objects in JavaScript and each object has properties and methods. Using DOM, we can get the address of an HTML element in different ways.

First way is to use the *document* object's *forms* array property along with the *elements* array property. To understand this, let's consider the following HTML code:

```
<html>
  <head><title>Simple form</title></head>
  <body>
    <form action="">
      Enter your name:
      <input type = "text" />
    </form>
  </body>
</html>
```

To obtain the address of the textbox in the above HTML code, you can write the following code in your script:

```
var name = document.forms[0].elements[0].value;
```

The disadvantage of this method is, it becomes difficult to obtain the address of elements when there are multiple forms in a document or if there are a large number of elements in a form or if new elements are added to a form.

The second way is by using the *name* attribute of the HTML elements. To demonstrate this, let's consider the following HTML code:

```
<html>
  <head><title>Simple form</title></head>
  <body>
    <form action="" name="frmmain">
      Enter your name:
      <input type = "text" name="txtname" />
    </form>
  </body>
</html>
```

To obtain the address of the textbox in the above HTML code, you can write the following code in your script:

```
var name = document.frmmain.txtname.value;
```

The problem with this method is, it doesn't work with a group of checkboxes or a group of radio buttons which will have the same value for their *name* attribute. Also XHTML 1.1 does not allow *name* attribute on form tag.

The third way and the most commonly used method is by using the *getElementById* method which was introduced in DOM 1. To demonstrate this let's consider the following HTML code:

```
<html>
  <head><title>Simple form</title></head>
  <body>
    <form action="">
      Enter your name:
      <input type = "text" id="txtname" />
    </form>
  </body>
</html>
```

To obtain the address of the textbox in the above HTML code, you can write the following code in your script:

```
var name = document.getElementById("txtname");
```


Since the value for *id* attribute can be different for checkboxes and radio buttons in a group, there will be no problems.

The *id* and *name* attributes can be used in combination for processing a group of checkboxes or radio buttons as shown below:

//HTML code

```
<form id = "genderGroup">
    <input type="radio" name="gender" value="male" />Male
    <input type="radio" name="gender" value="female" />Female
</form>
```

//JavaScript code

```
var count = 0;
var dom = document.getElementById("genderGroup");
for(index = 0; index < dom.gender.length; index++)
    if(dom.gender[index].checked)
        count++;
```

Events and Event Handling:

The programming which allows computations based on the activities in a browser or by the activities performed by the user on the elements in a document is known as *event-driven programming*.

An *event* is the specification (essentially an object created by the browser and JavaScript) of something significant has occurred. Examples of events are *click*, *submit*, *keyup* etc. In JavaScript all the event names are specified in lowercase. An *event handler* (essentially a function) is a set of statements (code) that handles the event.

Below table lists most commonly used events and their associated tag attributes:

| Event | Tag Attribute |
|--------------|----------------------|
| blur | onblur |
| change | onchange |
| click | onclick |
| dblclick | ondblclick |
| focus | onfocus |
| keydown | onkeydown |
| keypress | onkeypress |
| keyup | onkeyup |
| load | onload |
| mousedown | onmousedown |
| mouseup | onmouseup |
| mousemove | onmousemove |
| mouseover | onmouseover |
| mouseout | onmouseout |
| reset | onreset |
| select | onselect |
| submit | onsubmit |
| unload | onunload |

Below table lists event attributes and their corresponding tags in HTML:

| Attribute | Tag | Description |
|------------------|------------|--|
| onblur | <a> | The link loses input focus |
| | <button> | The button loses input focus |
| | <input> | The input element loses focus |
| | <textarea> | The text area loses focus |
| | <select> | The selection element loses focus |
| onchange | <input> | The input element is changed and loses focus |
| | <textarea> | The text area changes and loses focus |

| | | |
|-------------|--|---|
| | <select> | The selection element is changed and loses focus |
| onclick | <a> <input> | The user clicks on the link The input element is clicked |
| ondblclick | Most elements | The user double clicks the mouse left button |
| onfocus | <a> <input> <textarea> <select> | The link acquires focus The input element acquires focus A text area acquires focus A selection element acquires focus |
| onkeydown | <body> form elements | A key is pressed down |
| onkeypress | <body> form elements | A key is pressed down and released |
| onkeyup | <body> form elements | A key is released |
| onload | <body> | The document finished loading |
| onmousedown | Most elements | The user clicks the left mouse button |
| onmouseup | Most elements | The left mouse button is released |
| onmousemove | Most elements | The user moves the mouse cursor on the element |
| onmouseover | Most elements | The mouse cursor is moved over the element |
| onmouseout | Most elements | The mouse cursor is moved away from the element |
| onreset | <form> | The <i>reset</i> button is clicked |
| onselect | <input> <textarea> | The mouse cursor is moved over the element The text is selected within the text area |
| onsubmit | <form> | The <i>submit</i> button is pressed |
| onunload | <body> | The user exits the document |

The process of linking an event handler with an event is known as *registration*. There are two ways to register an event handler in DOM 0 event model. First way is by assigning the event handler script to an event tag attribute as shown below:

```
<input type="button" value="Click Me" onclick="func1( );" />
```

In the above code, the event handler *func1* is assigned to the event attribute *onclick* of a button.

The second way is by assigning the event handler to the event property of the element as shown below:

```
document.getElementById(ID_of_element).onclick = func1;
```

Form Validation:

One of the best uses of client-side JavaScript is the form input validation. The input given by the user can be validated either on the client-side or on the server-side. By performing validation on the client-side using JavaScript, the advantages are fewer loads on the server, saving network bandwidth and quicker response for the users.

Form input can be validated using different events, but the most common event used is *submit* i.e when the submit button is clicked. Corresponding attribute to be used is *onsubmit* of the *form* tag. Let's consider a simple form as shown below:

//HTML Code

```
<html>
  <head>
    <title>Login Form</title>
  </head>
  <body>
    <form id="loginForm" action="next.html">
      Username: <input type="text" id="txtuser" /><br />
      Password: <input type="pass" id="txtpass" /><br />
      <input type="submit" value="Submit" />
      <input type="reset" value="Clear" />
    </form>
    <script type="text/javascript" src="script.js"></script>
  </body>
</html>
```

```

//JavaScript Code - script.js
function checkData( )
{
    var txtuser = document.getElementById("txtuser").value;
    var txtpass = document.getElementById("txtpass").value;
    if(txtuser == "")
    {
        alert("Username must not be blank!");
        return false;
    }
    if(txtpass == "")
    {
        alert("Password must not be blank!");
        return false;
    }
    return true;
}
document.getElementById("loginForm").onsubmit = checkData;

```

The above example can be written in another way as shown below:

//HTML Code

```

<html>
    <head>
        <title>Login Form</title>
        <script type="text/javascript" src="script.js"></script>
    </head>
    <body>
        <form id="loginForm" action="next.html" onsubmit="checkData( );">
            Username: <input type="text" id="txtuser" /><br />
            Password: <input type="pass" id="txtpass" /><br />
            <input type="submit" value="Submit" />
            <input type="reset" value="Clear" />
        </form>
    </body>
</html>

```

```
//JavaScript Code - script.js
function checkData( )
{
    var txtuser = document.getElementById("txtuser").value;
    var txtpass = document.getElementById("txtpass").value;
    if(txtuser == "")
    {
        alert("Username must not be blank!");
        return false;
    }
    if(txtpass == "")
    {
        alert("Password must not be blank!");
        return false;
    }
    return true;
}
```

The general validation process is to check whether the user input matches the requirements. If it matches return *true* which will make the browser shift the control to the page mentioned in the *action* attribute of the *form* tag. Otherwise display appropriate error message and return *false* which will make the control stay in the current page.

Dynamic HTML:

Dynamic HTML is not a new markup language. It is a collection of technologies (generally HTML, CSS and JavaScript) which changes the document content once it is loaded into a web browser.

Some examples for dynamism in a web document are given below:

- ❖ Changes to document content on user interactions or after a specified time interval or on specific browser events.
- ❖ Moving elements to new positions.
- ❖ Making elements disappear and reappear.
- ❖ Changing the color of the background and foreground.
- ❖ Changing the font properties of elements.
- ❖ Changing the content of the elements.
- ❖ Overlapping elements in stack order.
- ❖ Allow dragging and dropping elements anywhere in the browser window.

As an example let's consider a web document which contains an image *image1* initially when the page is loaded. When the user clicks on the image, the image changes to *image2* which demonstrates dynamism and hence Dynamic HTML (DHTML).

//HTML Code

```
<html>
  <head>
    <title>DHTML Demo</title>
    <script type="text/javascript" src="script.js"></script>
  </head>
  <body>
    
  </body>
</html>
```



```
//JavaScript code -  
script.js
```

```
function change( )  
{  
var img = document.getElementById("image");  
img.src = image2.jpg;  
}  
document.getElementById("image").onclick = change;
```

PATTERN MATCHING USING REGULAR EXPRESSIONS

Pattern matching is a text search technique used in JavaScript for form validation, replacing text, processing user input, and parsing text.

Regular expression is a pattern of characters following a specialized syntax used to search text for a matching pattern.

literal: A literal is any character we use in a search or matching expression .

metacharacter: A meta character is one or more special characters that have a unique meaning and are NOT used as a literals in the search expression .

target string: This describes the string that we will be searching.

Search expression: This describes the expression that we will be using to search our target string .

Escape sequence: It is a way of indicating that we want to use one of our metacharacter as a literal.

| Metacharacter | Meaning |
|--------------------|---|
| *(asterisk) | It matches the preceding character 0 or more times. Ex: tre* will find T rough(0 times) t read(1 time) and tree (2times) |
| +(plus) | It matches the preceding character 1 or more times only. Ex: tre+ will find Tree (2 times) and tread (1 time) but not through (0 times) |
| ?(question mark) | It matches the preceding character 0 or 1 times only. Ex: Colou?r will find Both color (0times) and colour(1 times) |
| .(period) | Matches any character in this position Ex: pra. Will find prak ,praka |
| -(dash) | The dash(-) inside the square bracket is used to define the range. Ex: [0-9] means check for 0 to 9 |
| ^(circumflex) | 1. The circumflex (^) inside the square bracket is used to negate the expression. Ex: [^Ff] means anything except upper or lower case F . 2. The ^ outside the square bracket means look only at beginning of the target string. Ex: ^Moz will find ‘Mozilla fireFox’ |
| \$(dollar) | The \$ means look only at the end of the target string. Ex: fox\$ will find a match in “Silver fox” |
| [](square bracket) | Match anything inside the square brackets for one position once and only once. Ex: [0123456789] |
| (alternation) | The means find the left hand Or right hand values. Ex: gr(a e)y will find gray and grey |
| ()(paranthesis) | Used group parts of our search expression |
| {n} | Matches the preceding character n times exactly. Ex: [0-9]{2} –[0-9]{10} Will find any number of the form 91-9848143200 |
| {n,m} | Matches the preceding character atleast n times but not more than n times. Ex: ba{2,3}b will find ‘baab’ and ‘baaab’ but not ‘bab’ or ‘baaaaab’ |

4/4 B.TECH CSE 1ST SEMESTER
WEB TECHNOLOGIES

UNIT-3

S.JAYA PRAKASH
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
SIRCRRCOE, ELURU

UNIT-III:

XML: Document type Definition, XML schemas, Document object model, XSLT, DOM and SAX Approaches,

AJAX A New Approach: Introduction to AJAX, Integrating PHP and AJAX.

I. Introduction to XML:

XML stands for eXtensible Markup Language and is a text-based markup language derived from Standard Generalized Markup Language (SGML). The primary purpose of this standard is to provide way to store self-describing data easily. Self-describing data are those that describe both their structure and their content. But, HTML documents describe how data should appear on the browsers screen and no information about the data. XML documents, on the other hand describe the meaning of data. The content and structure of XML documents are accessed by software module called XML processor.

XML Characteristics:

1. **XML is extensible:** XML essentially allows you to create your own language, or tags, that suit your application.
2. **XML separates data from presentation:** XML allows you to store content with regard to how it will be presented.
3. **XML is a public standard:** XML was developed by an organization called the World Wide Web Consortium (W3C) and available as an open standard.

XML Usage:

A short list of XML's usage says it all

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange of information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange data in a way that is customizable for your needs.
- XML can easily be mixed with style sheets to create almost any output desired.

XML features:

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

II. XML document structure:

An XML document consists of following parts: 1) Prolog 2) Body

1. Prolog:

This part of XML document may contain following parts: XML declaration, Optional processing instructions, Comments and Document Type Declaration

XML Declaration:

Every XML document should start with one-line XML declaration which describes document itself. The XML declaration is written as below:

Syntax: <?xml version="1.0" encoding="UTF-8"?>

Where *version* is the XML version and *encoding* specify the character encoding used in the document. UTF-8 stands for Unicode Transformation Format used for set of ASCII characters. It also have *standalone* attribute indicates whether the document can be processed as standalone document or is dependent on other document like Document Type Declaration (DTD).

Syntax: <?xml version="1.0" encoding="UTF-8" standalone="yes|no"?>

Processing Instruction:

Processing Instructions starts with left angular bracket along with question mark (<?), ending with question mark followed by the right angular bracket(>?). These parameters instruct the application about how to interpret XML document. XML parser's do not take care of processing instructions and are not text portion of XML document.

Ex: <?xsl-stylesheet href="simple.xsl" type="text/xsl"?>

Comments:

Like HTML, comments may use anywhere in XML documents. An XML comments starts with `<!--` and ends with `-->`. Everything within these will be ignored by the parsers and will not be parsed.

Syntax: <!-- this is comments -->

Following points should be remembered while using comments: do not use double hyphens, never place inside entity declaration or within any tag, never place before XML declaration

Document Type Declaration (DTD):

XML allows creating new tags and having meaning if it has some logical structure created using set of related tags. `<!DOCTYPE >` is used to specify the logical structure of XML document by imposing constraints on what tags can be used and where. DTD may contain Name of root element, reference to external DTD, element and entity declarations.

2. Body:

This portion of XML document contains textual data marked up by tags. It must have one element called Document or Root element, which defines content in the XML document. Root element must be the top-level element in the document hierarchy and there can be one and only one root element.

Ex: <?xml version="1.0"?>

```
<book>  
  <title>WT</title>  
  <author>Uttam Roy</author>  
  <price>500</price>  
</book>
```

In this document, the name of root element is `<book>` which contains sub tags `<title>`, `<author>` and `<price>`. Each of these tags contains text “WT”, “Uttam Roy” and “500” respectively.

III. XML Elements

An XML element consists of starting tag, an ending tag and its contents and attributes. The contents may be simple text or other element or both. XML tags are very much similar to that of HTML tags. A tag begins with less than (<) and ends with greater than (>) character. It takes the form <tag-name> and must have corresponding ending tag tag-name>). An element consists of opening tag, closing tag and contents. Few tags may not contain any content and hence know as Empty elements. According to the well-formed ness constraint, every XML element must have closing tag. XML provides two ways for XML empty elements as follows:

*Syntax:
</br> or
*

Following are the rules that need to be followed for XML elements:

- An element *name* can contain any alphanumeric characters. The only punctuation allowed in names are the hyphen (-), under-score (_) and period (.)
- Names are case sensitive. For example Address, address, ADDRESS are different names
- Element start and end tag should be identical
- An element which is a container can contain text or elements as seen in the above example

Attributes:

Attributes are used to describe elements or to provide more information about elements. They appear in the starting tag of element. The syntax of specifying an attribute in element is:

Syntax: <element-name attribute-name="value">...</element-name> Ex: <employee gender="male">ABCD</employee>

There is no strict rule that describes when to use elements and when to use attributes. However, it is recommended not to use attributes as far as possible due to following reasons:

- Too many attributes reduce readability of XML document
- Attributes cannot contain multiple values, but elements can
- Attributes are not easily extendable
- Attributes cannot represent logical structure, but elements together with their child elements can
- Attributes are difficult to access by parsers
- Attribute values are not easy to check against DTD

Well-formed XML:

An XML document is said to be well-formed if it contains text and tags that conform with the basic XML well-formed ness constraints. XML can extend existing documents by creating new elements that fit their applications. The only thing is to remember the well-formed ness constraints. The following rules must be followed by XML documents:

- An XML document must have one and only one root element
- All tags must be closed
- All tags must be properly nested
- XML tags are case-sensitive
- Attributes must always be quoted
- Certain characters are reserved for processing like pre-defined entities

Pre-defined Entities: W3C specification defined few entities each of which represents a special character that cannot be used in XML document directly. All XML processors must recognize those entities, whether they are declared or not.

| Entity Name | Entity Number | Description | Character |
|--------------------|----------------------|--------------------|------------------|
| < | < | Less than | < |
| > | > | Greater than | > |
| & | & | Ampersand | & |
| " | " | Quotation mark | “ |
| ' | ' | Apostrophe | ‘ |

Valid XML:

Well-formed XML documents obey only basic well-formedness constraints. So, valid XML documents are those that are well formed and comply with rules specified in DTD or Schema.

Name Space:

XML was developed to be used by many applications. If many applications want to communicate using XML documents, problems may occur. In XML document, element and attribute names are selected by developers. In some cases two different documents may have same root element. For example, both client.xml and server.xml contains same root tag <config> as shown below.

| <u>Client.xml</u> | <u>Server.xml</u> |
|--------------------------|--------------------------|
| <config> | <config> |
| <version>1.0</version> | <version>1.0</version> |
| </config> | </config> |

XML namespace provides simple, straightforward way to distinguish between element names in XML document. Namespace suggests to use prefix with every element as follows:

| <u>Client.xml</u> | <u>Server.xml</u> |
|----------------------------|----------------------------|
| <c:config> | <s:config> |
| <c:version>1.0</c:version> | <s:version>1.0</s:version> |
| </c:config> | </s:config> |

Uniform Resource Identifier (URI) is used to guarantee the prefixes used by different developers. In general URI's are used to choose unique name. But, URI must be prefixed for each tag instead of them we use prefix. Prefixes are just shorthand placeholders of URIs. Association of prefix and URI is done in the starting tag using reserved XML attribute *xmlns*.

Syntax: xmlns:prefix="URI"

Name Space Rules: The *xmlns* attribute identifies namespace and makes association between prefix and created namespace. Many prefixes may be associated with one namespace.

Default Namespace: Namespaces may not have their associated prefixes and are called default namespace. In such cases, a blank prefix is assumed for element and all of its descendants.

XML Schema languages:

Schema is an abstract representation of object characteristics and its relationship to other objects. An XML schema represents internal relationship between elements and attributes in XML document. It defines structure of XML documents by specifying list of valid elements and attributes. XML schema language is a formal language to express XML schemas. Most popular and primary schema languages are: DTD and W3C Schema.

IV. Document Type Declaration (DTD):

It is one of the several XML schema languages and was introduced as part of XML 1.0. Even though DTD is not mandatory for an application to read and understand XML document, many developers recommend writing DTDs for XML applications. Using DTD we can specify various elements types, attributes and their relationship with in another. Basically DTD is used to specify set of rules for structuring data in any XML file.

Using DTD in XML document:

To validate XML document against DTD, we must tell validator where to find DTD so that it knows rules to be verified during validation. A Document Type Declaration is used to make such link and DOCTYPE keyword is used for this purpose. There are three ways to make this link: Internal DTD, External DTD and Combined internal and external.

1. Internal DTD:

When we embed DTD in XML document, DTD information is included within XML document itself. Specifically, DTD information is placed between square brackets in DOCTYPE declaration. The general syntax of internal DTD is

Syntax: `<!DOCTYPE root-element [
 element-declarations
]>`

Where *root-element* is the name of root element and *element-declarations* is where we declare the elements. Since every XML document must have one and only one root element, this is also structure definition of the entire document. Here, DOCTYPE must be in uppercase, document type declaration must appear before first element and name following word DOCTYPE i.e. root-element must match with name of root element.

Advantage of internal DTD is that we have to handle only single xml document instead of many which is useful for debugging and editing. It is a good idea to use with smaller sized documents. Problem of internal DTD is that it makes documents difficult to read for big sized document.

Ex: `<?xml version="1.0" ?>`

```
<!DOCTYPE bookstore [  
    <!ELEMENT bookstore (book*)>  
    <!ELEMENT book (title,author,price)>  
    <!ELEMENT title (#PCDATA)>  
    <!ELEMENT author (#PCDATA)>  
    <!ELEMENT publisher (#PCDATA)>  
    <!ELEMENT price (#PCDATA)>  
>
```

```
<bookstore>  
    <book>  
        <title>WT</title>  
        <author>Uttam Roy</author>  
        <publisher>Oxford</publisher>  
        <price>500</price>  
    </book>  
    <book>  
        <title>AJ</title>
```

```

    <author>Schildt</author>
    <publisher>TMH</publisher>
    <price>200</price>
  </book>
</bookstore>

```

2. External DTD:

Another way of connection DTD to XML document is to reference it with in XML document i.e. create separate document, put DTD information there and point to it from XML document. The general syntax for external DTD is.

Syntax: <!DOCTYPE root-element SYSTEM | PUBLIC "uri">

Where *uri* is the Uniform Resource Identifier of the *.dtd* file. This declaration states that we are going to define structure of root-element of XML document and its definition can be found from *uri* specified like *book.dtd*. both *xml* and *dtd* files should be kept in same directory.

Ex:

| <u>book.xml</u> | <u>book.dtd</u> |
|--|---|
| <pre> <?xml version="1.0" ?> <!DOCTYPE book SYSTEM "book.dtd"> <bookstore> <book> <title>WT</title> <author>Uttam Roy</author> <publisher>Oxford</publisher> <price>500</price> </book> <book> <title>AJ</title> <author>Schildt</author> <publisher>TMH</publisher> <price>200</price> </pre> | <pre> <!ELEMENT bookstore (book*)> <!ELEMENT book (title,author,price)> <!ELEMENT title (#PCDATA)> <!ELEMENT author (#PCDATA)> <!ELEMENT publisher (#PCDATA)> <!ELEMENT price (#PCDATA)> </pre> |

```
</book>
</bookstore>
```

Location of DTD need not always be local file, it can be any valid URL. Following declaration for XHTML uses PUBLIC DTD:

Syntax: `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">`

Disadvantage of using separate DTD is we have to deal with two documents.

3. Combining Internal and External DTD:

External DTD are useful for common rules for set of XML documents, whereas internal DTDs are beneficial for defining customized rules for specific document. XML allows combining both internal and external DTD for complete collection of rules for given document.

The general form of such DTD is:

Syntax: `<!DOCTYPE root-element SYSTEM | PUBLIC "uri" [DTD declarations...]>`

Ex: `<?xml version="1.0" ?>
<!DOCTYPE book SYSTEM
"book.dtd" [<!ELEMENT excl
''>
]>
<msg>Hello, World! </msg>`

DTD validation:

We'll use Java based DTD validator to validate the *bookstore.xml* against the *books.dtd*

DTDValidator.java

```
import java.io.*;
import javax.xml.parsers.*;

import org.w3c.dom.*;

public class DTDValidator
{
```

S.JAYA PRAKASH
ASSISTANT PROFESSOR

```

public static void main(String[]
args) { try {
    DocumentBuilderFactory f =
    DocumentBuilderFactory.newInstance(); f.setValidating(true); //
    Default is false
    Document d = f.newDocumentBuilder().parse(arg[0]);
}
catch (Exception e) {    System.out.println(e.toString());    }}

```

Element Type Declaration:

Elements are primary building blocks in XML document. Element type declaration set the rules for type and number of elements that may appear in XML document, what order they may appear in.

Syntax: <!ELEMENT *element-name* *type*>

Or

<!ELEMENT *element-name* (*content*)>

Here, *element-name* is name of element to be defined. The *content* could include specific rule, data or another element(s). The keyword ELEMENT must be in upper case, element names are case sensitive, all elements used in XML must be declared and same name cannot be used in multiple declarations.

In DTD, elements are classified depending upon their content as follows:

- **Standalone/Empty elements:** these elements cannot have any content and may have attributes. They can be declared using type keyword EMPTY as follows:

Syntax: <!ELEMENT *element-name* EMPTY> *Ex:* <!ELEMENT *br* EMPTY>

- **Unrestricted elements:** element with content can be created using content type as ANY. Keyword ANY indicates that *element-name* can be anything including text and other elements in any order and any number of times.

Syntax: <!ELEMENT *element-name* ANY> *Ex:* <!ELEMENT *msg* ANY>

- **Simple elements:** simple element cannot contain other elements, but contains only text.

Syntax: <!ELEMENT *element-name* (#PCDATA)> *Ex:* <!ELEMENT *author* (#PCDATA)>

This interprets that element *element-name* can have only text content. The type of text id PCDATA means Parsed Character DATA and the text will be parsed by parser and will examine for entities and markups and expanded as and when necessary. Sometimes we can use CDATA means Character DATA in place of PCDATA.

- **Compound elements:** compound elements contain other elements known as child elements.

Syntax: `<!ELEMENT element-name (child-elements-names)>`

Ex: `<!ELEMENT book (title, author, price)>`

Occurrence Indicator:

Sometimes it is necessary to specify how many times element may occur in document which is done by Occurrence Indicator. When no occurrence indicator is specified, child element must occur exactly once in XML document.

| Operator | Syntax | Description |
|-------------------|--------|--|
| None | A | Exactly one occurrence of a |
| * (Astrisck) | a* | Zero or more occurrences of a i.e. any number of times |
| + (Plus) | a+ | One or more occurrences of a i.e. at least once |
| ? (Question mark) | a? | Zero or one occurrences of a i.e. at most once |

Declaring multiple children:

Elements with multiple children are declared with names of the child elements inside parenthesis. The child elements must also be declared.

| Operator | Syntax | Description |
|----------------|--------------|---------------------------------|
| , (Sequence) | a , b | a followed by b |
| (Choice) | a b | a or b |
| () (Singleton) | (expression) | Expression is treated as a unit |

Attribute Declaration:

Attributes are used to associate name, value pairs with elements. They are useful when we want to provide some additional information about elements content. The declaration starts with ATTLIST followed by name of the element the attributes are associated with and declaration of individual declarations:

Syntax: <!ATTLIST element-name attribute-name attribute-type default-value> Ex: *<!ATTLIST employee gender CDATA 'male'/>*

Here, ATTLIST must be in upper case. The *default-value* can be any of the following:

- **Default:** in this case, attribute is optional and developer may or may not provide this attribute. When attribute is declared with default value, the value of attribute is whatever value appears as attributes content in instance document.
Ex: <!ATTLIST line width CDATA '100'/>
- **#REQUIRED:** attribute must be specified with value every time enclosing element is listed
Ex: <!ATTLIST line width CDATA #REQUIRED />
- **#FIXED:** attribute is optional and is used to ensure that the attributes are set to particular values.
Ex: <!ATTLIST line width CDATA #FIXED '50'/>
- **#IMPLIED:** similar to that of default attribute except that no default value is provided by XML
Ex: <!ATTLIST line width CDATA '#IMPLIED' />

Attribute types:

The *attribute-type* can be one among string or CDATA, tokenized and enumerated types.

- **String type:** may take any literal string as value and can be declared using keyword CDATA. An attribute of CDATA type can contain any character if it conforms to well-formed ness constraints. Some it can contains escape characters like <, > etc.
- **Tokenized type:** following tokenized types are available
 - **ID:** it is globally unique identifier of attribute, this means value of ID attribute must not appear more than once throughout the XML document and resembles primary key concept of data base.

<!ATTLIST question no ID #REQUIRED>

- **IDREF**: similar to that of foreign key concept in databases and is used to establish connections between elements. IDREF value of the attribute must refer to ID value declared

<!ATTLIST answer qno IDREF #REQUIRED>

- **IDREFS**: it allows a list of ID values separated by white spaces

<!ATTLIST answer qno IDREFS #REQUIRED>

- **NMTOKEN**: it restricts attributes value to one that is valid XML name means allows punctuation marks and white spaces.

<!ATTLIST car serial NMTOKEN #REQUIRED>

- **NMTOKENS**: can contains same characters and white spaces as NMTOKEN. White space includes one or more characters, carriage returns, line feeds, tabs

<!ATTLIST car serial NMTOKENS #REQUIRED>

- **ENTITY**: refers to external non parsed entities

<!ATTLIST car serial ENTITY #REQUIRED>

- **ENTITIES**: values of ENTITIES attribute may contain multiple entity names separated by one or more white spaces

<!ATTLIST car serial ENTITIES #REQUIRED>

- **Enumerated type**: enumerated attribute values are used when we want attribute value to be one of fixed set of values. There are two kinds of enumerated types:

- **Enumeration**: attributes are defined by a list of acceptable values from which document author must choose a value. The values are explicitly specified in declaration, separated by pipe(|)

<!ATTLIST employee gender (male|female) #REQUIRED>

- **Notation**: it allows using value that has been declared a NOTATION in DTD. Notation is used to specify format of non-XML data and common used is to describe MIME types like image/gif, image/jpeg etc.

<!NOTATION jpg SYSTEM 'image/gif'>

<!ENTITY logo SYSTEM 'logo.jpg' NDATA jpg>

<!ATTLIST photo format NOTATION (jpg) #IMPLIED>

Entity Declaration:

Entities are variables that represent other values. If a text contains entities, the value of entity is substituted by its actual value whenever the text is parsed. Entity must be defined in DTD declaration to use custom entities in XML document. Built-in entities and character entities do not require any declaration. There are two types of entity declarations: General entity and Parameter entity. Each type can be again parsed or unparsed.

- **General and Parameter entities:** General entities are used with in the document content. Parameter entities are parsed entities used with in DTD. These two types of entities use different forms of references and are recognized in different contexts. They occupy different namespaces
- **Parsed and Unparsed entities:** Parsed entity is an entity whose content is parsed and checked for well-formed ness constraint during parsing procedure. Unparsed entity is resource whose contents may or may not be text and if text may not be XML. It means there are no constraints on contents of unparsed entities. Each unparsed entity has associated notation, identified by name.

General Entity Declaration:

There are three kinds of general entity declarations:

- **Internal parsed:** an internal entity declaration has following form

Syntax: `<!ENTITY entity-name "entity-value">`

Ex: `<!ENTITY UKR "Uttam Kumar Roy">`

The entity UKR can be referred in XML document as follows:

`<author>&UKR;</author>`

This will be interpreted as : `<author>Uttam Kumar Roy</author>`

- **External parsed:** external entities allow an XML document to refer to external resource. Parse external entities refer to data that an XML parser has to parse and used for long replacement text which is kept in another file. There are two type of external parsed entities: Public and Private. Public external entities are identified by PUBLIC keyword and intended for general use. Private external entities are identified by SYSTEM keyword and are intended for use by single author or group of authors.

Syntax: <!ENTITY entity-name SYSTRM | PUBLIC "URI">
Ex: <!ENTITY author SYSTEM "author.xml">

- **External unparsed:** refer to data that an XML processor does not have to parse. For example, there are numerous ASCII text files, JPRG photographs etc. None of these are well-formed XML. Mechanism that XML suggests for embedding these files is external unparsed entity. They can be either private or public.

Syntax: <!ENTITY logo SYSTEM "logo.jpg" NDATA jpeg>

Parameter Entity Declaration:

DTD supports another kind of entity called parameter entity. It is used within DTD which allows to assign collection of elements, attributes and attribute values to name and refer them using name instead of explicitly listing them every time they are used.

- **Internal parsed entity:** it has following form

Syntax: <!ENTITY % entity-name entity-definition>

Ex: <!ENTITY % name "firstname, middlename, lastname">

This parameter entity describes portion of content model that can be referenced with in elements ind DTD. They can be referenced using entity name between percent sign (%) and semicolon (:).

Syntax: %Entity-name; *Ex:* %name;

- **External parsed entity:** These are used to link external DTDs. It may be private or public and is identified by keywords SYSTEM and PUBLIC. Private entities are intended for use by single author whereas public entities can be used by anyone.

Syntax: <!ENTITY % entity-name SYSTEM | PUBLIC "URI">

V.XML Schema:

XML Schema Definition commonly known as XSD, is a way to describe precisely the XML language. XSD check the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language. An XML document can be defined as:

- **Well-formed:** If the XML document adheres to all the general XML rules such as tags must be properly nested, opening and closing tags must be balanced and empty tags must end with '>', then it is called as *well-formed*.
- **Valid:** An XML document said to be valid when it is not only *well-formed*, but it also conforms to available XSD that specifies which tags it uses, what attributes those tags can contain and which tags can occur inside other tags, among other properties.

Limitations of Document Type Declaration (DTD)

- There is no built-in data type in DTDs
- No new data types can be created in DTDs
- The use of cardinality in DTDs is limited
- Namespaces are not supported
- DTDs provide very limited support for modularity and reuse
- We cannot put any restrictions on text content
- Defaults for elements cannot be specified
- We have very little control over mixed content
- DTDs are written in strange format and are difficult to validate

Strengths of XML Schema (XSD)

- XML schema provided much greater specification than DTDs
- They support large number of built-in data types
- They support namespaces
- They are extensible to future additions
- They support uniqueness and referential integrity constraints in much better way
- It is easier to define data restrictions

XSD Structure:

An XML XSD is kept in a separate document and then the document having extension *.xsd* and is be linked to the XML document to use it. Schema is the root element of XSD and it is always required.

Syntax: <?xml version="1.0"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 ...
 </xs:schema>

Above fragment specifies that elements and data types used in the schema are defined in "http://www.w3.org/2001/XMLSchema" namespace and these elements/data types should be prefixed with *xs*. Similarly, XSD can be linked to xml file with following syntax:

Syntax: <root-tag xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="uri">

Above fragment specifies default namespace declaration i.e. "http://www.w3.org/2001/XMLSchema-instance". This namespace is used by schema validator check that all the elements are part of this namespace. It is optional. Use *schemaLocation* attribute to specify the location of the *xsd* file.

Ex: book.xml

```
<?xml version="1.0" ?>
<bookstore xsi:schemaLocation="book.xsd" xmlns:xsi=" http://www.w3.org/2001/XMLSchema-
instance"
>
<book>
    <title>WT</title>
    <author>Uttam Roy</author>
    <publisher>Oxford</publisher>
    <price>500</price>
</book>
<book>
    <title>AJ</title>
    <author>Schildt</author>
```

S.JAYA PRAKASH
ASSISTANT PROFESSOR

```

    <publisher>TMH</publisher>
    <price>200</price>
</book>
</bookstore>
book.xsd
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string"/>
              <xs:element name="publisher" type="xs:string"/>
              <xs:element name="price" type="xs:integer"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

XSD Validation:

We'll use Java based XSD validator to validate the *bookstore.xml* against the *books.xsd*.

XSDValidator.java

```

import java.io.*;
import javax.xml.*;
import javax.xml.transform.dom.*;
import javax.xml.parsers.*;
import javax.xml.validation.*;

```

```

import org.w3c.dom.*;

public class XSDValidator {
    public static void main(String[]
        args) { try {
        SchemaFactory factory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Schema schema = factory.newSchema(new
                File(args[1])); Validator validator =
                schema.newValidator();
                DocumentBuilder
                parser=DocumentBuilderFactory.newInstance().newDocumentBuilder(); Document
                document=parser.parse(new File(args[0]));
                validator.validate(new DOMSource(document));
            }
            catch (Exception e)
            {      System.out.println("Exception: "+e.getMessage()); }
        }
}

```

Element declaration:

Elements are primary building blocks in XML document. Element type declaration can be done using <xs:element> tag with following syntax.

Syntax: <xs:element name="element-name" type="elementtype">

Ex: <xs:element name="title" type="xs:string">

Each element declaration within the XSD has mandatory attribute *name*. The value of this name attribute is the element name attribute is the element name that will appear in the XML document. Element definition may also have optional *type* attribute, which provides description of what can be contained within the element when it appears in XML document. Every XML document must have root element. This root element must be declared first in schema for conforming XML documents.

Ex: `<!xml version="1.0"?>`

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    </xs:element>
  </xsd:schema>
```

Declaring simple elements:

Simple type elements can contain only text and/or data. They cannot have child elements or attributes, and cannot be empty. Simple elements are defined as follows:

Syntax: `<xs:element name="element-name" type="element-type">`

Ex: `<xs:element name="title" type="xs:string"/>`

The value of *type* attributes specifies an elements content type and can be any simple type.

This attribute can be any complex type.

- **Default Value:** Simple Element can have default value that specifies the default content to be used when not content is specified. When an element is declared with default value, the value of the element is whatever value appears as elements content in instance document. Following example illustrates this:

```
<xs:element name="gender" type="xs:boolean" default="true" />
```

- **Fixed Value:** Simple Element can also have optional fixed value. Fixed attribute is used to ensure that elements content is always set to particular value. Consider the following syntax:

```
<xs:element name="branch" type="xs:string" fixed="IT" />
```

- **Occurrence indicators:** an element have two optional attributes : minOccurs and maxOccurs. They are used to specify the number of times an element can occur in XML document.

- **minOccurs:** this attribute specifies minimum number of times an element can occur.

The following is example of usage of this attribute:

```
<xs:element name="option" type="xs:string" minOccurs="0"/>
```

- **maxOccurs:** this attribute specifies maximum number of times an element can occur.

The declaration of element will be as follows:

```
<xs:element name="option" type="xs:string" maxOccurs="10"/>
```

| Schema | DTD | Meaning |
|---|------|--------------|
| minOccurs='0', maxOccurs='unbounded' | * | Zero or more |
| minOccurs='1', maxOccurs='unbounded' | + | One or more |
| Minoccurs='0' | ? | Optional |
| None | None | Exactly one |

Declaring complex elements:

Complex types can be named or can be anonymous. They are associated with complex elements in the same manner, typically using a type definition and an element declaration. By default, complex type elements have complex content i.e. they have child elements. Complex type elements can be limited to having simple content i.e. they contain only text. General form of element declaration is:

Syntax: `<xs:complexType name="complex-type-name"><xs:sequence>`

`</xs:sequence></xs:complexType>`

Ex: `<xs:complexType name="sName"><xs:sequence>`

`<xs:element name="first" type="xs:string"/>`

`<xs:element name="middle" type="xs:string"/>`

`<xs:element name="lase" type="xs:string"/>`

`</xs:sequence></xs:complexType>`

Attribute declaration:

Attributes are used to describe properties of an element. Attributes themselves are always declared as simple types as follows:

Syntax: `<xs:attribute name"attribute-name" type="attribute-type">`

Ex: `<xs:attribute name="id" type="xs:string"/>`

Simple types cannot have attributes. Element that have attributes are complex types. So, attributes declaration always occurs as part of complex type declaration, immediately after its content model.

Ex: `<xs:complexType name="sName"><xs:sequence>`
`<xs:element name="first" type="xs:string"/>`
`<xs:element name="middle" type="xs:string"/>`
`<xs:element name="lase" type="xs:string"/>`
`</xs:sequence>`
`<xs:attribute name="id" type="xs:string"/>`
`</xs:complexType>`

A part from this simple definition, there can be additional specifications for attributes:

- **Attribute element properties:**

- **use:** possible values are optional, required and prohibited.

`<xs:attribute name="id" type="xs:string" use="required"/>`

- **default:** this specifies the value to be used if attribute is not specified

`<xs:attribute name="gender" type="xs:boolean" default="false"/>`

- **fixed:** it specifies that attribute, if it appears must always have fixed value specified. If the attribute does not appear, the schema processor will provide attribute with value specified here.

`<xs:attribute name="unit" type="xs:boolean" default="rpm"/>`

- **Order Indicators**

- **All:** Child elements can occur in any order.

`<xs:all>`

`<xs:element name="first" type="xs:string"/>`

`<xs:element name="middle" type="xs:string"/>`

`<xs:element name="last" type="xs:string"/>`

`</xs:all>`

- **Choice:** Only one of the child element can occur.

`<xs:choice>`

`<xs:element name="first" type="xs:string"/>`

`<xs:element name="middle" type="xs:string"/>`

`<xs:element name="last" type="xs:string"/>`

`</xs:choice>`

- **Sequence:** Child element can occur only in specified order.

```
<xs:sequence>
  <xs:element name="first" type="xs:string"/>
  <xs:element name="middle" type="xs:string"/>
  <xs:element name="last" type="xs:string"/>
</xs:sequence>
```

- **Occurrence Indicators**

- **maxOccurs** - Child element can occur only maxOccurs number of times.
- **minOccurs** - Child element must occur minOccurs number of times.

- **Group Indicators**

- **Group**: a set of related elements can be created using this indicator. the general form for creating an element group is as follows:

Syntax: `<xs:group name="group-name"> ... </xs:group>`

Ex: `<xs:group name="personInfo">`

`<xs:element name="first" type="xs:string"/>`

`<xs:element name="middle" type="xs:string"/>`

`<xs:element name="last" type="xs:string"/>`

`</xs:group>`

- **attributeGroup**: XML Schema provides this element, which is used to group a set of attributes declarations so that they can be incorporated into complex types definitions with syntax:

Syntax: `<xs:attributeGroup name="group-name"> ... </xs:attributeGroup>`

Ex: `<xs:attributeGroup name="personInfo">`

`<xs:element name="first" type="xs:string"/>`

`<xs:element name="middle" type="xs:string"/>`

`<xs:element name="last" type="xs:string"/>`

`</xs:attributeGroup>`

Annotations declaration:

XML schema provides three annotation elements for documentation purposes in XML schema instance. An annotation is represented by `<annotation>` element which typically appears at the beginning of most schemas. However, it can appear inside any complex element definition. It can contain only two elements `<appinfo>` and `<documentation>` any number of times.

Following is an example:

```
<xs:annotation>  
  <xs:documentation> <author>Uttam Roy</author></xs:documentation>  
  <xs:appinfo><version>2.1</version></xs:appinfo>  
</xs:annotation>
```

XML Schema data types:

An element is limited by its type. Depending upon content model, elements are categorized as Simple or Complex type. A simple type can further be divided into three types: Atomic, List and Union. XML schema 1.0 specification provides about 46 built in data types. Some of the built-in data types are as follows:

XSD String Data Types:

String data types are used to represent characters in the XML documents.

- **<xs:string>:** The <xs:string> data type can take characters, line feeds, carriage returns, and tab characters. XML processor does not replace line feeds, carriage returns, and tab characters in the content with space and keep them intact. For example, multiple spaces or tabs are preserved during display.

Syntax: <xs:element name="element-name" type="xs:string"/>

Ex: < xs:element name="sname" type="xs:string"/>

- **<xs:token>:** The <xs:token> data type is derived from <string> data type and can take characters, line feeds, carriage returns, and tab characters. XML processor removes line feeds, carriage returns, and tab characters in the content and keep them intact. For example, multiple spaces or tabs are removed during display.

Syntax: <xs:element name="element-name" type="xs:token"/>

Following is the list of commonly used data types which are derived from <string> data type.

- **ID:** Represents the ID attribute in XML and is used in schema attributes.
- **IDREF:** Represents the IDREF attribute in XML and is used in schema attributes.

S.JAYA PRAKASH
ASSISTANT PROFESSOR

- **Language:** Represents a valid language id
- **Name:** Represents a valid XML name
- **NMTOKEN:** Represents a NMTOKEN attribute in XML and is used in schema attributes.
- **normalizedString:** Represents a string that does not contain line feeds, carriage returns, or tabs.
- **String:** Represents a string that can contain line feeds, carriage returns, or tabs.
- **Token:** Represents a string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces

XSD Date & Time Data Types:

Date and Time data types are used to represent date and time in the XML documents.

- **<xs:date> data type:** The <xs:date> data type is used to represent date in YYYY-MM-DD format. Each component is required. **YYYY-** represents year, **MM-** represents month, **DD-** represents day

Syntax: <xs:element name="birthdate" type="xs:date"/>

Ex: <birthdate>1980-03-23</birthdate>

- **<xs:time> data type:** The <xs:time> data type is used to represent time in hh:mm:ss format. Each component is required. **hh-** represents hours, **mm-** represents minutes, **ss-** represents seconds

Syntax: <xs:element name="startTime" type="xs:time"/>

Ex: <startTime>10:20:15</startTime>

- **<xs:datetime> data type:** The <xs:datetime> data type is used to represent date and time in YYYY-MM-DDThh:mm:ss format. Each component is required. **YYYY-** represents year, **MM-** represents month, **DD-** represents day, **T-** represents start of time section, **hh-** represents hours, **mm-** represents minutes, **ss-** represents seconds

Syntax: <xs:element name="startTime" type="xs:datetime"/>

Ex: <startTime>1980-03-23T10:20:15</startTime>

- **<xs:duration> data type:** The <xs:duration> data type is used to represent time interval in PnYnMnDTnHnMnS format. Each component is optional except P. **P-** represents year, **nY-** represents month, **nM-** represents day, **nD-** represents day, **T-** represents start of time section, **nH-** represents hours, **nM-** represents minutes, **nS-** represents seconds.

Syntax: <xs:element name="period" type="xs:duration"/>

Ex: <period>P6Y3M10DT15H</period>

Following is the list of commonly used date data types.

- **Date:** Represents a date value
- **dateTime:** Represents a date and time value
- **duration:** Represents a time interval
- **gDay:** Represents a part of a date as the day (DD)
- **gMonth:** Represents a part of a date as the month (MM)
- **gMonthDay:** Represents a part of a date as the month and day (MM-DD)
- **gYear:** Represents a part of a date as the year (YYYY)
- **gYearMonth:** Represents a part of a date as the year and month (YYYY-MM)
- **time:** Represents a time value

XSD Numeric Data Types:

Numeric data types are used to represent numbers in the XML documents.

- **<xs:decimal> data type:** The <xs:decimal> data type is used to represent numeric values. It support decimal numbers upto 18 digits.

Syntax: <xs:element name="score" type="xs:decimal"/>

Ex: <score>9.12</score>

- **<xs:integer> data type:** The <xs:integer> data type is used to represent integer values.

Syntax: <xs:element name="score" type="xs:integer"/>

Ex: <score>9</score>

Following is the list of commonly used numeric data types.

- **Byte:** A signed 8 bit integer
- **Decimal:** A decimal value
- **Int:** A signed 32 bit integer
- **Integer:** An integer value
- **Long:** A signed 64 bit integer
- **negativeInteger:** An integer having only negative values (...,-2,-1)
- **nonNegativeInteger:** An integer having only non-negative values (0,1,2,..)
- **nonPositiveInteger:** An integer having only non-positive values (...,-2,-1,0)
- **positiveInteger:** An integer having only positive values (1,2,..)
- **short:** A signed 16 bit integer
- **unsignedLong:** An unsigned 64 bit integer
- **unsignedInt:** An unsigned 32 bit integer
- **unsignedShort:** An unsigned 16 bit integer
- **unsignedByte:** An unsigned 8 bit integer

XSD Miscellaneous Data Types:

Other Important Miscellaneous data types used are boolean, binary and anyURI.

- **<xs:boolean> data type:** The <xs:boolean> data type is used to represent true, false, 1 (for true) or 0 (for false) value.

Syntax: <xs:element name="pass" type="xs:boolean"/>

Ex: <pass>false</pass>

- **Binary data types:** The Binary data types are used to represent binary values. Two binary types are common in use. **base64Binary**- represents base64 encoded binary data, **hexBinary** - represents hexadecimal encoded binary data

Syntax: <xs:element name="blob" type="xs:hexBinary"/>

Ex: <blob>9FEEF</blob>

- **<xs:anyURI> data type:** The <xs:anyURI> data type is used to represent URI.

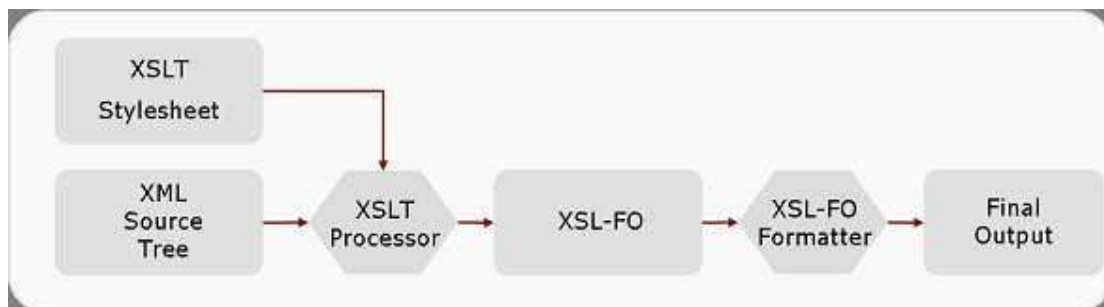
Syntax: <xs:attribute name="resource" type="xs:anyURI"/>

Ex: <image resource="http://www.tutorialspoint.com/images/smiley.jpg" />

VI. eXtensible Stylesheet Language Transformation(XSLT):

XML documents contain self-describing and structured data. The set of tags and their structure varies widely in different applications. Web browsers cannot display such non-HTML files as they have no prior knowledge about the meaning of set of tags used in different XML documents. Users may also want to generate new XML documents from one or more existing XML documents for processing or sharing of data between different applications. One possible solution is to generate separate XML document such that the former contains only insensitive data. XSLT comes into play in this scenario.

XSLT, Extensible Stylesheet Language Transformations provides the ability to transform XML data from one format to another automatically. An XSLT stylesheet is used to define the transformation rules to be applied on the target XML document. XSLT stylesheet is written in XML format. XSLT Processor takes the XSLT stylesheet and apply the transformation rules on the target XML document and then it generates a formatted document in form of XML, HTML or text format. This formatted document then is utilized by XSLT formatter to generate the actual output which is to be displayed to the end user.



Following are the main parts of XSL.

- **XSLT** - used to transform XML document into various other types of document.
- **XPath** - used to navigate XML document.
- **XSL-FO** - used to format XML document.

In general following tasks can be performed using XSLT: Constant text generation, reformatting of information, sensitive information suppression, adding new information, copying information and sorting document with respect to a criteria.

Advantages

- Independent of programming. Transformations are written in a separate xsl file which is again an XML document.
- Output can be altered by simply modifying the transformations in xsl file. No need to change in any code. So Web designers can edit stylesheet and can see the change in output quickly.

1. Stylesheet structure:

XSLT files are themselves XML documents and hence must follow the well-formedness constraints. The W3C defined the exact syntax of an XSLT 2.0 document by XML schema. XSLT file starts with XML declaration. Every XSLT file must have either <stylesheet> or <transform> as root element. Following are simple structure of XSLT document:

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">
...
</xsl:stylesheet>
```

Or

```
<?xml version="1.0"?>
<xsl:transform version="2.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">
...
</xsl:transform >
```

These elements must have the attribute *version* and namespace attribute *xmlns*. Version attribute indicates version of XSLT being used. Namespace attribute distinguishes XSLT elements from other elements. There are different ways to apply XSLT document to XML document. One way to add link to XML document which points to actual XSLT files and lets the browsers do transformation with following declaration:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="URI">
<root> ... </root>
```

| | |
|--|---|
| <pre> <i>students.xml</i> <?xml version="1.0"?> <?xml-stylesheet type="text/xsl" href="students.xsl"> <class> ... </class> </pre> | <pre> <i>students.xsl</i> <?xml version="1.0"?> <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform" > ... </xsl:stylesheet> </pre> |
|--|---|

2. XSLT Elements:

An XSLT file contains elements, which instruct processor how an XML document is to be transformed. It may contain elements that are not defined by XSLT. In such cases, XSLT processor does not process these non-XSLT elements and add them to the output in the same order they occurred in the source XSLT document. This means that the transformed XML document may use original mark-ups as well as new mark-ups.

| Element | Description |
|-----------------|--|
| stylesheet | Defines the root element of a style sheet |
| transform | Defines the root element of a style sheet |
| template | Rules to apply when a specified node is matched |
| apply-templates | Applies a template rule to the current element or to the current element's child nodes |
| call-template | Calls a named template |
| element | Creates an element node in the output document |
| variable | Declares a local or global variable |
| param | Declares a local or global parameter |
| value-of | Extracts the value of a selected node |
| attribute | Adds an attribute |
| attribute-set | Defines a named set of attributes |
| if | Contains a template that will be applied only if a specified condition is true |
| choose | Used in conjunction with <when> and <otherwise> to express multiple conditional tests |
| when | Specifies an action for the <choose> element |

| | |
|------------------------|--|
| for-each | Loops through each node in a specified node set |
| import | Imports the contents of one style sheet into another. Note: An imported style sheet has lower precedence than the importing style sheet |
| include | Includes the contents of one style sheet into another. Note: An included style sheet has the same precedence as the including style sheet |
| sort | Sorts the output |
| processing-instruction | Writes a processing instruction to the output |
| comment | Creates a comment node in the result tree |
| copy | Creates a copy of the current node (without child nodes and attributes) |
| copy-of | Creates a copy of the current node (with child nodes and attributes) |

3. XSLT templates:

An XSLT document is all about template rules. A template specifies rule and instruction, which is executed when rule matches. The rule is specified by XSLT <template> element. It has attribute *match*, which specifies pattern. The value of match attribute is subset of expression.

Syntax: <xsl:template match="expression">

...

</xsl:template>

Ex: <xsl:template match="/">

<h1>Hello! World.</h1>

</xsl:template>

XSLT document contain single template rule. It has match attribute with expression “/”, which means the document root of any XML document. This instruction with in this template specifies the string *Hello! World* has to be added to the output and the resulting document obtained is as follows:

<html> <body><h1>Hello! World.</h1></body> </html>

Applying templates:

In general, if a node matches with template pattern, the templates action part is processed. It is also possible to instruct XSLT processor to process other template rules if any. This is done using <apply-templates> element with following syntax:

<xsl:template match="/">

```
<xsl:apply-templates/>
</xsl:template>
```

This example states that whenever document root is encountered, XSLT processor has to process all templates that match with document roots children roots. The XSLT engine in turn, compares each child element of document root against templates in style sheet and if match is found, it processes the corresponding template.

Processing Sequence and default templates:

When XSLT processor is supplied XML document for transformation using XSLT document, it first creates document tree. Processing always starts from document root of this tree. So, XSLT processor looks for template for it. If no template is found for document root, XSLT processor provides default templates. This default template for document root looks like this:

```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>
```

The behavior of default template for any element node looks as follows:

```
<xsl:template match="*">
  <xsl:apply-templates/>
</xsl:template>
```

Default template for text nodes as follows:

```
<xsl:template match="text()">
  <xsl:apply-templates/>
</xsl:template>
```

Default templates and their behavior:

- **Root:** process template for its children
- **Element:** process templates for its children
- **Attribute:** output attribute name and value
- **Text:** output text value
- **Processing instruction:** do nothing
- **Comment:** do nothing

Named templates:

XSLT named templates resemble the functions in any procedural programming language. The <template> element has *name* attribute, which can be used to give name to template. Once template is created this way, it can be called by using <call-template> element and specifying its name.

```
<xsl:template match="/">
    <xsl:call-template name="header"/>
</xsl:template>

<xsl:template name="header">
    <title>XSLT</title>
</xsl:template>
```

4. Selecting values:

The value of a node can also be added using <value-of> element. Value of node depends on type of the node. For example, the value of text node is the text itself, whereas the value of element node is concatenation of values of all text descendants. If multiple nodes are selected by select attribute, value is concatenation of values of those selected attributes. Consider simple XML document:

```
<book>
    <title>Web Technologies</title>
</book>
```

One can now extract the value of title element using <value-of> element as follows:

```
<xsl:template match="/">
    Title: <xsl:value-of select="book/title"/>
</xsl:template>
```

This XSLT file, on applying previous XML document produces following result:

Title: Web Technologies

Values of different node types:

- **Text:** text of node
- **Element:** concatenation of values of all text descendants
- **Attribute:** attribute value without quotation marks
- **Namespace:** the URI of the namespace
- **Comment:** anything between <!--and -->
- **Processing instruction:** anything between <? and ?>

XSLT has another element <copy-of>, which returns all selected elements including nested elements and text. Consider the following XSLT document.

```
<xsl:template match="/">
  <xsl:copy-of select="."/>
</xsl:template>
```

When we apply this XSLT document to any XML document, it produces the same XML document. This is because, when root element (/) is selected, <copy-of> copies root element together with all child elements recursively.

5. Variables and Parameters:

Named template resembles the functions in any procedural programming language. Like function, named templates may accept argument. Formal parameters are declared with in template using <param> element as follows:

```
<xsl:template name="add">
  <xsl:param name="a"/>
  <xsl:param name="b"/>
  <xsl:value-of select="$a+$b"/>
</xsl:template>
```

This example defines named template *add*, which takes two parameters *a* and *b*. The purpose of this template is to add two arguments taken and produce the result to output. Arguments can then be passed to template using <with-param> element during template call.

```
<xsl:call-template name="add">
  <xsl:with-param name="a" select="2"/>
  <xsl:with-param name="b" select="4"/>
</xsl:call-template>
```

```
</xsl:call-template>
```

This code calls template add with parameters 2 and 4. If this XSLT applied to XML document the output will be 6. The scope of formal is within the template only. XSLT allows declaring and using variable. Consider the following code:

```
<xsl:template>
    <xsl:variable name="a">4</xsl:variable>
    <xsl:variable name="b">6</xsl:variable>
    <xsl:value-of select="$a+$b"/>
</xsl:template>
```

6. Conditional Processing:

There are two types of branching constructs in XSLT: <if> and <choose>

Using if:

XSLT <if> element has attribute *test*, which takes Boolean expression. If the effective Boolean value of this expression is evaluated to true, the action under <if> construct is followed. The general syntax of <if> construct is as follows:

```
<xsl:if test="condition">
...
</xsl:if>
```

The following extracts information about only that book having title as "Web Technologies":

```
<xsl:template match="//book">
    <xsl:if test="@title='Web Technologies'">
        Author: <xsl:value-of
        select="@author"/> Price:
        <xsl:value-of
        select="@price"/>
    </xsl:if>
</xsl:template>
```


Using choose:

XSLT <choose> element allows us to select particular condition among set of conditions specified by <when> element. The general format of <choose> construct is:

```
<xsl:choose>
  <xsl:when text="expression1">..</xsl:when>
  <xsl:when text="expression2">..</xsl:when>
  ...
  <xsl:when text="expressionN">..</xsl:when>
  <xsl:otherwise> ...</xsl:otherwise>
</xsl:choose>
```

Consider the following XML file *result.xml*, containing marks of different students:

```
<result>
  <student><rollno>01</rollno><marks>80</marks></student>
  <student><rollno>02</rollno><marks>70</marks></student>
  <student><rollno>03</rollno><marks>60</marks></student>
  <student><rollno>04</rollno><marks>55</marks></student>
  <student><rollno>05</rollno><marks>77</marks></student>
</result>
```

The following XSLT document displays results of the students:

```
<xsl:choose>
  <xsl:when test="marks > 80 and marks <= 100">A Grade</xsl:when>
  <xsl:when test="marks > 70 and marks <= 80">B Grade</xsl:when>
  <xsl:when test="marks > 60 and marks <= 70">C Grade</xsl:when>
  <xsl:when test="marks <=60">D Grade</xsl:when>
</xsl:choose>
```

Repetition:

XSLT allows `<for-each>` construct, which can be used to process set of instructions repeatedly for different items in sequence. The attribute *select* evaluates sequence of nodes. For each of the elements in this sequence, instruction under `<for-each>` element are processed. Consider the following XML file *result.xml*, containing marks of different students:

```
<result>
  <student><rollno>01</rollno><marks>80</marks></student>
  <student><rollno>02</rollno><marks>70</marks></student>
  <student><rollno>03</rollno><marks>60</marks></student>
  <student><rollno>04</rollno><marks>55</marks></student>
  <student><rollno>05</rollno><marks>77</marks></student>
</result>
```

The following XSLT document displays results of the students:

```
<xsl:for-each select="result">
  Roll No: <xsl:value-of select
    ="rollno"/><br> Marks: <xsl:value-of
    select ="marks"/><br>
</xsl:for-each>
```

7. Creating nodes and Sequences:

XSLT allows to directly creating custom nodes such as element node, text nodes etc. or sequences of nodes and atomic values that appear in output.

- **Creating element nodes:**

An element node is created using `<element>` tag. The content of created element is whatever is generated between the starting and closing of `<element>` tag. If an element has attributes, they are declared using `<attribute>` tag described in the next section.

```
<xsl:element name="msg">
  Hello world!
</xsl:element>
```

- **Create attribute node:**

An attributes of an element is created using enclosed `<attribute>` tag. The mandatory attribute *name* specifies name of the generated attributes. The value is indicated by content of `<attribute>` element.

```
<xsl:element name="msg">
  <xsl:attribute name="lang">en</xsl:attribute>
  Hello world!
</xsl:element>
```

This code segment creates the element *msg* with attribute *lang* as follows:

```
<msg lang="en">Hellow World!</msg>
```

- **Create text nodes:**

Generally, XSLT processor outputs text that appears in the stylesheet. However, extra white spaces are not provided in such case. Secondly, special characters such as `<` and `&` are represented in text by escape character sequence `<` and `&`; respectively. For this reason, it provides `<text>` element to add literal text to result with following syntax:

```
<xsl:text> Hello World &amp;</xsl:text>
```

- **Creating document node:**

XSLT allows creating new document node using `<document>` element. For example, following code create temporary document node, which is stored in variable named "tempTree".

```
<xsl:variable name="tempTree" as="document-node()">
  <xsl:document> <xsl:apply-templates/> </xsl:document>
</xsl:variable>
```

- **Creating processing instructions:**

Processing instruction is added in the result using `<processing-instruction>` element. The most popular use of this element is to insert the `<stylesheet>` element in output HTML/XML document with syntax.

```
<xsl:processing-instruction name="xml-stylesheet">
  <xsl:text> href="sort.xml" type="text/xsl"</xsl:text>
</xsl:processing-instruction>
```

- **Creating comments:**

Comment is added using `<comment>` element as follows:

```
<xsl:comment>This is XSLT document</xsl:document>
```

8. Grouping nodes:

XSLT allows us to group related items based on common values. Consider the following XML document.

```
<result>
  <student><rollno>01</rollno><marks>80</marks><dept>IT</dept></student>
  <student><rollno>02</rollno><marks>70</marks><dept>IT</dept></student>
  <student><rollno>03</rollno><marks>60</marks><dept>CSE</dept></student>
  <student><rollno>04</rollno><marks>55</marks><dept>IT</dept></student>
  <student><rollno>05</rollno><marks>77</marks><dept>CSE</dept></student>
</result>
```

The following XSLT document displays results of the students as groups by dept:

```
<xsl:template match="/result">
  <xsl:for-each-group select="student" group-by="@dept">
    <xsl:value-of select="current-grouping-key()" />
    <xsl:for-each select="current-group()">
      <xsl:value-of select="@rollno" />
    </xsl:for-each>
  </xsl:template>
```

This enumerates group items based either on common value of grouping key or pattern specified by group-by attribute. The current-group () function returns the current group item in the iteration and current-grouping-key () returns common key of current group.

9. Sorting nodes:

We can sort group of similar elements using <sort> element. The attributes of the <sort> element describe how to perform sorting. For example, sorting can be done alphabetically or numerically or in increasing or decreasing order. The attribute select is used to specify sorting key. The order attribute specifies order and can have values ascending or descending. The type of data to be sorted can be specified using attribute data-type. Following example sorts list of student respect to their marks.

```

<table><xsl:for-each select="/result">
  <xsl:sort select="marks" data-type="number"/>
  <tr><td><xsl:value-of select="rollno"/></td>
  <td><xsl:value-of select="marks"/></td>
  <td><xsl:value-of select="dept"/></td></tr>
</xsl:for-each></table>

```

10. Functions:

XSLT also allows custom functions to be defined in stylesheet. A function is defined using <function> element. It has attribute name, which specifies the name of the function. Once function is defined, it can be called from any expression. The function name must have prefix. This is required to avoid conflict with any function from default namespace. A prefix cannot be bound to reserved namespace.

```

<xsl:function name="f:fact">
  <xsl:param name="n">
    <xsl:value-of select="if ($n le 1) then 1 else $n*f:fact($n-1)"/>
  </xsl:param>
</xsl:function>
<xsl:template match="/">
  <xsl:value-of select="f:fact(3)"/>
</xsl:template>

```

11. Copying nodes:

The <copy> element copies the current node to the output. If the node is an element node, its namespace nodes are copied automatically, but attributes and children of element nodes are not copied automatically. Consider the simple XML document:

```

<result>
  <student><rollno>01</rollno><marks>80</marks><dept>IT</dept></student>
  <student><rollno>02</rollno><marks>70</marks><dept>IT</dept></student>
  <student><rollno>03</rollno><marks>60</marks><dept>CSE</dept></student>
  <student><rollno>04</rollno><marks>55</marks><dept>IT</dept></student>

```

S.JAYA PRAKASH
ASSISTANT PROFESSOR

```
<student><rollno>05</rollno><marks>77</marks><dept>CSE</dept></student>  
</result>
```

Now consider following XSLT document:

```
<xsl:template match="/student">  
  <xsl:copy />  
</xsl:template>
```

12. Numbering:

The <number> element allows inserting and formatting number into the result tree.

```
<xsl:template match="/result">  
  <xsl:for-each-group select="student" group-by="@dept">  
    <xsl:number value="position()" />  
    <xsl:value-of select="current-grouping-key()" />  
    <xsl:for-each select="current-group()">  
      <xsl:number value="position()" />  
      <xsl:value-of select="@rollno" />  
    </xsl:for-each>  
  </xsl:template>
```

VII. Document Object Model (DOM):

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 accessed and manipulated. 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. Main objectives of DOM are accessing the elements of document, deleting the elements of documents and changing the elements of document.

DOM models document as hierarchical structure consisting of different kinds of nodes. Each of these nodes represents specific portion of the document. Some kind of nodes may have children of different types. Some nodes cannot have anything below it in the hierarchical structure and are leaf nodes. 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. The DOM is separated into 3 different parts/levels:

1. Core DOM: standard model for any structured document.
2. HTML DOM: standard model for HTML documents.
3. XML DOM: standard model for XML documents.

1. Core DOM:

This portion defines the basic set of interfaces and objects for any structured document.

2. HTML DOM:

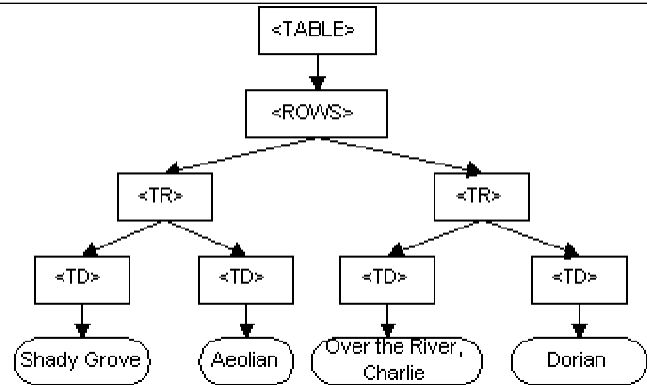
The HTML Document Object Model (DOM) is a programming API for HTML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. With the Document Object Model, programmers can create and build documents, navigate their structure, and add, modify, or delete elements and content.

Anything found in an HTML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the internal subset and external subset have not yet been specified.

```

<TABLE>
<ROWS>
<TR>
  <TD>Shady Grove</TD>
  <TD>Aeolian</TD>
</TR>
<TR>
  <TD>Over the River, Charlie</TD>
  <TD>Dorian</TD>
</TR>
</ROWS>
</TABLE>

```



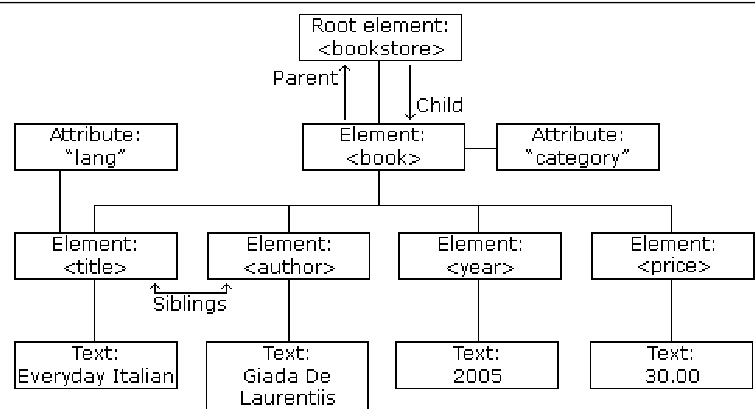
3. XML DOM:

According to the DOM, everything in an XML document is a Node. The DOM says: The entire document is a document node, Every XML element is an element node, the text in the XML elements are text nodes, every attribute is an attribute node, and comments are comment nodes.

```

<bookstore>
<book category="cooking">
<title lang="en">Everday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
</book>
</bookstore>

```



The root node in the XML above is named <bookstore>. All other nodes in the document are contained within <bookstore>. The root node <bookstore> holds four <book> nodes. The first <book> node holds four nodes: <title>, <author>, <year>, and <price>, which contains one text node each, "Everyday Italian", "Giada De Laurentiis", "2005", and "30.00". The XML DOM views an XML document as a tree- structure. The tree structure is called a node-tree. All nodes can be accessed through the tree. Their contents can be modified or deleted, and new elements can be created.

The node tree shows the set of nodes, and the connections between them. The tree starts at the root node and branches out to the text nodes at the lowest level of the tree. The nodes in the node tree have a hierarchical relationship to each other. The terms parent, child, and sibling are used to describe the relationships. Parent nodes have children. Children on the same level are called siblings (brothers or sisters).

In a node tree, the top node is called the root, every node except the root has exactly one parent node, a node can have any number of children, a leaf is a node with no children, and siblings are nodes with the same parent.

Using XML processors:

Parsing XML refers to going through XML document to access data or to modify data in one or other way. XML Parser provides way how to access or modify data present in an XML document. Java provides multiple options to parse XML document. Following are various types of parsers which are commonly used to parse XML documents.

- **Dom Parser:** Parses the document by loading the complete contents of the document and creating its complete hierarchical tree in memory.
- **SAX Parser:** Parses the document on event based triggers. Does not load the complete document into the memory.

Difference between DOM and SAX:

| DOM | SAX |
|---|--|
| DOM is a tree based parsing method | SAX is an event based parsing method |
| We can insert or delete a node | We can insert or delete a node |
| Traverse in any direction | Top to bottom traversing |
| Stores the entire XML document in to memory before processing | Parses node by node |
| Occupies more memory | Doesn't store the XML in memory |
| DOM preserves comments | SAX doesn't preserve comments. |
| import javax.xml.parsers.*; import org.w3c.dom.*; | import javax.xml.sax.*; import org.xml.sax.helpers.*; |

1. Java DOM Parser:

The Document Object Model is an official recommendation of the World Wide Web Consortium (W3C). It defines an interface that enables programs to access and update the style, structure, and contents of XML documents. XML parsers that support the DOM implement that interface. In order to use this, we need to know a lot about the structure of a document, need to move parts of the document around and need to use the information in the document more than once. When we parse an XML document with a DOM parser, we get back a tree structure that contains all of the elements of your document. The DOM provides a variety of functions you can use to examine the contents and structure of the document.

DOM interfaces:

The DOM defines several Java interfaces. Here are the most common interfaces:

- **Node:** The base data type of the DOM.
- **Element:** The vast majority of the objects you'll deal with are Elements.
- **Attr:** Represents an attribute of an element.
- **Text:** The actual content of an Element or Attr.
- **Document:** Represents entire XML document, a Document object is often referred to as a DOM tree.

Common DOM methods:

When you are working with the DOM, there are several methods you'll use often:

- **Document.getDocumentElement()** - Returns the root element of the document.
- **Node.getFirstChild()** - Returns the first child of a given Node.
- **Node.getLastChild()** - Returns the last child of a given Node.
- **Node.getNextSibling()** - These methods return the next sibling of a given Node.
- **Node.getPreviousSibling()** - These methods return the previous sibling of a given Node.
- **Node.getAttribute(attrName)** - For a given Node, returns the attribute with the requested name.

Steps to Use DOM parser:

Following are the steps used while parsing a document using DOM Parser.

1. *Import XML-related packages:*

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;
```

2. **Create a DocumentBuilder**

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

3. **Create a Document from a file or stream**

```
StringBuilder strb = new StringBuilder();
strb.append("<?xml version='1.0'?> <bookstore> </bookstore>");
ByteArrayInputStream input = new ByteArrayInputStream(strb.toString().getBytes("UTF-8"));
Document doc = builder.parse(input);
```

4. **Extract the root element**

```
Element root = document.getDocumentElement();
```

5. **Examine attributes**

```
getAttribute("attributeName");
getAttributes();
```

6. **Examine sub-elements**

```
getElementsByTagName("subelementName");
getChildNodes();
```

Example for using of DOM parser:

class.xml

```
<?xml version="1.0"?>
<class>
    <student rollno="393">
        <firstname>dinkar</firstname>
        <lastname>kad</lastname>
        <nickname>dinkar</nickname>
        <marks>85</marks>
    </student>
    <student rollno="493">
        <firstname>Vaneet</firstname>
        <lastname>Gupta</lastname>
        <nickname>vinni</nickname>
        <marks>95</marks>
    </student>
</class>
```

DomParserDemo.java

```
import java.io.File;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
public class DomParserDemo
{
public static void main(String[] args){
    try {
        File inputFile = new File("input.txt");
```

```

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();
System.out.println("Root element :" + doc.getDocumentElement().getNodeName());
NodeList nList = doc.getElementsByTagName("student");
System.out.println(".....");
for (int temp = 0; temp < nList.getLength(); temp++)
    {
        Node nNode = nList.item(temp);
System.out.println("\nCurrent Element :" + nNode.getNodeName());
        if (nNode.getNodeType() == Node.ELEMENT_NODE)
            {
                Element eElement = (Element) nNode;
                    System.out.println("Student roll no : " + eElement.getAttribute("rollno"));
System.out.println("FirstName:" + eElement.getElementsByTagName("firstname").item(0).getTextContent());
System.out.println("LastName:" + eElement.getElementsByTagName("lastname").item(0).getTextContent());
System.out.println("NickName:" + eElement.getElementsByTagName("nickname").item(0).getTextContent());
System.out.println("Marks" + eElement.getElementsByTagName("marks").item(0).getTextContent());
                }
            }
    }
catch (Exception e) {
e.printStackTrace();
}
}
}

```

2. Java SAX Parser:

SAX (the Simple API for XML) is an event-based parser for xml documents. Unlike a DOM parser, a SAX parser creates no parse tree. SAX is a streaming interface for XML, which means that applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element. Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document. Tokens are processed in the same order that they appear in the document. Reports the application program the nature of tokens that the parser has encountered as they occur. The application program provides an "event" handler that must be registered with the parser. As the tokens are identified, callback methods in the handler are invoked with the relevant information

ContentHandler Interface

This interface specifies the callback methods that the SAX parser uses to notify an application program of the components of the XML document that it has seen.

- **void startDocument()** - Called at the beginning of a document.
- **void endDocument()** - Called at the end of a document.
- **void startElement(String uri, String localName, String qName, Attributes atts)** - Called at the beginning of an element.
- **void endElement(String uri, String localName, String qName)** - Called at the end of an element.
- **void characters(char[] ch, int start, int length)** - Called when character data is encountered.
- **void ignorableWhitespace(char[] ch, int start, int length)** - Called when a DTD is present and ignorable whitespace is encountered.
- **void processingInstruction(String target, String data)** - Called when a processing instruction is recognized.

- **void setDocumentLocator(Locator locator)** - Provides a Locator that can be used to identify positions in the document.
- **void skippedEntity(String name)** - Called when an unresolved entity is encountered.
- **void startPrefixMapping(String prefix, String uri)** - Called when a new namespace mapping is defined.
- **void endPrefixMapping(String prefix)** - Called when a namespace definition ends its scope.

Attributes Interface

This interface specifies methods for processing the attributes connected to an element.

- **int getLength()** - Returns number of attributes.
- **String getQName(int index)**
- **String getValue(int index)**
- **String getValue(String qname)**

Example for using of SAX parser: class.xml

```
<?xml version="1.0"?>
<class>
  <student rollno="393">
    <firstname>dinkar</firstname>
    <lastname>kad</lastname>
    <nickname>dinkar</nickname>
    <marks>85</marks>
  </student>
  <student rollno="493">
    <firstname>Vaneet</firstname>
    <lastname>Gupta</lastname>
    <nickname>vinni</nickname>
    <marks>95</marks>
  </student>
</class>
```

SAXParserDemo.java

```
import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXParserDemo {
    public static void main(String[] args){
        try {
            File inputFile = new File("input.txt");
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser saxParser = factory.newSAXParser();
            UserHandler userhandler = new UserHandler();
            saxParser.parse(inputFile, userhandler);
        } catch (Exception e) { e.printStackTrace(); }
    }
}

class UserHandler extends DefaultHandler
{
    boolean bFirstName = false;
    boolean bLastName = false;
    boolean bNickName = false;
    boolean bMarks = false;
    public void startElement(String uri, String localName, String qName, Attributes
        attributes) throws SAXException {
        if (qName.equalsIgnoreCase("student"))
        {
            String rollNo = attributes.getValue("rollno");
            System.out.println("Roll No : " + rollNo);
        }
    }
}
```



```

else if (qName.equalsIgnoreCase("firstname"))
{    bFirstName = true;  }
else if (qName.equalsIgnoreCase("lastname"))
{    bLastName = true;  }
else if (qName.equalsIgnoreCase("nickname"))
{    bNickName = true;  }
else if (qName.equalsIgnoreCase("marks"))
{    bMarks = true;    }
}
public void endElement(String uri, String localName, String qName) throws SAXException {
if (qName.equalsIgnoreCase("student"))
    {    System.out.println("End Element : " + qName);}
}
public void characters(char ch[], int start, int length) throws SAXException {
if (bFirstName) {
System.out.println("First Name: " + new String(ch, start, length));
bFirstName = false;
} else if (bLastName) {
System.out.println("Last Name: " + new String(ch, start, length));
bLastName = false;
} else if (bNickName) {
System.out.println("Nick Name: " + new String(ch, start, length));
bNickName = false;
} else if (bMarks) {
System.out.println("Marks: " + new String(ch, start, length));
bMarks = false;
}
}}

```

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:

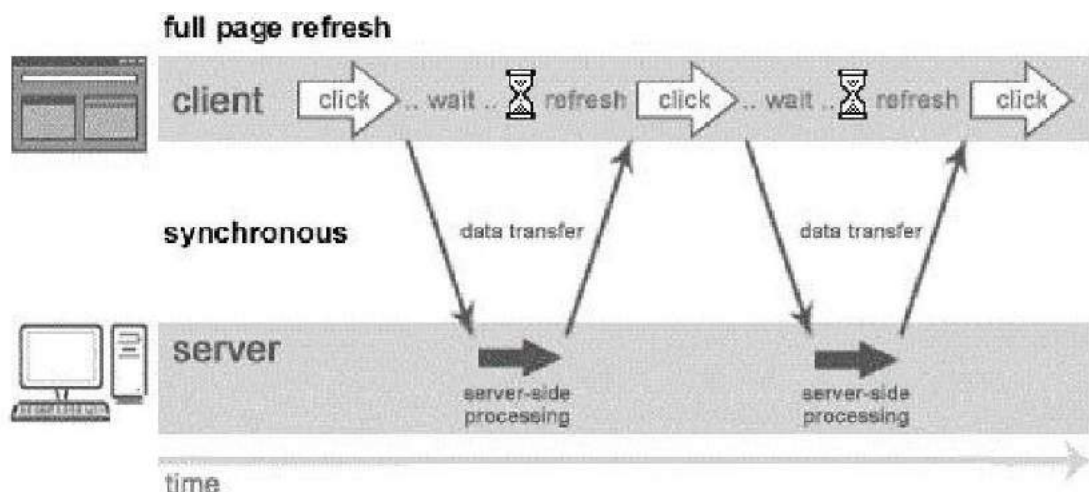
1. Gmail
2. Facebook
3. Twitter
4. Google maps
5. YouTube etc.,

Synchronous Vs. Asynchronous Application

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

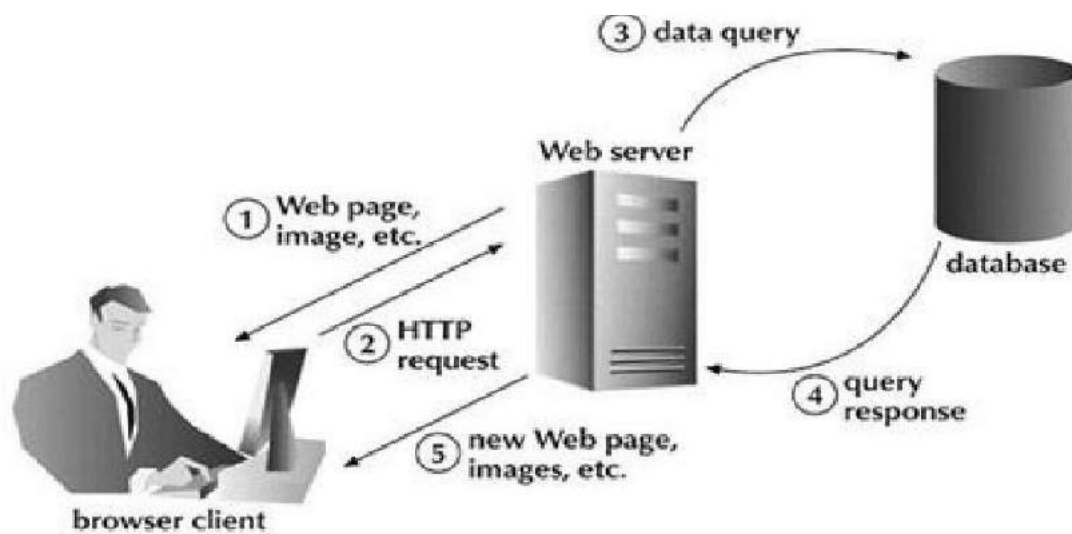
Synchronous (Classic Web-Application Model)

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.



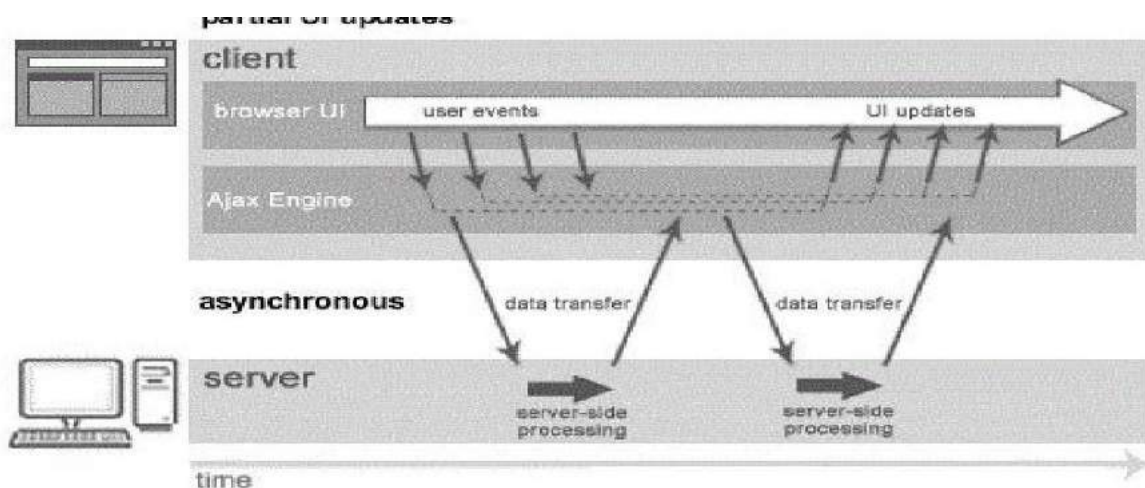
S.JAYA PRAKASH
ASSISTANT PROFESSOR

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.

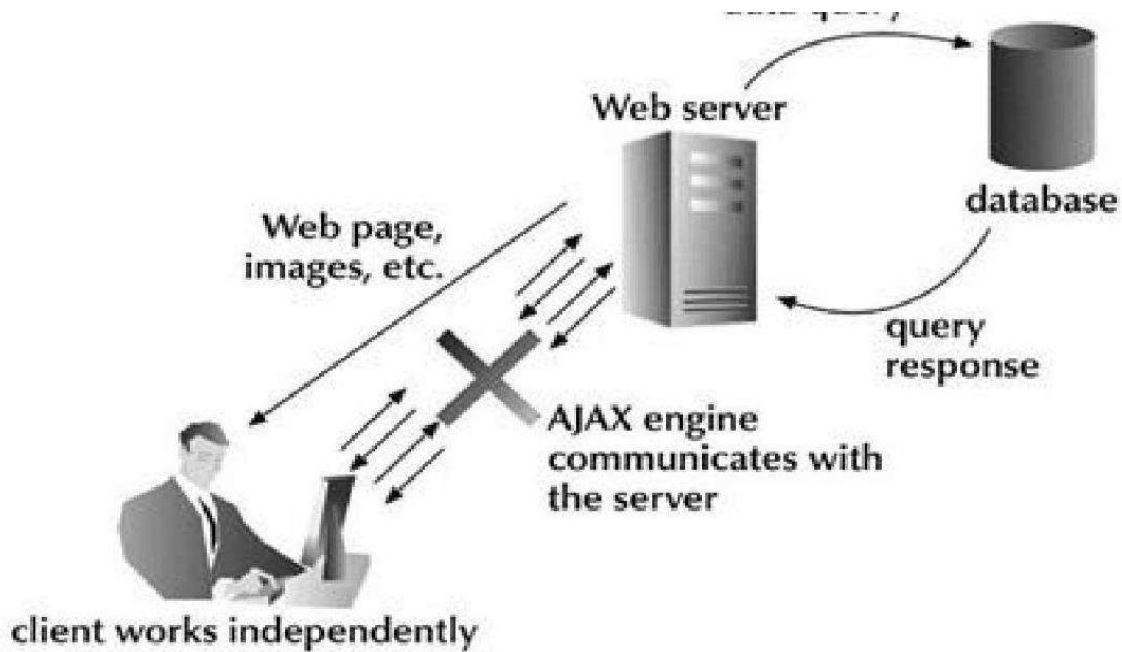


Asynchronous (AJAX Web-Application Model)

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 ready state 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. |
| responseText | Returns response as TEXT. |
| responseXML | Returns response as XML |

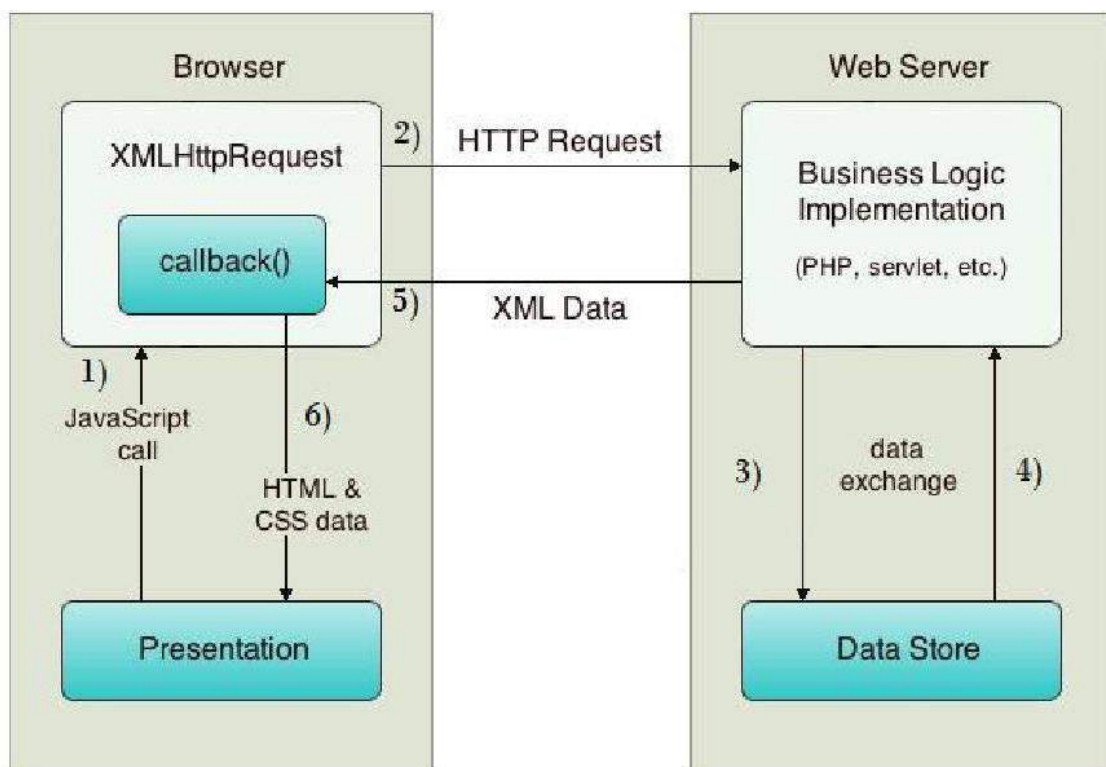
❖ Methods of XMLHttpRequest object:

| 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.



Integrating PHP and AJAX:-

The following example will demonstrate how a web page can communicate with a web server while a user type characters in an input field:

Example:

Start typing a name in the input field below:

First name:

Suggestions:

Example Explained

In the example above, when a user types a character in the input field, a function called "showHint()" is executed. The function is triggered by the onkeyup event.

Here is the HTML code:

Example:

```
<html>
<head>
<script>
function showHint(str) {
    if (str.length == 0) {
        document.getElementById("txtHint").innerHTML = "";
        return;
    } else {
        var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                document.getElementById("txtHint").innerHTML = this.responseText;
            }
        };
        xmlhttp.open("GET", "gethint.php?q=" + str,
            true); xmlhttp.send();
    }
}
</script>
</head>
<body>
```

```
<p><b>Start typing a name in the input field below:</b></p>
<form>
First name: <input type="text" onkeyup="showHint(this.value)">
</form>
<p>Suggestions: <span id="txtHint"></span></p>
</body>
</html>
```

Code explanation:

First, check if the input field is empty (`str.length == 0`). If it is, clear the content of the `txtHint` placeholder and exit the function.

However, if the input field is not empty, do the following:

- ❖ Create an XMLHttpRequest object
- ❖ Create the function to be executed when the server response is ready
- ❖ Send the request off to a PHP file (`gethint.php`) on the server
- ❖ Notice that `q` parameter is added to the url (`gethint.php?q="+str`)
- ❖ And the `str` variable holds the content of the input field

The PHP File - "gethint.php"

The PHP file checks an array of names, and returns the corresponding name(s) to the browser:

```
<?php
// Array with names
$a[] = "Anna";
$a[] = "Brittany";
$a[] = "Cinderella";
$a[] = "Diana";
$a[] = "Eva";
$a[] = "Fiona";
$a[] = "Gunda";
$a[] = "Hege";
$a[] = "Inga";
$a[] = "Johanna";
$a[] = "Kitty";
$a[] = "Linda";
$a[] = "Nina";
$a[] = "Ophelia";
```



```

$a[] = "Petunia";
$a[] = "Amanda";
$a[] = "Raquel";
$a[] = "Cindy";

$a[] = "Doris";
$a[] = "Eve";
$a[] = "Evita";
$a[] = "Sunniva";
$a[] = "Tove";
$a[] = "Unni";
$a[] = "Violet";
$a[] = "Liza";
$a[] = "Elizabeth";
$a[] = "Ellen";
$a[] = "Wenche";
$a[] = "Vicky";

// get the q parameter from URL
$q = $_REQUEST["q"];

$hint = "";

// lookup all hints from array if $q is different from ""
if ($q !== "") {
    $q = strtolower($q);
    $len=strlen($q);
    foreach($a as $name) {
        if (striestr($q, substr($name, 0, $len))) {
            if ($hint === "") {
                $hint = $name;
            } else {
                $hint .= ", $name";
            }
        }
    }
}

```

Web Services

Technology keep on changing, users were forced 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 use particular protocols for communication & with no standardization. **Web service** is the concept of creating services that can be accessed over web.

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
- 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 lightweight & 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/en
  coding">
  <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 multipart 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>
<?xml 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:definitions 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/definitions"
xmlns:tns="http://example.com/stockquote/definitions"
xmlns:xsd="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 searching 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 & transact 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 them 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.

4/4 B.TECH CSE 1ST SEMESTER
WEB TECHNOLOGIES

UNIT-4

S.JAYA PRAKASH
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
SIRCRRCOE, ELURU

UNIT-IV:

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.

Introduction to PHP:

Hypertext Pre-processor (PHP) is a server side scripting language that allows web developers to create dynamic content that interacts with databases.

PHP was created by Rasmus Lerdorf in the year of 1994. PHP is basically used for developing web based software

- PHP is a server side scripting language that is embedded in HTML. It is used to manage dynamic content, databases, session tracking, even build entire e-commerce sites.
- It is integrated with a number of popular databases, including MySQL, PostgreSQL, Oracle, Sybase, Informix, and Microsoft SQL Server.
- PHP supports a large number of major protocols such as POP3, IMAP, and LDAP. PHP4 added support for Java and distributed object architectures (COM and CORBA), making n-tier development a possibility for the first time.

Uses of PHP:

- PHP performs system function, i.e. from files on a system it can create, open, read, write, and close them.
- PHP can handle forms, i.e. gather data from files, save data to a file, through email you can send data, return data to the user.
- You add, delete, and modify elements within your database through PHP.
- Access cookies variables and set cookies.
- Using PHP, you can restrict users to access some pages of your website.

- It can encrypt data.

Characteristics of PHP:

Five important characteristics make PHP's practical nature possible:

- Simplicity
- Efficiency
- Security
- Flexibility
- Familiarity

Creating and Running a PHP Script:

PHP scripts can be embedded in a HTML document or written separately without any HTML markup. In both cases, the file must be saved with the extension .php.

Creating PHP Script:

Any PHP script (code) must be enclosed within the PHP tags which are represented using `<?php` (opening tag) and `?>` (closing tag). Let's consider a simple PHP script which prints "Hello World" on the web document:

```
<?php?>  
print "Hello World";  
?>
```

Save the above file as hello.php.

Running a PHP Script:

To run a PHP script, a web server must be installed and started. Well known web servers are Apache Http Server and Microsoft's Internet Information Service (IIS).

After a web server is installed, place the PHP file *hello.php* in the web server's root directory and start the web server. Now, open a web browser like chrome, firefox or internet explorer and type the following URL in the address bar:

`http://localhost/hello.php`

or

`http://localhost:80/hello.php`

80 is the port at which the web server listens for incoming HTTP requests. The output of the PHP script is:

Hello World

General Syntactic Features:

All the PHP code should be delimited with `<?php` and `?>` tags. All variable names begin with a dollar (\$) sign. Rules for variable names are as in other programming languages. PHP variable names are case-sensitive. Unlike variable names, reserved words and function names are case-insensitive. Below table lists the reserved words in PHP:

| | | | | |
|----------|----------|---------|---------|---------|
| and | else | global | require | virtual |
| break | elseif | if | return | xor |
| case | extends | include | static | while |
| class | false | list | switch | |
| continue | for | new | this | |
| default | foreach | not | true | |
| do | function | or | var | |

PHP allows comments to be specified in three ways. Single-line comments can be specified using # or //. A multi-line comment can be specified using /* and */. PHP statements are terminated with a semi-colon (;). Compound statements and control structures are delimited using braces.

Variables:

A variable is a named location in memory to store data temporarily. PHP is dynamically typed language. So, there is no need for mentioning the data type of a variable. The type will be detected automatically based on the value assigned to the variable. A variable can be created as shown below:

```
$var1 = 10;
```

A variable which is not assigned a value contains NULL. In expressions containing numbers, NULL will be coerced to 0 and in case of strings; NULL will be coerced to an empty string. A variable can be printed as shown below:

```
print("Value of var1 is: $var1");
```

A variable can be checked if it contains a value other than NULL by using the function *IsSet*. This function returns *TRUE* if the variable contains a non-NULL value and *FALSE* if it contains a NULL value.

Constants:

Constants are identifiers which are used to store values that cannot be changed once initialized. A constant doesn't begin with a dollar (\$) sign. The convention for constants is, the name of a constant should always be written in uppercase. We use the function *define* to create constants. Below example demonstrates creating and using constants in PHP:

```
<?php
    define("PI", 3.142);
    print(PI);
    $area = PI*10*10; //Let radius be 10
    print("<br />Area of circle is: $area");
?>
```


Data Types:

PHP provides four scalar types namely *Boolean*, *integer*, *double* and *string* and two compound types namely *array* and *object* and two special types namely *resource* and *NULL*.

PHP has a single integer type, named *integer*. This type is same as *long* type in C. The size of an integer type is generally the size of word in the machine. In most of the machines that size will be 32 bits.

PHP's *double* type corresponds to the *double* type in C and its successors. Double literals can contain a decimal point, an exponent or both. An exponent is represented using E or e followed by a signed integer literal. Digits before and after the decimal point are optional. So, both .12 and 12. are valid double literals.

String is a collection of characters. There is no special type for characters in PHP. A character is considered as a string with length 1. String literals are represented with single quotes or double quotes. In a string literal enclosed in single quotes, escape sequences and variables are not recognized and no substitutions occur. Such substitution is known as *interpolation*. In string literals enclosed in double quotes, escape sequence and variables are recognized and corresponding action is taken.

The only two possible values for a *Boolean* type are *TRUE* and *FALSE* both of which are case-insensitive. Integer value 0 is equal to Boolean *FALSE* and anything other than 0 is equal to *TRUE*. An empty string and string "0" are equal to Boolean *FALSE* and remaining other strings is equal to *TRUE*. Only double value equal to Boolean *FALSE* is 0.0.

Operators:

Operators are used in expressions to perform operations on operands. There are several operators supported by PHP which are categorized into following categories:

- ❖ Arithmetic operators
- ❖ Assignment operators

- ❖ Comparison operators
- ❖ Increment/Decrement operators
- ❖ Logical operators
- ❖ String operators
- ❖ Array operators

Arithmetic Operators:

PHP arithmetic operators are used along with numbers to perform operations like addition, subtraction, multiplication etc. Below is a list of arithmetic operators:

| Operator | Name | Example | Description |
|----------|----------------|--------------|--------------------------------------|
| + | Addition | $\$x + \y | Sum of x and y |
| - | Subtraction | $\$x - \y | Difference of x and y |
| * | Multiplication | $\$x * \y | Product of x and y |
| / | Division | $\$x / \y | Quotient of x divided by y |
| % | Modulus | $\$x \% \y | Remainder of x divided by y |
| ** | Exponentiation | $\$x ** \y | Result of x raised to the power of y |

Assignment Operators:

PHP assignment operators are used in assignment expressions to store the value of expression in to a variable. Below is a list of assignment operators:

| Assignment | Same as | Description |
|------------|--------------|--|
| $x = y$ | $x = y$ | Assigning value of y to x |
| $x += y$ | $x = x + y$ | Adding x and y and store the result in x |
| $x -= y$ | $x = x - y$ | Subtracting y from x and store the result in x |
| $x *= y$ | $x = x * y$ | Multiplying x and y and store the result in x |
| $x /= y$ | $x = x / y$ | Dividing x by y and store the quotient in x |
| $x \% = y$ | $x = x \% y$ | Dividing x by y and store the remainder in x |

Increment/Decrement Operators:

The increment/decrement operators are used to increment the value of variable by 1 or decrement the value of variable by 1. The increment operator is ++ and decrement operator is --.

Relational or Comparison Operators:

PHP comparison operators are used to compare two values and are frequently seen in Boolean expressions. Below is a list of comparison operators:

| Operator | Name | Example | Description |
|----------|--------------------------|-------------|--|
| == | Equal | \$x == \$y | Returns true if x and y are equal |
| === | Identical | \$x === \$y | Returns true if x and y are equal and of same type |
| != | Not equal | \$x != \$y | Returns true if x and y are not equal |
| !== | Not identical | \$x !== \$y | Returns true if x and y are not equal and of same type |
| < | Less than | \$x < \$y | Returns true if x is less than y |
| <= | Less than or equal to | \$x <= \$y | Returns true if x is less than or equal to y |
| > | Greater than | \$x > \$y | Returns true if x is greater than y |
| >= | Greater than or equal to | \$x >= \$y | Returns true if x is greater than or equal to y |
| <> | Not equal | \$x <> \$y | Returns true if x and y are not equal |

Logical Operators:

PHP logical operators are used to find the Boolean value of multiple conditional expressions. Below is a list of logical operators:

| Operator | Name | Example | Description |
|----------|------|-------------|--|
| And | And | \$x and \$y | Returns true when both x and y are true |
| or | Or | \$x or \$y | Returns true when either x or y or both of them are true |

| | | | |
|-----|-----|------------------------|--|
| xor | Xor | $\$x \text{ xor } \y | Returns true when either x or y is true |
| && | And | $\$x \ \&\& \ \y | Returns true when both x and y are true |
| | Or | $\$x \ \ \y | Returns true when either x or y or both of them are true |
| ! | Not | $!\$x$ | Returns true when x is false and vice versa |

String Operators:

PHP provides two operators which are used with strings only. They are listed below:

| Operator | Name | Example | Description |
|----------|--------------------------|------------------|--------------------------------|
| . | Concatenation | $\$str1.\$str2$ | str1 and str2 are concatenated |
| .= | Concatenation Assignment | $\$str1.=\$str2$ | str2 is appended to str1 |

Array Operators:

Below is a list of operators which are used with arrays:

| Operator | Name | Example | Description |
|----------|--------------|-------------|--|
| == | Equality | $\$x==\y | Returns true if x and y have the same key-value pairs |
| === | Identity | $\$x===\y | Returns true if x and y have the same key-value pairs in same order and are of same type |
| != | Inequality | $\$x!=\y | Returns true if x and y are not equal |
| !== | Non-Identity | $\$x!==\y | Returns true if x and y are not identical |
| <> | Inequality | $\$x<>\y | Returns true if x and y are not equal |
| + | Union | $\$x+\y | Returns union of x and y |

String Operations:

The only operators that can operate on strings are . and .= which are for concatenation and concatenation assignment respectively. Strings can be treated as arrays i.e., individual characters can be accessed using the index values. The index value begins with zero. A character at index *i* can be retrieved as \$str[i]. An example is given below:

```
<?php
    $str1 = "master";
    print($str1[2]);
?>
```

The *print* statement in the above code prints 's'.

There are several predefined functions to manipulate strings. Some of them are listed below:

| Function Name | Parameters | Description |
|----------------------|----------------------------|---|
| strlen | One string | Returns number of characters in the string |
| strcmp | Two strings | Returns zero if both strings are equal, a -ve number if the first string occurs before second string or a +ve number if the first string occurs after the second string |
| strpos | Two strings | Returns position of second string in the first string or false if not found |
| substr | One string and one integer | Returns the substring from the specified string from the position specified as an integer. If a third integer value is specified, it represents the length of the substring to be retrieved |
| chop | One string | Returns the string with all white space characters removed from the end |
| trim | One string | Returns the string with all white space characters removed on both sides |
| ltrim | One string | Returns the string with all white space characters removed from the beginning |
| strtolower | One string | Returns the string with all the characters converted to lowercase |
| strtoupper | One string | Returns the string with all the characters converted to uppercase |
| strrev | One string | Returns the reverse of the given string |
| str_replace | Three strings | Returns the string in which a old substring is replaced by the new substring |
| str_word_count | One string | Returns the word count in the given string |

Output:

The output of a PHP processor is HTML code. There are multiple ways to output information onto the web document. One way is to use *echo* statement with or without parentheses as shown below:

```
echo("Hello World");
```

Second way is by using *print* statement with or without parentheses as shown below:

```
print("This is PHP");
```

The difference between *echo* and *print* are: *echo* doesn't return anything while *print* returns 1. So *print* can be used in expressions. *echo* can accept multiple arguments while *print* accepts only one argument. Also *echo* is faster than *print*.

Third way is by using the *printf* function. The syntax of this function is given below:

```
printf(literal_string, param1, param2, );
```

Below is an example which demonstrates *echo*, *print* and *printf* statements:

```
<?php  
    $a = 10;  
    $b = 20;  
    $sum = $a + $b;  
    echo("Sum of $a and $b is: $sum <br />"); print("Sum  
of $a and $b is: $sum <br />"); printf("Sum of %d and  
%d is: %d", $a, $b, $sum);  
?>
```

Control Statements:

Control statements are used to control the flow of execution in a script. There are three categories of control statements in PHP: selection statements, iteration / loop statements and jump statements.

Selection statements:

The selection statements in PHP allow the PHP processor to select a set of statements based on the truth value of a condition or Boolean expression. Selection statements in PHP are if, if-else, elseif ladder and switch statement.

Syntax of *if* is given below:

```
if(condition / expression)  
{  
    statements(s);  
}
```

Syntax of *if-else* is given below:

```
if(condition / expression)  
{  
    statements(s);  
}  
else  
{  
    statements(s);  
}
```


Syntax of *elseif* ladder is given below:

```
if(condition / expression)  
{  
    statements(s);  
}  
elseif(condition / expression)  
{  
    statements(s);  
}  
elseif(condition / expression)  
{  
    statements(s);  
}  
else  
{  
    statements(s);  
}
```

Syntax of *switch* statement is shown below:

```
switch(expression)  
{  
    case label1:  
        statement(s);  
        break;  
    case label2:  
        statement(s);  
        break;  
    case label3:  
        statement(s);  
        break;  
    default:  
        statement(s);  
}
```

The labels for *case* statements can be an integer, double or a string. The *default* block is optional. If the *break* statement is absent, the following cases will also execute until a *break* statement is found.

Iteration or Loop Statements:

The iteration statements in PHP allow PHP processor to iterate over or repeat a set of statements for finite or infinite times. Iteration statements supported by PHP are while, do-while, for and foreach.

Syntax of ***while*** loop is given below:

```
while(condition / expression)  
{  
    statements(s);  
}
```

Syntax of ***do-while*** loop is given below:

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

Syntax of ***for*** loop is given below:

```
for(initialization; condition / expression; increment/decrement)  
{  
    statement(s);  
}
```

The ***foreach*** loop is used to iterate over array elements and its syntax is given below:

```
//For normal arrays
foreach(array as variable_name)
{
    statement(s);
}
```

or

```
//For associative arrays
foreach(array as key => value)
{
    statement(s);
}
```

Jump Statements:

The jump statements available in PHP are *break* and *continue*. The *break* statement is used to break the control from a loop, take it to the next statement after the loop and *continue* is used to skip the control from current line to the next iteration of the loop.

Arrays:

Array is a collection of heterogeneous elements. There are two types of arrays in PHP. First type of array is a normal one that contains integer keys (indexes) which can be found in any typical programming languages. The second type of arrays is an *associative array*, where the keys are strings. Associative arrays are also known as hashes.

Array Creation:

Arrays in PHP are dynamic. There is no need to specify the size of an array. A normal array can be created by using the integer index and the assignment operator as shown below:

```
$array1[0] = 10;
```

If no integer index is specified, the index will be set to 1 larger than the previous largest index value used. Consider the following example:

```
$array2[3] = 5;
```

```
$array2[ ] = 90;
```

In the above example, 90 will be stored at index location 4.

There is another way for creating an array, using the *array* construct, which is not a function. The data elements are passed to the *array* construct as shown in the below example:

```
$array3 = array(10, 15, 34, 56);
```

```
$array4 = array( );
```

In the above example *array4* is an empty array which can be used later to store elements.

A traditional array with irregular indexes can be created as shown below:

```
$array5 = array(3 => 15, 4 =>, 37, 5 => 23);
```

The above code creates an array with indexes 3, 4 and 5.

An associative array which contains named keys (indexes) can be created as shown below:

```
$ages = array("Ken" => 29, "John" => 30, "Steve" => 26, "Bob" => 28);
```

An array in PHP can be a mixture of both traditional and associative arrays.

Accessing Array Elements:

Array elements can be accessed using the subscript (index or key) value which is enclosed in square brackets.

Consider a traditional array as shown below:

```
$array1 = array(10, 20, 30, 40);
```

Third element (30) in the above array can be accessed by writing `$array1[2]`. The index of any traditional array starts with 0.

Consider an associative array or hash as shown below:

```
$ages = array("Ken" => 29, "John" => 30, "Steve" => 26, "Bob" => 28);
```

We can access the age (30) of John by writing `$ages['John']`.

Iterating through Arrays:

The *foreach* loop can be used to print / access the values in a traditional or an associative array.

Consider a traditional array as shown below:

```
$array1 = array(10, 20, 30, 40);
```

The *foreach* loop to print the values in the above array can be written as shown below:

```
foreach($array1 as $val)
    print("$val <br />");
```

Consider an associative array or hash as shown below:

```
$ages = array("Ken" => 29, "John" => 30, "Steve" => 26, "Bob" => 28);
```

The *foreach* loop to print the keys and values in the above array can be written as shown below:

```
foreach($ages as $name => $age)  
    print("$name age is $age <br />");
```

Array Functions:

The array elements can be manipulated or accessed in different ways. PHP has an extensive list of predefined functions that comes in handy while working with arrays. Some these functions are mentioned below:

| Function Name | Parameters | Description |
|------------------|---------------|---|
| count | One array | Returns the number of elements in the array |
| unset | One array | Deletes the element from memory |
| array_keys | One array | Returns the keys (indexes) as an array |
| array_values | One array | Returns the values as an array |
| array_key_exists | Key, array | Returns <i>true</i> if the key is found, <i>false</i> otherwise |
| is_array | One array | Returns <i>true</i> if the argument is an array, <i>false</i> otherwise |
| in_array | Exp, array | Returns <i>true</i> if <i>exp</i> is in the array, <i>false</i> otherwise |
| explode | Delim, string | Returns an array of substrings based on the <i>delim</i> |
| implode | Delim, array | Returns a string joining the array elements with <i>delim</i> |
| sort | One array | Sorts the values in the array and renames the keys to integer values 0, 1, 2, ... |
| asort | One array | Sorts the values in the array preserving the keys |
| ksort | One array | Sorts the keys in the array preserving the values |
| rsort | One array | Reverse of sort (descending order) |
| rasort | One array | Reverse of asort (descending order) |
| rsort | One array | Reverse of ksort (descending order) |

Functions:

A function is a part of program which contains set of statements that can perform a desired task. A function can be defined with the *function* keyword followed by the function name and an optional set of parameters enclosed in parentheses. A function definition may not occur before a function call. It can be anywhere in the script. Although function definitions can be nested, it is not encouraged as they can make the script complex. Syntax for defining a function is given below:

```
function function_name([param1, param2,... ])  
{  
    //Body of the function  
    [return expression;]  
}
```

The execution of the function terminates whenever a *return* statement is encountered or whenever the control reaches the end of the function's body.

Parameters:

As in any typical programming language, parameters in a function definition are known as *formal parameters* and the parameters in a function call are known as *actual parameters*. The number of formal parameters and normal parameters may not be equal.

The default parameter passing mechanism is pass-by-value where a copy of the actual parameters is passed to actual parameters. Below example demonstrates pass-by-value:

```
function swap($x, $y)  
{  
    $temp = $x;  
    $x = $y;  
    $y = $temp;  
}  
$a = 10;  
$b = 20;  
sum($a, $b);
```

After the above code is executed the values of *a* and *b* remains 10 and 20 even after the function call.

Another way of passing parameters is *pass-by-reference* where the address of the variable is passed rather than a copy. In this mechanism changes on formal parameters are reflected on actual parameters. The address can be passed using & symbol before the variable as shown below:

```
function swap($x, $y)
{
    $temp = $x;
    $x = $y;
    $y = $temp;
}
$a = 10;
$b = 20;
sum(&$a, &$b);
```

or the above code may be also written as shown below:

```
function swap(&$x, &$y)
{
    $temp = $x;
    $x = $y;
    $y = $temp;
}
$a = 10;
$b = 20;
sum($a, $b);
```

Pass-by-reference can be used to return multiple values from a function.

Scope of Variables:

Scope refers to the visibility of a variable in the PHP script. A variable defined inside a function will have local scope i.e., within the function. A variable outside the function can have the same name as a local variable in a function, where the local variable has higher precedence over the outer variable.

In some cases, a function might need access to a variable declared outside the function definition. In such cases, *global* declaration can be used before the variable name which allows the accessibility of the variable that is declared outside the function definition. Below example demonstrates local and global variables:

```
function sum_array($list)
{
    global $allsum; //Global variable
    $sum = 0; //Local variable
    foreach($list as $x)
        $sum += $x;
    $allsum += $sum;
    return $sum;
}
$allsum = 0;
$array1 = new array(10, 20, 30, 40);
$array2 = new array(1, 2, 3, 4);
$ans1 = sum_array($array1);
$ans2 = sum_array($array2); print("Sum
of array1 is: $ans1<br />"); print("Sum
of all arrays is: $allsum");
```

In the above example, first print statement gives 100 and the second print statement gives 110.

Lifetime of Variables:

The lifetime of a normal variable in a function is until the function execution completes. Sometimes, there might be a need to retain the value of a variable between function calls. In such cases, the variable must be declared with *static*. The lifetime of *static* variable is until the execution of script completes. Below example demonstrates the use of *static* variables:

```
function func1( )  
{  
    static $count = 0;  
    $count++;  
}  
for($i = 1; $i <= 10; $i++)  
    func1( );  
print("func1 is called $count times.");
```

In the above example, the *print* statement prints 10 instead of 0.

Form Data Processing:

One of the applications of PHP is processing the data provided by the users in HTML forms. PHP provides two implicit arrays `$_GET` and `$_POST` which are global variables and are accessible anywhere in a PHP script.

The array `$_GET` is used when the attribute *method* of the form tag is set to *GET* and the array `$_POST` is used when the attribute *method* of the form tag is set to *POST*. Both arrays contain the form data stored as key-value pairs, where key contains the value of *name* attribute of the form element and value contains the value of the *value* attribute of the form element.

Below example demonstrates validation of user input in a simple (X)HTML form using PHP:

```
//HTML Code
<html>
  <head>
    <title>Login Form</title>
  </head>
  <body>
    <form action="validate.php">
      Username: <input type="text" name="txtuser" /><br />
      Password: <input type="password" name="txtpass" /><br />
      <input type="submit" value="Submit" />
      <input type="reset" value="Clear" />
    </form>
  </body>
</html>
```

By default when the *method* attribute is not set, it will be implicitly set to GET. So, in PHP we have to use the array `$_GET` to retrieve the form data as shown below:

//PHP Code - validate.php

<?php

\$user = \$_GET["txtuser"]; //txtuser is the name of textfield in the HTML form

\$pass = \$_GET["txtpass"]; //txtpass is the name of password field in the HTML form if(\$user == "")

print("Username cannot be empty!");

elseif(\$pass == "")

print("Password cannot be empty!");

else

print("Login success! Click Here to proceed");

?>

Database Access using PHP:

Relational databases are the primary choice for data storage in the Web. A DBMS provides a database along with a set of tools to manage the data in the database. One example for DBMS is MYSQL. More than half of the websites in the Internet are created using PHP and the data store as MYSQL.

The steps required to access data from a table in a database provided by MYSQL can be summarized as follows:

- ❖ Establish or open a connection to the MYSQL server.
- ❖ Select a database.
- ❖ Execute the query against the database.
- ❖ Process the result returned by the server.
- ❖ Close the connection

To work with MYSQL databases, PHP provides in-built support in the form of predefined functions. To establish a connection with the MYSQL server, use the function *mysql_connect* which accepts three optional parameters. First parameter is the server name, second parameter is the MYSQL server's user name and the third parameter is the password for MYSQL server. If the connection to MYSQL fails, the function returns *false*. This function is generally used in conjunction with *die* function which can be used to print errors using the function *mysql_error* and terminate the execution of the script.

After opening a connection to the database, a database must be selected to execute the SQL queries. A database can be selected by using the function *mysql_select_db* which accepts a single string parameter, the name of the database.

After selecting a database, the next step is to specify the query which is generally stored as a string in a variable. This variable will be passed as a parameter to the function *mysql_query* which executes the query against the database.

The result of execution of the query can be stored in a variable. The result can be parsed row by row as an array by using the function *mysql_fetch_array*.

Below is a list of functions provides by PHP to work with MYSQL databases:

| Function | Description |
|--|---|
| mysql_connect(localhost, username, password) | Established a connection with the MYSQL server. Returns <i>false</i> if the connection fails |
| mysql_select_db(link, "DB_Name") | Selects the specified database |
| mysql_query(link, "SQL_Query") | Executes the specified query against the database and returns the result |
| mysql_fetch_array(\$result) | Returns an array of the next row |
| mysql_num_rows(\$result) | Returns the number of rows in the result |
| mysql_num_fields(\$result) | Returns the number of columns in the result row |
| mysql_error() | Returns error message |

Below example demonstrates user validation against the details stored in a database using PHP and MYSQL:

//HTML Code

```
<html>
  <head>
<title>Login Form</title>
  </head>
  <body>
    <form action="getdb.php" method="get">
      <label>Username: </label>
          <input type="text" name="user" /><br />
      <label>Password: </label>
          <input type="password" name="pass" /><br />
          <input type="submit" value="Submit" />
          <input type="reset" value="Clear" />
    </form>
```

S.JAYA PRAKASH

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

//PHP Code - getdb.php

```
<?php
    $utext = $_REQUEST["user"];
    $ptext = $_REQUEST["pass"];
    $flag = false;

    $hostname = "localhost";
    $username = "root";
    $password = "123456";

    $con = mysql_connect($hostname, $username, $password) or die(mysql_error());
    mysql_select_db($con, "myapp") or die(mysql_error());
    $result = mysql_query($con, "select * from users") or die(mysql_error());

    while($x = mysql_fetch_array($result))
    {
        if($utext == $x["uname"] && $ptext == $x["pwd"])
            $flag = true;
    }

    if($flag)
        echo "Valid user!";
    else
        echo "Invalid username or password!";
?>
```


4/4 B.TECH CSE 1ST SEMESTER
WEB TECHNOLOGIES

UNIT-5

S.JAYA PRAKASH
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
SIRCRRCOE, ELURU

UNIT-V:

Introduction to PERL, Operators and if statements, Program design and control structures, Arrays, Hashes and File handling, Regular expressions, Subroutines, Retrieving documents from the web with Perl.

Perl:

Perl stands for “Practical Extraction and Report Language” or “Pathologically Eclectic Rubbish Lister.” It is a powerful glue language useful for tying together the loose ends of computing life.

History:

Perl is the natural outgrowth of a project started by Larry Wall in 1986. Originally intended as a configuration and control system for six VAXes and six SUNs located on opposite ends of the country, it grew into a more general tool for system administration on many platforms. Since its unveiling to programmers at large, it has become the work of a large body of developers. Larry Wall, however, remains its principle architect.

Although the first platform Perl inhabited was UNIX, it has since been ported to over 70 different operating systems including, but not limited to, Windows 9x/NT/2000, MacOS, VMS, Linux, UNIX (many variants), BeOS, LynxOS, and QNX.

Uses of Perl

1. Tool for general system administration
2. Processing textual or numerical data
3. Database interconnectivity
4. Common Gateway Interface (CGI/Web) programming
5. Driving other programs! (FTP, Mail, WWW, OLE)

Perl Basics

Script names

While generally speaking you can name your script/program anything you want, there are a number of conventional extensions applied to portions of the Perl bestiary:

- `.pm` - Perl modules
- `.pl` - Perl libraries (and scripts on UNIX)
- `.plx` - Perl scripts

Language properties

- Perl is an interpreted language – program code is interpreted at run time. Perl is unique among interpreted languages, though. Code is compiled by the interpreter before it is actually executed.
- Many Perl idioms read like English
- Free format language – whitespace between tokens is optional
- Comments are single-line, beginning with `#`
- Statements end with a semicolon (`;`)
- Only subroutines and functions need to be explicitly declared
- Blocks of statements are enclosed in curly braces `{ }`
- A script has no `main()`

Invocation

On platforms such as UNIX, the first line of a Perl program should begin with

```
#!/usr/bin/perl
```

and the file should have executable permissions. Then typing the name of the script will cause it to be executed.

Unfortunately, Windows does not have a real equivalent of the UNIX “shebang” line. On Windows 95/98, you will have to call the Perl interpreter with the script as an argument:

```
> perl myscript.plx
```

On Windows NT, you can associate the `.plx` extension with the Perl interpreter:

```
> assoc .plx=Perl
> ftype Perl=c:\myperl\bin\perl.exe %1% %*
> set PATHEXT=%PATHEXT%;.plx
```

After taking these steps, you can execute your script from the command line as follows:

```
> myscript
```

The *ActivePerl* distribution includes a `pl2bat` utility for converting Perl scripts into batch files.

You can also run the interpreter by itself from the command line. This is often useful to execute short snippets of code:

```
perl -e 'code'
```

Alternatively, you can run the interpreter in “debugging” mode to obtain a shell-like environment for testing code scraps:

```
perl -de 1
```

Data Types & Variables

Basic Types

The basic data types known to Perl are scalars, lists, and hashes.

| | | |
|--------|--------------------|--|
| Scalar | <code>\$foo</code> | Simple variables that can be a number, a string, or a reference. A scalar is a “thingy.” |
| List | <code>@foo</code> | An ordered array of scalars accessed using a numeric subscript. <code>\$foo[0]</code> |
| Hash | <code>%foo</code> | An unordered set of key/value pairs accessed using the keys as subscripts. <code>\$foo{key}</code> |

Perl uses an internal type called a *typeglob* to hold an entire symbol table entry. The effect is that scalars, lists, hashes, and filehandles occupy separate namespaces (i.e., `$foo[0]` is not part of `$foo` or of `%foo`). The prefix of a typeglob is `*`, to indicate “all types.” Typeglobs are used in Perl programs to pass data types by reference.

You will find references to literals and variables in the documentation. *Literals* are symbols that give an actual value, rather than represent possible values, as do *variables*. For example in `$foo = 1`, `$foo` is a scalar variable and `1` is an integer literal.

Variables have a value of `undef` before they are defined (assigned). The upshot is that accessing values of a previously undefined variable will not (necessarily) raise an exception.

Variable Contexts

Perl data types can be treated in different ways depending on the context in which they are accessed.

| | |
|---------------|--|
| Scalar | Accessing data items as scalar values. In the case of lists and hashes, <code>\$foo[0]</code> and <code>\$foo{key}</code> , respectively. Scalars also have numeric, string, and don't-care contexts to cover situations in which conversions need to be done. |
| List | Treating lists and hashes as atomic objects |
| Boolean | Used in situations where an expression is evaluated as true or false. (Numeric: 0=false; String: null=false, Other: undef=false) |
| Void | Does not care (or want to care) about return value |
| Interpolative | Takes place inside quotes or things that act like quotes |

Special Variables (defaults)

Some variables have a predefined and special meaning to Perl. A few of the most commonly used ones are listed below.

| | |
|---------------------------|---|
| <code>\$_</code> | The default input and pattern-searching space |
| <code>\$0</code> | Program name |
| <code>\$\$</code> | Current process ID |
| <code>#!</code> | Current value of <code>errno</code> |
| <code>@ARGV</code> | Array containing command-line arguments for the script |
| <code>@INC</code> | The array containing the list of places to look for Perl scripts to be evaluated by the <code>do</code> , <code>require</code> , or <code>use</code> constructs |
| <code>%ENV</code> | The hash containing the current environment |
| <code>%SIG</code> | The hash used to set signal handlers for various signals |

Scalars

Scalars are simple variables that are either numbers or strings of characters. Scalar variable names begin with a dollar sign followed by a letter, then possibly more letters, digits, or underscores. Variable names are case-sensitive.

Numbers

Numbers are represented internally as either signed integers or double precision floating point numbers. Floating point literals are the same used in C. Integer literals include decimal (255), octal (0377), and hexadecimal (0xff) values.

Strings

Strings are simply sequences of characters. String literals are delimited by quotes:

| | | |
|--------------|------------------------|---|
| Single quote | <code>'string'</code> | Enclose a sequence of characters |
| Double quote | <code>"string"</code> | Subject to backslash and variable interpolation |
| Back quote | <code>`command`</code> | Evaluates to the output of the enclosed command |

The backslash escapes are the same as those used in C:

| | | | |
|-----------------|-----------------|-----------------|--------------|
| <code>\n</code> | Newline | <code>\e</code> | Escape |
| <code>\r</code> | Carriage return | <code>\\</code> | Backslash |
| <code>\t</code> | Tab | <code>\"</code> | Double quote |
| <code>\b</code> | Backspace | <code>\'</code> | Single quote |

In Windows, to represent a path, use either `"c:\\temp"` (an escaped backslash) or `"c:/temp"` (UNIX-style forward slash).

Strings can be concatenated using the `.` operator: `$foo = "hello" . "world";`

Basic I/O

The easiest means to get operator input to your program is using the "diamond" operator:

```
$input = <>;
```

The input from the diamond operator includes a newline (`\n`). To get rid of this pesky character, use either `chop()` or `chomp()`. `chop()` removes the last character of the string, while `chomp()` removes *any* line-ending characters (defined in the special variable `$/`). If no argument is given, these functions operate on the `$_` variable.

To do the converse, simply use Perl's print function:

```
print $output."\n";
```

Basic Operators

Arithmetic

| Example | Name | Result |
|-------------------------|----------------|---|
| <code>\$a + \$b</code> | Addition | Sum of <code>\$a</code> and <code>\$b</code> |
| <code>\$a * \$b</code> | Multiplication | Product of <code>\$a</code> and <code>\$b</code> |
| <code>\$a % \$b</code> | Modulus | Remainder of <code>\$a</code> divided by <code>\$b</code> |
| <code>\$a ** \$b</code> | Exponentiation | <code>\$a</code> to the power of <code>\$b</code> |

String

| Example | Name | Result |
|-----------------------------|---------------|---|
| <code>\$a . "string"</code> | Concatenation | String built from pieces |
| <code>"\$a string"</code> | Interpolation | String incorporating the value of <code>\$a</code> |
| <code>\$a x \$b</code> | Repeat | String in which <code>\$a</code> is repeated <code>\$b</code> times |

Assignment

The basic assignment operator is "=": `$a = $b`.

Perl conforms to the C idiom that `lvalue operator= expression` is evaluated as: `lvalue = lvalue operator expression`

So that `$a *= $b` is equivalent to `$a = $a * $b`
`$a += $b` `$a = $a + $b`

This also works for the string concatenation operator: `$a .= "\n"`

Autoincrement and Autodecrement

The autoincrement and autodecrement operators are special cases of the assignment operators, which add or subtract 1 from the value of a variable:

| | | |
|---------------------------|---------------|---------------------|
| <code>++\$a, \$a++</code> | Autoincrement | Add 1 to \$a |
| <code>--\$a, \$a--</code> | Autodecrement | Subtract 1 from \$a |

Logical

Conditions for truth:

Any string is true except for "" and "0"

Any number is true except for 0

Any reference is true

Any undefined value is false

| Example | Name | Result |
|---------------------------------|------|--|
| <code>\$a && \$b</code> | And | True if both <code>\$a</code> and <code>\$b</code> are true |
| <code>\$a \$b</code> | Or | <code>\$a</code> if <code>\$a</code> is true; <code>\$b</code> otherwise |
| <code>!\$a</code> | Not | True if <code>\$a</code> is not true |
| <code>\$a and \$b</code> | And | True if both <code>\$a</code> and <code>\$b</code> are true |
| <code>\$a or \$b</code> | Or | <code>\$a</code> if <code>\$a</code> is true; <code>\$b</code> otherwise |
| <code>not \$a</code> | Not | True if <code>\$a</code> is not true |

Logical operators are often used to "short circuit" expressions, as in:

```
open(FILE,"< input.dat") or die "Can't open file";
```

Comparison

| Comparison | Numeric | String | Result |
|--------------------|------------------------|------------------|---|
| Equal | <code>==</code> | <code>eq</code> | True if \$a equal to \$b |
| Not equal | <code>!=</code> | <code>ne</code> | True if \$a not equal to \$b |
| Less than | <code><</code> | <code>lt</code> | True if \$a less than \$b |
| Greater than | <code>></code> | <code>gt</code> | True if \$a greater than \$b |
| Less than or equal | <code><=</code> | <code>le</code> | True if \$a not greater than \$b |
| Comparison | <code><=></code> | <code>cmp</code> | 0 if \$a and \$b equal 1 if \$a greater -1 if \$b greater |

Operator Precedence

Perl operators have the following precedence, listed from the highest to the lowest, where operators at the same precedence level resolve according to associativity:

| Associativity | Operators | Description |
|---------------|------------------------------------|---|
| Left | Terms and list operators | |
| Left | <code>-></code> | Infix dereference operator |
| | <code>++</code> | Auto-increment |
| | <code>--</code> | Auto-decrement |
| Right | <code>\</code> | Reference to an object (unary) |
| Right | <code>! ~</code> | Unary negation, bitwise complement |
| Right | <code>+ -</code> | Unary plus, minus |
| Left | <code>=~</code> | Binds scalar to a match pattern |
| Left | <code>!~</code> | Same, but negates the result |
| Left | <code>* / % x</code> | Multiplication, Division, Modulo, Repeat |
| Left | <code>+ - .</code> | Addition, Subtraction, Concatenation |
| Left | <code>>> <<</code> | Bitwise shift right, left |
| | <code>< > <= >=</code> | Numerical relational operators |
| | <code>lt gt le ge</code> | String relational operators |
| | <code>== != <=></code> | Numerical comparison operators |
| | <code>eq ne cmp</code> | String comparison operators |
| Left | <code>&</code> | Bitwise AND |
| Left | <code> ^</code> | Bitwise OR, Exclusive OR |
| Left | <code>&&</code> | Logical AND |
| Left | <code> </code> | Logical OR |
| | <code>..</code> | In scalar context, range operator |
| | | In array context, enumeration |
| Right | <code>?:</code> | Conditional (if ? then : else) operator |
| Right | <code>= += -= etc</code> | Assignment operators |
| Left | <code>,</code> | Comma operator, also list element separator |
| | <code>=></code> | Same, enforces left operand to be string |
| Right | <code>not</code> | Low precedence logical NOT |
| Right | <code>and</code> | Low precedence logical AND |
| Right | <code>or xor</code> | Low precedence logical OR |

Parentheses can be used to group an expression into a term.

A list consists of expressions, variables, or lists, separated by commas. An array variable or an array slice many always be used instead of a list.

Control Structures

Statement Blocks

A statement block is simply a sequence of statements enclosed in curly braces:

```
{
    first_statement;
    second_statement;
    last_statement
}
```

Conditional Structures (If/elsif/else)

The basic construction to execute blocks of statements is the **if** statement. The **if** statement permits execution of the associated statement block if the test expression evaluates as true. It is important to note that unlike many compiled languages, it is necessary to enclose the statement block in curly braces, even if only one statement is to be executed.

The general form of an if/then/else type of control statement is as follows:

```
if (expression_one) {
    true_one_statement;
} elsif (expression_two) {
    true_two_statement;
} else {
    all_false_statement;
}
```

For convenience, Perl also offers a construct to test if an expression is false:

```
unless (expression) {
    false_statement;
} else {
    true_statement;
}
```

Note that the order of the conditional can be inverted as well:

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

The “ternary” operator is another nifty one to keep in your bag of tricks:

```
$var = (expression) ? true_value : false_value;
```

It is equivalent to:

```
if (expression) {
    $var = true_value;
} else {
    $var = false_value;
}
```

Loops

Perl provides several different means of repetitively executing blocks of statements.

While

The basic *while* loop tests an expression before executing a statement block

```
while (expression) {  
    statements;  
}
```

Until

The *until* loop tests an expression at the end of a statement block; statements will be executed until the expression evaluates as true.

```
until (expression) {  
    statements;  
}
```

Do while

A statement block is executed at least once, and then repeatedly until the test expression is false.

```
do {  
    statements;  
} while (expression);
```

Do until

A statement block is executed at least once, and then repeatedly until the test expression is true.

```
do {  
    statements;  
} until (expression);
```

For

The *for* loop has three semicolon-separated expressions within its parentheses. These expressions function respectively for the initialization, the condition, and re-initialization expressions of the loop. The for loop

```
    for (initial_exp; test_exp; reinit_exp) {
        statements;
    }
```

This structure is typically used to iterate over a range of values. The loop runs until the `test_exp` is false.

```
    for ($i; $i<10;$i++) {
        print $i;
    }
```

Foreach

The *foreach* statement is much like the *for* statement except it loops over the elements of a list:

```
    foreach $i (@some_list) {
        statements;
    }
```

If the scalar loop variable is omitted, `$_` is used.

Labels

Any statement block can be given a label. Labels are identifiers that follow variable naming rules. They are placed immediately before a statement block and end with a colon:

```
    SOMELABEL: {
        statements;
    }
```

You can short-circuit loop execution with the directives **next** and **last**:

- **next** skips the remaining statements in the loop and proceeds to the next iteration (if any)
- **last** immediately exits the loop in question
- **redo** jumps to the beginning of the block (restarting current iteration)

Next and last can be used in conjunction with a label to specify a loop by name. If the label is omitted, the presumption is that next/last refers to the innermost enclosing loop.

Usually deprecated in most languages, the `goto` expression is nevertheless supported by Perl. It is usually used in connection with a label

```
    goto LABEL;
```

to jump to a particular point of execution.

Indexed Arrays (Lists)

A list is an ordered set of scalar data. List names follow the same basic rules as for scalars. A reference to a list has the form `@foo`.

List literals

List literals consist of comma-separated values enclosed in parentheses:

```
(1,2,3)
("foo",4.5)
```

A range can be represented using a list constructor function (such as `..`):

```
(1..9) = (1,2,3,4,5,6,7,8,9)
($a..$b) = ($a, $a+1, ... , $b-1,$b)
```

In the case of string values, it can be convenient to use the “quote-word” syntax

```
@a = ("fred","barney","betty","wilma");
@a = qw( fred barney betty wilma );
```

Accessing List Elements

List elements are subscripted by sequential integers, beginning with 0

```
$foo[5] is the sixth element of @foo
```

The special variable `$#foo` provides the index value of the last element of `@foo`.

A subset of elements from a list is called a *slice*.

```
@foo[0,1] is the same as ($foo[0],$foo[1])
```

You can also access slices of list literals:

```
@foo = (qw( fred barney betty wilma ))[2,3]
```

List operators and functions

Many list-processing functions operate on the paradigm in which the list is a stack. The highest subscript end of the list is the “top,” and the lowest is the bottom.

| | |
|----------------|---|
| push | Appends a value to the end of the list push(@mylist,\$newvalue) |
| pop | Removes the last element from the list (and returns it) pop(@mylist) |
| shift | Removes the first element from the list (and returns it) shift(@mylist) |
| unshift | Prepends a value to the beginning of the list unshift(@mylist,\$newvalue) |
| splice | Inserts elements into a list at an arbitrary position splice(@mylist,\$offset,\$replace,@newlist) |

The **reverse** function reverses the order of the elements of a list

```
@b = reverse(@a);
```

The **sort** function sorts the elements of its argument as strings in ASCII order. You can also customize the sorting algorithm if you want to do something special.

```
@x = sort(@y);
```

The **chomp** function works on lists as well as scalars. When invoked on a list, it removes newlines (record separators) from each element of its argument.

Associative Arrays (Hashes)

A *hash* (or associative array) is an unordered set of key/value pairs whose elements are indexed by their keys. Hash variable names have the form `%foo`.

Hash Variables and Literals

A literal representation of a hash is a list with an even number of elements (key/value pairs, remember?).

```
%foo = qw( fred wilma barney betty );
%foo = @foolist;
```

To add individual elements to a hash, all you have to do is set them individually:

```
$foo{fred} = "wilma";
$foo{barney} = "betty";
```

You can also access slices of hashes in a manner similar to the list case:

```
@foo{"fred","barney"} = qw( wilma betty );
```

Hash Functions

The **keys** function returns a list of all the current keys for the hash in question.

```
@hashkeys = keys(%hash);
```

As with all other built-in functions, the parentheses are optional:

```
@hashkeys = keys %hash;
```

This is often used to iterate over all elements of a hash:

```
foreach $key (keys %hash) {
    print $hash{$key}."\n";
}
```

In a scalar context, the **keys** function gives the number of elements in the hash.

Conversely, the **values** function returns a list of all current *values* of the argument hash:

```
@hashvals = values(%hash);
```

The **each** function provides another means of iterating over the elements in a hash:

```
while (($key, $value) = each (%hash)) {
    statements;
}
```

You can remove elements from a hash using the **delete** function:

```
delete $hash{'key'};
```

Pattern Matching

Regular Expressions

Regular expressions are patterns to be matched against a string. The two basic operations performed using patterns are matching and substitution:

Matching **/pattern/**
Substitution **s/pattern/newstring/**

The simplest kind of regular expression is a literal string. More complicated expressions include *metacharacters* to represent other characters or combinations of them.

The [...] construct is used to list a set of characters (a *character class*) of which *one* will match. Ranges of characters are denoted with a hyphen (-), and a negation is denoted with a circumflex (^). Examples of character classes are shown below:

[a-zA-Z] Any single letter
[0-9] Any digit
[^0-9] Any character **not** a digit

Some common character classes have their own predefined symbols:

| Code | Matches |
|------|--|
| . | Any character |
| \d | A digit, such as [0-9] |
| \D | A nondigit, same as [^0-9] |
| \w | A word character (alphanumeric) [a-zA-Z_0-9] |
| \W | A nonword character [^a-zA-Z_0-9] |
| \s | A whitespace character [\t\n\r\f] |
| \S | A non-whitespace character [^ \t\n\r\f] |

Regular expressions also allow for the use of both variable interpolation and *backslashed representations* of certain characters:

| Code | Matches |
|------|-----------------------|
| \n | Newline |
| \r | Carriage return |
| \t | Tab |
| \f | Formfeed |
| \/ | Literal forward slash |

Anchors don't match any characters; they match places within a string.

| Assertion | Meaning |
|-----------|--|
| ^ | Matches at the beginning of string |
| \$ | Matches at the end of string |
| \b | Matches on word boundary |
| \B | Matches except at word boundary |
| \A | Matches at the beginning of string |
| \Z | Matches at the end of string or before a newline |
| \z | Matches only at the end of string |

Quantifiers are used to specify how many instances of the previous element can match.

| Maximal | Minimal | Allowed Range |
|--------------------|---------------------|---|
| <code>{n,m}</code> | <code>{n,m}?</code> | Must occur at least n times, but no more than m times |
| <code>{n,}</code> | <code>{n,}?</code> | Must occur at least n times |
| <code>{n}</code> | <code>{n}?</code> | Must match exactly n times |
| <code>*</code> | <code>*?</code> | 0 or more times (same as <code>{0,}</code>) |
| <code>+</code> | <code>+</code> | 1 or more times (same as <code>{1,}</code>) |
| <code>?</code> | <code>??</code> | 0 or 1 time (same as <code>{0,1}</code>) |

It is important to note that quantifiers are greedy by nature. If two quantified patterns are represented in the same regular expression, the leftmost is greediest. To force your quantifiers to be non-greedy, append a question mark.

If you are looking for two possible patterns in a string, you can use the alternation operator (`|`). For example,

```
/you|me|him|her/;
```

will match against any one of these four words. You may also use parentheses to provide boundaries for alternation:

```
/And(y|rew)/;
```

will match either “Andy” or “Andrew”.

Parentheses are used to group characters and expressions. They also have the effect of “remembering” parts of a matched pattern for further processing. To recall the “memorized” portion of the string, include a backslash followed by an integer representing the location of the parentheses in the expression:

```
/fred(.)barney\1/;
```

Outside of the expression, these “memorized” portions are accessible as the special variables `$1`, `$2`, `$3`, etc. Other special variables are as follows:

```
$&      Part of string matching regexp
$`     Part of string before the match
$'     Part of string after the match
```

Regular expression grouping precedence

```

Parentheses      ( ) (?: )
Quantifiers      ? + * {m,n} ?? +? *?
Sequence and    abc ^ $ \A \Z (?= ) (?! )
anchoring
Alternation      |
```

To select a target for matching/substitution other than the default variable (`$_`), use the `=~` operator:

```
$var =~ /pattern/;
```

Operators

m/pattern/gimosx

The “match” operator searches a string for a pattern match. The preceding “m” is usually omitted. The trailing modifiers are as follows

| Modifier | Meaning |
|-----------------|--------------------------------------|
| g | Match globally; find all occurrences |
| i | Do case-insensitive matching |
| m | Treat string as multiple lines |
| o | Only compile pattern once |
| s | Treat string as a single line |
| x | Use extended regular expressions |

s/pattern/replacement/egimosx

Searches a string for a pattern, and replaces any match with replacement. The trailing modifiers are all the same as for the match operator, with the exception of “e”, which evaluates the right-hand side as an expression. The substitution operator works on the default variable (\$_), unless the =~ operator changes the target to another variable.

tr/pattern1/pattern2/cds

This operator scans a string and, character by character, replaces any characters matching **pattern1** with those from **pattern2**. Trailing modifiers are:

| Modifier | Meaning |
|-----------------|--|
| c | Complement pattern1 |
| d | Delete found but unreplaced characters |
| s | Squash duplicated replaced characters |

This can be used to force letters to all uppercase:

```
tr/a-z/A-Z/;
```

@fields = split(pattern,\$input);

Split looks for occurrences of a regular expression and breaks the input string at those points. Without any arguments, split breaks on the whitespace in \$_:

```
@words = split;      is equivalent to  
@words = split(/\s+/, $_);
```

\$output = join(\$delimiter,@inlist);

Join, the complement of split, takes a list of values and glues them together with the provided delimiting string.

Subroutines and Functions

Subroutines are defined in Perl as:

```
sub subname {
    statement_1;
    statement_2;
}
```

Subroutine definitions are global; there are no local subroutines.

Invoking subroutines

The ampersand (&) is the identifier used to call subroutines. They may also be called by appended parentheses to the subroutine name:

```
name();
&name;
```

You may use the explicit **return** statement to return a value and leave the subroutine at any point.

```
sub myfunc {
    statement_1;
    if (condition) return $val;
    statement_2;
    return $val;
}
```

Passing arguments

Arguments to a subroutine are passed as a single, flat list of scalars, and return values are passed the same way. Any arguments passed to a subroutine come in as `@_`.

To pass lists of hashes, it is necessary to pass *references* to them:

```
@returnlist = ref_conversion(\@inlist, \%inhash);
```

The subroutine will have to *dereference* the arguments in order to access the data values they represent.

```
sub myfunc {
    my($inlistref, $inhashref) = @_;
    my(@inlist) = @$inlistref;
    my(%inhash) = %$inhashref;
    statements;
    return @result;
}
```

Prototypes allow you to design your subroutines to take arguments with constraints on the number of parameters and types of data.

Variable Scope

Any variables used in a subroutine that aren't declared private are global variables.

The **my** function declares variables that are lexically scoped within the subroutine. This means that they are private variables that only exist within the block or routine in which they are called. The **local** function declares variables that are dynamic. This means that they have global scope, but have temporary values within the subroutine. Most of the time, use **my** to localize variables within a subroutine.

Files and I/O

Filehandles

A filehandle is the name for the connection between your Perl program and the operating system. Filehandles follow the same naming conventions as labels, and occupy their own namespace.

Every Perl program has three filehandles that are automatically opened for it: STDIN, STDOUT, and STDERR:

| | |
|--------|--|
| STDIN | Standard input (keyboard or file) |
| STDOUT | Standard output (print and write send output here) |
| STDERR | Standard error (channel for diagnostic output) |

Filehandles are created using the `open()` function:

```
open(FILE,"filename");
```

You can open files for reading, writing, or appending:

```
open(FILE,"> newout.dat") Writing, creating a new file  
open(FILE,">> oldout.dat") Appending to existing file  
open(FILE,"< input.dat") Reading from existing file
```

As an aside, under Windows, there are a number of ways to refer to the full path to a file:

```
"c:\\temp\\file" Escape the backslash in double quotes  
'c:\temp\file' Use proper path in single quotes  
"c:/temp/file" UNIX-style forward slashes
```

It is important to realize that calls to the `open()` function are not always successful. Perl will not (necessarily) complain about using a filehandle created from a failed `open()`. This is why we test the condition of the open statement:

```
open(F,"< badfile.dat") or die "open: $!"
```

You may wish to test for the existence of a file or for certain properties before opening it. Fortunately, there are a number of file test operators available:

| File test | Meaning |
|------------------|------------------------------------|
| -e file | File or directory exists |
| -T file | File is a text file |
| -w file | File is writable |
| -r file | File is readable |
| -s file | File exists and has nonzero length |

These operators are usually used in a conditional expression:

```
if (-e myfile.dat) {  
    open(FILE,"< myfile.dat") or die "open: $!\n";  
}
```

Even more information can be obtained about a given file using the `stat()` function.

Using filehandles

After a file has been opened for reading you can read from it using the diamond operator just as you have already done for STDIN:

```
$_ = <FILE>;           or  
while (<FILE>) {  
    statements;  
}
```

To print to your open output file, use the filehandle as the first argument to the print statement (N.B. no commas between the filehandle and the string to print).

```
print FILE "Look Ma! No hands!\n";
```

To change the default output filehandle from STDOUT to another one, use select:

```
select FILE;
```

From this point on, all calls to print or write without a filehandle argument will result in output being directed to the selected filehandle. Any special variables related to output will also then apply to this new default. To change the default back to STDOUT, select it:

```
select STDOUT;
```

When you are finished using a filehandle, close it using close():

```
close(FILE);
```

Formats

Perl has a fairly sophisticated mechanism for generating formatted output. This involves using pictorial representations of the output, called *templates*, and the function **write**.

Using a format consists of three operations:

1. Defining the format (template)
2. Loading data to be printed into the variable portions of the format
3. Invoking the format.

Format templates have the following general form:

```
format FORMATNAME =  
fieldline  
$value_one, $value_two, ...  
.
```

Everything between the “=” and the “.” is considered part of the format and everything (in the *fieldlines*) will be printed exactly as it appears (whitespace counts). Fieldlines permit variable interpolation via *fieldholders*:

```
Hi, my name is @<<<<, and I'm @< years old.Fieldline  
$name, $age Valueline
```

Fieldholders generally begin with a @ and consist of characters indicated alignment/type.

```
@<<<<    Four character, left-justified field  
@>>>>    Four character, right-justified field  
@| | |    Four character, centered field  
@###.##   Six character numeric field, with two decimal places  
@*        Multi-line field (on line by itself – for blocks of text)  
^<<<<    Five character, “filled” field (“chews up” associated variables)
```

The name of the format corresponds to the name of a filehandle. If write is invoked on a filehandle, then the corresponding format is used. Naturally then, if you're printing to standard output, then your format name should be STDOUT. If you want to use a format with a name other than that of your desired filehandle, set the **\$~** variable to the format name.

There are special formats which are printed at the top of the page. If the active format name is FNAME, then the “top” format name is FNAME_TOP. The special variable **\$\$** keeps a count of how many times the “top” format has been called and can be used to number pages.

Manipulating files & directories

The action of opening a file for writing creates it. Perl also provides functions to manipulate files without having to ask the operating system to do it.

unlink(filename)

Delete an existing file. Unlink can take a list of files, or wildcard as an argument as well: unlink(<*.bak>)

rename(oldname, newname)

This function renames a file. It is possible to move files into other directories by specifying a path as part of the new name.

Directories also have some special function associated with them

mkdir(dirname, mode)

Create a new directory. The “mode” specifies the permissions (set this to 0777 to be safe).

rmdir(dirname)

Removes (empty) directories

chdir(dirname)

Change current working directory to dirname

File and directory attributes can be modified as well:

chmod(permission, list of files)

Change the permissions of files or directories:

666 = read and write

444 = read only

777 = read, write, and executable

utime(atime, mtime, list of files)

Modify timestamps on files or directories. “atime” is the time of the most recent access, and “mtime” is the time the file/directory was last modified.

Modules

Namespaces and Packages

Namespaces store identifiers for a package, including variables, subroutines, filehandles, and formats, so that they are distinct from those of another package. The default namespace for the body of any Perl program is `main`. You can refer to the variables from another package by “qualifying” them with the package name. To do this, place the name of the package followed by two colons before the identifier’s name:

```
$Package::varname
```

If the package name is null, the `main` package is assumed.

Modules

Modules are Perl’s answer to software packages. They extend the functionality of core Perl with additional compiled code and scripts. To make use of a package (if it’s installed on your system), call the `use` function:

```
use CGI;
```

This will pull in the module’s subroutines and variables at compile time. `use` can also take a list of strings naming entities to be imported from the module:

```
use Module qw(const1 const2 func1 func2);
```

Perl looks for modules by searching the directories listed in `@INC`.

Modules can be obtained from the Comprehensive Perl Archive Network (CPAN) at

<http://www.cpan.org/modules/>

or from the ActiveState site:

<http://www.ActiveState.com/packages/zips/>

To install modules under UNIX, unarchive the file containing the package, change into its directory and type:

```
perl Makefile.PL  
make  
make install
```

On Windows, the ActivePerl distribution makes use of the “Perl Package Manager” to install/remove/update packages. To install a package, run `ppm` on the `.ppd` file associated with the module:

```
ppm install module.ppd
```

Object-Oriented Perl

In Perl, modules and object-oriented programming go hand in hand. Not all modules are written in an object-oriented fashion, but most are. A couple of definitions are warranted here:

- An *object* is simply a referenced thingy that happens to know which class it belongs to.
- A *class* is simply a package that happens to provide methods to deal with objects.
- A *method* is simply a subroutine that expects an object reference (or a package name, for class methods) as its first argument.

To create an object (or instance of a class), use the class constructor. Usually the class constructor will be a function named “new,” but may be called “Create” for some Win32 modules. For example,

```
$tri = new Triangle::Right (side1=>3, side2=>4);
```

The constructor takes a list of arguments describing the properties of the object to be created (see the documentation of the module in question to determine what these should be) and returns a reference to the created object.

An example of a class constructor (internal to the module) is shown below:

```
package critter; # declare the name of the package  
  
sub new {  
    my $class = shift; # Get class name  
    my $self = {}; # Initialize the object to nothing  
    bless $self, $class; # Declare object to be part of class  
    $self->_initialize();# Do other initializations  
    return $self;  
}
```

Methods (subroutines expecting an object reference as their first argument) may be invoked in two ways:

```
Package->constructor(args)->method_name(args)
```

Or:

```
$object = Package->constructor(args);  
$object->method_name(args);
```

Methods are simply declared subroutines in the package source file.

Common Gateway Interfaces

Perl is the most commonly used language for CGI programming on the World Wide Web. The Common Gateway Interface (CGI) is an essential tool for creating and managing comprehensive websites. With CGI, you can write scripts that create interactive, user-driven applications.

Simple CGI Programs

CGI programs are invoked by accessing a URL (uniform resource locator) describing their coordinates:

```
http://www.mycompany.com/cgi-bin/program.plx
```

even though the actual location of the script on the hard drive of the server might be something like:

```
c:\webserver\bin\program.plx
```

The simplest CGI programs merely write HTML data to STDOUT (which is then displayed by your browser):

```
print << ENDOFTEXT;  
Content-type: text/html  
  
<HTML>  
<HEAD><TITLE>Hello, World!</TITLE></HEAD>  
<BODY>  
<H1>Hello, World!</H1>  
</BODY>  
</HTML>  
ENDOFTEXT
```

CGI.pm

CGI.pm is a Perl module written to facilitate CGI programming. It contains within itself the wherewithal to generate HTML, parse arguments and environment variables, and maintain state over multiple transactions. Another feature which is not to be underestimated is the ability to reliably run CGI programs from the command line for the purposes of debugging. A simple example of the CGI.pm module in action is shown below:

```
use CGI;  
  
$query = CGI::new();    # create CGI object  
$bday = $query->param("birthday"); # get parameters  
print $query->header(); # generate Content-type line  
print $query->p("Your birthday is $bday");
```

Passing Parameters

CGI programs really shine, however, when they take arguments and format their output depending on those arguments. To pass arguments to a CGI script, ampersand-delimited key-value pairs to the URL:

```
http://www.mycompany.com/cgi-bin/icecream.plx?flavor=mint&container=cone
```

Everything after the question mark is an argument to the script.

Environment variables provide the script with data about the server, client, and the arguments to the script. The latter are available as the “QUERY_STRING” environment variable. The following example prints all of the environment variables:

```
# Print all of the necessary header info
print <<ENDOFTEXT;
Content-type: text/html

<HTML>
<HEAD><TITLE>Environment Variables</TITLE></HEAD>
<BODY>
<CENTER><H1>Environment Variables</H1></CENTER>
<TABLE>
<TR><TH>Variable</TH><TH>Value</TH>
ENDOFTEXT

# Now loop over and format environment variables
foreach $key (sort keys %ENV) {
    print "<TR><TD>$key</TD><TD>$ENV{$key}</TD></TR>\n";
}
print "</TABLE></BODY></HTML>\n";
```

CGI.pm is particularly good at extracting parameters in a platform-independent way:

```
use CGI;

$query = CGI::new();
print $query->header();
print $query->start_html(-title=>'Form Parameters');
print $query->h1('Form Parameters');
foreach $name ( $query->param() ) {
    $value = $query->param($name);
    print "$name => $value\n";
    print $query->br();          # Insert a line break
}
print $query->end_html();
```

Database Access

There are two primary means of accessing databases under Perl. The first (and oldest) makes use of the DBM (Database Management) libraries available for most flavors of UNIX. The second (and more powerful) allows for a platform-independent interaction with more sophisticated database management systems (DBMS's) such as Oracle, Sybase, Informix, and MySQL.

DBM

A DBM is a simple database management facility for most UNIX systems. It allows programs to store a collection of key-value pairs in binary form, thus providing rudimentary database support for Perl. To use DBM databases in Perl, you can associate a hash with a DBM database through a process similar to opening a file:

```
use DB_File;
tie(%ARRAYNAME, "DB_File", "dbmfilename");
```

Once the database is opened, anything you do to the hash is immediately written to the database. To break the association of the hash with the file, use the `untie()` function.

DBI/DBD

DBI is a module that provides a consistent interface for interaction with database solutions. The DBI approach relies on database-specific drivers (DBD's) to translate the DBI calls as needed for each database. Further, actual manipulation of the contents of the database is performed by composing statements in Structured Query Language (SQL) and submitting them to the database server.

DBI methods make use of two different types of handles

1. Database handles (like filehandles)
2. Statement handles (provide means of executing statements and manipulating their results)

Database handles are created by the `connect()` method:

```
$db_handle = DBI->connect('DBI:mysql:dbname:hostname',
                          $username, $password);
```

and destroyed by the `disconnect()` method:

```
$result = $db_handle->disconnect();
```

The first argument to the `connect()` method is a string describing the data source, typically written in the form:

```
DBI:driver_name:database_name:host_name
```

Statement handles are created by the `prepare()` method

```
$st_handle = $db_handle->prepare($sql)
```

where `$sql` is a valid SQL statement, and "destroyed" using the `finish()` method.

The SQL statement is then executed using the `execute()` method

```
$result = $st_handle->execute();
```

and the results obtained using any of the `fetch()` methods:

```
@ary = $st_handle->fetchrow_array(); # fetch a single row of the
                                     # query results
```

```
$hashref = $st_handle->fetchrow_hashref();
%hash = %$hashref;
```

Note that you do not directly access the results the SQL statement, but obtain them one row at a time via the `fetch()` methods.

The following script connects to a MySQL database and prints the contents of one of its tables:

```
use DBI;
use strict;

my($dsn) = 'DBI:mysql:test:localhost'; # Data source name
my($username) = 'user';             # User name
my($password) = 'secret';          # Password
my($dbh,$sth);                      # Database and statement handles
my(@ary);                             # array for rows returned by query

# connect to database
$dbh = DBI->connect($dsn, $username, $password);

# issue query
$sth = $dbh->prepare('SELECT * FROM tablename');
$sth->execute();

# read results of query, then clean up
while(@ary = $sth->fetchrow_array()) {
    print join("\t", @ary), "\n";
}
$sth->finish();

$dbh->disconnect();
```

4/4 B.TECH CSE 1ST SEMESTER
WEB TECHNOLOGIES

UNIT-6

S.JAYA PRAKASH
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
SIRCRRCOE, ELURU

UNIT-VI

Introduction to Ruby, Variables, types, simple I/O, Control, Arrays, Hashes, Methods, Classes, Iterators, Pattern Matching, Overview of Rails.

Introduction to Ruby:

Ruby is a scripting language designed by Yukihiro Matsumoto of Japan, also known as Matz. It was released in 1996. It runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Ex: puts "welcome to ruby world"

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 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 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.

Variables:

A variable is a data name and which stores data value used by any program. Types of variables:

1. Global variables
2. Instance variables
3. Class variables
4. Local variables
5. Ruby Pseudo variables

Global variables:

Global variable are variables which are declare outside of class. Global variables begin with \$. Uninitialized global variables have the value *nil*. These are valid in all class methods.

Ex:

```
$X=5  
class A  
.....  
...  
...  
end
```

Instance variables:

These are hold with instance. Instance variables begin with @. Uninitialized instance variables have the value *nil*.

```
EX:
class A
  def show
    @X=10
    puts X
  end
end
```

Class Variables:

Class variables begin with @@ and must be initialized before they can be used in method definitions. Class variables are shared among descendants of the class or module in which the class variables are defined.

```
Ex:
class A
  @@ X=5
  .....
  .....
end
```

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 { }.

Pseudo-Variables:

They are special variables that have the appearance of local variables but behave like constants. You can not 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.

Constants:

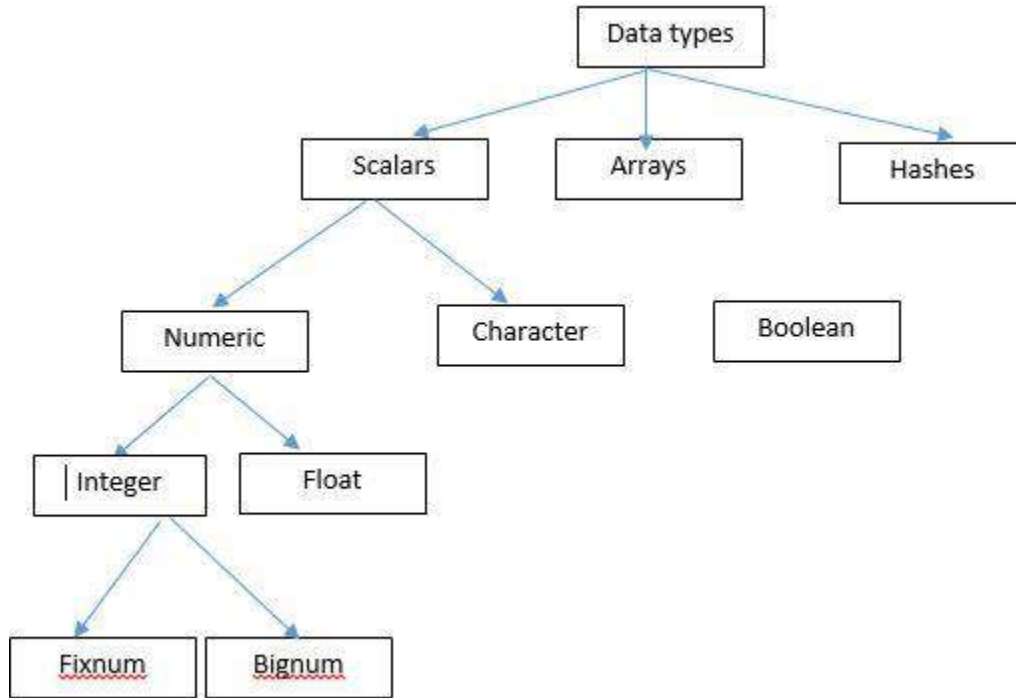
Constants variables are hold data values that never change values during the execution. Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally.

```
class Example
  VAR1 = 100
  VAR2 = 200
  def show
    puts "Value of first Constant is #{VAR1}"
    puts "Value of second Constant is #{VAR2}"
  end
end
```

Data Types:

A data type or simply type is a classification identifying one of various types of data, such as real, integer or **Boolean**, that determines the possible values for that type.

Ruby has three categories of data types scalars, arrays, and hashes.



Primitive Data Types and Operations

Ruby is a dynamically typed language. Variables are declared without type. Omit semicolons at the end of statements.

Strings

Strings are declared with double or single quotes and are similar to strings in Java.

```
1 string1 = "First String"
2 string2 = 'Second String'
```

You can add and multiple Strings:

```
1 puts string1 + string2
2 First StringSecond String</code>
3
4 puts string1 * 3
5 First StringFirst StringFirst String
```

Use the reverse method to reverse the characters in a String:

```
1 puts string2.reverse
2 gnirts dnoceS
```

Use the `gsub` method to replace pieces of Strings:

```
1 string3 = string1 * 3
2 Third StringThird StringThird String
3
4 string3 = string3.gsub("First", "Third")
5 puts string3
6 Third StringThird StringThird String
```

Numbers

Numbers can be integers or floating point and are similar numbers in Java. However, the increment and decrement operators are not available in Ruby.

To cast from Integer to Float and vice-versa, place parenthesis around the argument instead of the type:

```
1 puts Float(2)
2 2.0
```

You can also cast a String to Integer or Float type:

```
1 puts Integer("1");
2 1
```

Symbols

Symbols are lightweight strings that do not have methods and cannot be changed at runtime. You symbols if you are looking for a performance increase or you want to ensure the symbol remains constant through runtime.

Symbols are a colon followed by a string without quotes unless you have a sentence with spaces:

```
1 :'Hello World!'
```

You can obtain the String representation of a symbol by using the `to_s` method:

```
1 puts :'Hello World'.to_s
2 Hello World!
```

If you want to manipulate the string value of a symbol you must copy it into another variable:

```
1 string_symbol = :'Hello World!'.to_s
```

Arrays

You can create a literal array like:

```
1 numbers = ["one", "two", "three", "four", "five"]
```

Using the new method without parameters creates an empty array:

```
1 letters = Array.new()
```

You can create an array of a given size:

```
1 months = Array.new(12)
```

You can print the value of an array to the screen using the inspect method:

```
1 puts numbers.inspect
2 ["one", "two", "three", "four", "five"]
```

Some other popular methods are length, push, pop, reverse, rotate, sort, and more.

Hashes

Hashes are nearly identical to arrays. The only difference is a hash is an associative array, meaning indices are string based instead of integer based.

Creating a literal hash is similar to an array:

```
1 months = {"January", 1, "February", 2, "March", 3} => {"January"=>1, "February"=>2,
  "March"=>3}
```

or using

```
1 colors = Hash["blue" => 1, "green" => 2, "yellow" => 3]
```

You can also create a new hash using the new method similar to array.

You can insert entries into the hash:

```
1 colors["red"] = 1
```

To determine if a hash has a key or value, use the has_key? and has_value? methods. Other popular methods are length, clear, delete, and inspect.

Boolean

Boolean values true and false are instances of the TrueClass and FalseClass. All objects evaluate to true except for false and nil.

```
1 cars = ["Chevy", "Ford", "Toyota", "Dodge"]
2 => ["Chevy", "Ford", "Toyota", "Dodge"]
3 if cars
4   puts "cars evaluates to true!"
5 end
6 cars evaluates to true!
7
8 if false
9   puts "false is true!"
10 else
11   puts "false is false!"
```

```

11 end
12 false is false!
13

```

NOTE: Nil

Nil is the equivalent of null in Java. However, nil is an instance of the NilClass. nil and false are the only objects to evaluate to false.

Simple I/O:

Ex.

```

puts "enter a number"
A=gets
puts A.to_i

```

} integer

Ex.

```

puts "enter a number"
A=gets
puts A.to_f

```

} float

Ex.

```

puts "enter name"
A=gets
puts A

```

} string

Ex.

```

puts "enter two numbers"
A=gets
B=gets
C=A.to_i + B.to_i
puts "addition is #{C}"

```

} sum of 2 numbers

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 \'

Ex:

```

puts 'escape using "\\\"';
puts 'That\'s right';

```

output:
 escape using "\\"
 That's right

Backslash Notations:

Following is the list of Backslash notations in 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) |
| \cx, \C-x | Control-x |
| \M-x | Meta-x (c 0x80) |
| \M-\C-x | Meta-Control-x |
| \x | Character x |

String Functions:

A String object in Ruby holds and manipulates an arbitrary sequence of one or more bytes, typically representing characters that represent human language.

| S.No | Method | Action |
|------|------------|---|
| 1 | capitalize | Convert the first letter to uppercase and the rest of the letters to lowercase |
| 2 | chop | Removes the last character |
| 3 | chomp | Removes a newline from right end, if there is one |
| 4 | upcase | Converts all of the lowercase letters in the object to uppercase |
| 5 | downcase | Converts all of the uppercase letters in the object to lowercase |
| 6 | strip | Removes the spaces on both ends |
| 7 | lstrip | Removes the spaces on the left end |
| 8 | rstrip | Removes the spaces on the right end |
| 9 | reverse | Reverses the characters of the string |
| 10 | swapcase | Convert all uppercase letters to lowercase and all lowercase letters to uppercase |

Examples:

```
puts "enter a string"
S=gets
puts s.chomp
puts s.upcase
puts s.downcase
puts s.strip
puts s.reverse
puts A.ord-----→65
puts 65.chr-----→A
```

Operators and Precedence

Ruby supports a rich set of operators. Most operators are actually method calls. For example, $a + b$ is interpreted as $a.+(b)$, where the $+$ method in the object referred to by variable a is called with b as its argument. Ruby supports following operators.

1. Arithmetic Operators:

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

2. Comparison Operators:

| Operator | Description |
|----------|---|
| == | Checks if the value of two operands are equal or not, if yes then |
| != | Checks if the value of two operands are equal or not, if values |
| > | Checks if the value of left operand is greater than the value of |
| < | Checks if the value of left operand is less than the value of right |
| >= | Checks if the value of left operand is greater than or equal to |
| <= | Checks if the value of left operand is less than or equal to the |
| <=> | Combined comparison operator. Returns 0 if first operand equals |
| === | Used to test equality within a when clause of a case statement. |
| .eql? | True if the receiver and argument have both the same type and |
| equal? | True if the receiver and argument have the same object id. |

3. Assignment Operators:

| Operator | Description |
|----------|---|
| = | Simple assignment operator, Assigns values from right |
| += | Add AND assignment operator, It adds right operand to |
| -= | Subtract AND assignment operator, It subtracts right |
| *= | Multiply AND assignment operator, It multiplies right |
| /= | Divide AND assignment operator, It divides left operand |
| %= | Modulus AND assignment operator, It takes modulus |
| **= | Exponent AND assignment operator, Performs exponential |

4. Ruby Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

| Operator | Description |
|----------|---|
| & | Binary AND Operator copies a bit to the result if it exists in both |
| | Binary OR Operator copies a bit if it exists in either operand. |

| | |
|----|--|
| ^ | Binary XOR Operator copies the bit if it is set in one operand but |
| ~ | Binary Ones Complement Operator is unary and has the effect of |
| << | Binary Left Shift Operator. The left operands value is moved left |
| >> | Binary Right Shift Operator. The left operands value is moved |

5. Logical Operators:

| Operator | Description |
|----------|---|
| and | Called Logical AND operator. If both the operands are true, then the |
| or | Called Logical OR Operator. If any of the two operands are non zero, |
| && | Called Logical AND operator. If both the operands are non zero, then |
| | Called Logical OR Operator. If any of the two operands are non zero, |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its |
| not | Called Logical NOT Operator. Use to reverses the logical state of its |

6. Ternary operator:

| Operator | Description | Example |
|----------|------------------------|---------------------------------------|
| ? : | Conditional Expression | If Condition is true ? Then value X : |

7. Range operators:

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value and a range of values in between.

| Operator | Description | Example |
|----------|---|--|
| .. | Creates a range from start point to end point inclusive | 1..10 Creates a range from 1 to 10 inclusive |
| ... | Creates a range from start point to end point exclusive | 1...10 Creates a range from 1 to 9 |

8. double Colon "::" Operator:

The :: is a unary operator that allows: constants, instance methods and class methods defined within a class or module, to be accessed from anywhere outside the class or module.

Operator precedence:

| Method | Operator | Description |
|--------|-------------------------|---|
| ✓ | [] []= | Element reference, element set |
| ✓ | ** | Exponentiation |
| ✓ | ! ~ + - | Not, complement, unary plus and minus (method names for the last two are +@ and -@) |
| ✓ | * / % | Multiply, divide, and modulo |
| ✓ | + - | Plus and minus |
| ✓ | >> << | Right and left shift (<< is also used as the append operator) |
| ✓ | & | “And” (bitwise for integers) |
| ✓ | ^ | Exclusive “or” and regular “or” (bitwise for integers) |
| ✓ | <= < > >= | Comparison operators |
| ✓ | <=> == === != =~ !~ | Equality and pattern match operators |
| | && | Logical “and” |
| | | Logical “or” |
| | | Range (inclusive and exclusive) |
| | ?: | Ternary if-then-else |
| | = %= /= -= += = &= >>= | Assignment |
| | <<= *= &&= = **= ^= | |
| | not | Logical negation |
| | or and | Logical composition |
| | if unless while until | Expression modifiers |
| | begin/end | Block expression |

Table 16—Ruby operators (high to low precedence)

Ruby Comments:

Comments are lines of annotation within Ruby code that are ignored at runtime.

Single line comment:

A single line comment starts with # character and they extend from # to the end of the line as follows:

```
# This is a single line comment.
puts "Hello"
```

Multiline comment:

Multiple lines comment using **=begin** and **=end** syntax as follows:

```
puts "Hello"
=begin
This is a multiline comment and can span as many lines as you
like. But =begin and =end should come in the first line only.
=end
```

Control Structures:

A **control structure** is a block of programming that analyzes variables and chooses a direction in which to go based on given parameters. The term flow **control** details the direction the program takes.

1.if...else Statement:

Syntax:

```
if conditional [then]
    code...
[elsif conditional [then]
    code...]...
[else
    code...]
end
```

Ex:

```
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
```

2. case Statement:

Syntax:

```
case expression
[when expression [, expression ...] [then]
    code ]...
[else
    code ]
end
```

Ex:

```
$age = 5
case $age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
end
```

3.while Statement:

Syntax:

```
while conditional [do]
    code
end
```

EX:

```
$i = 0
$num = 5

while $i < $num do
  puts("Inside the loop i = #$i" )
  $i +=1
end
```

4. while modifier:

Syntax:

```
begin
  code
end while conditional
```

Ex:

```
$i = 0
$num = 5
begin
  puts("Inside the loop i = #$i" )
  $i +=1
end while $i < $num
```

5.until Statement:

```
until conditional [do]
  code
end
```

Executes code while conditional is false. An until statement's conditional is separated from code by the reserved word do, a newline, or a semicolon.

Ex :

```
$i = 0
$num = 5
until $i > $num do
  puts("Inside the loop i = #$i" )
  $i +=1;
end
```

6.until modifier:

Syntax:

```
begin
  code
end until conditional
```

Executes *code* while *conditional* is false.

If an *until* modifier follows a *begin* statement with no *rescue* or *ensure* clauses, *code* is executed once before *conditional* is evaluated.

Example:

```
$i = 0
$num = 5
begin
  puts("Inside the loop i = #$i" )
```

```
$i +=1;
end until $i > $num
```

7. for Statement:

Syntax:

```
for variable [, variable ...] in expression [do]
  code
end
```

Executes code once for each element in expression.

Example:

```
for i in 0..5
  puts "Value of local variable is #{i}"
end
```

8. Ruby *break* Statement:

Syntax: `break`

Terminates the most internal loop. Terminates a method with an associated block if called within the block

Example:

```
for i in 0..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 0
Value of local variable is 1
Value of local variable is 2
```

9. Ruby *next* Statement:

Syntax: `next`

Jumps to next iteration of the most internal loop. Terminates execution of a block if called within a block (with *yield* or call returning nil).

Example:

```
for i in 0..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 2
Value of local variable is 3
```

Value of local variable is 4

Value of local variable is 5

10. Ruby *redo* Statement:

Syntax: redo

Restarts this iteration of the most internal loop, without checking loop condition. Restarts *yield* or *call* if called within a block.

Example:

```
for i in 0..5
  if i < 2 then
    puts "Value of local variable is #{i}"
    redo
  end
end
```

This will produce the following result and will go in an infinite loop:

Value of local variable is 0

Value of local variable is 0

.....

11. Ruby *retry* Statement:

Syntax: retry

If *retry* appears in rescue clause of begin expression, restart from the beginning of the `begin` body.

```
do_something # exception raised
rescue
  # handles error
  retry # restart from beginning
end
```

If *retry* appears in the iterator, the block, or the body of the `for` expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

```
for i in 1..5
  retry if some_condition # restart from i == 1
end
```

Example:

```
for i in 1..5
  retry if i > 2
  puts "Value of local variable is #{i}"
end
```

This will produce the following result and will go in an infinite loop:

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

Classes:

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes 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:

Defining a Class in Ruby:

```
class Customer
  .....
  .....
end
```

Creating Objects in Ruby using *new* Method:

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. 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
```

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*.

Here is the example to define a Ruby method:

```
class Sample
  def function
    statement 1
    statement 2
  end
end
```

```
Ex: class Sample
  def hello
    puts "Hello Ruby!"
  end
end
```

Arrays

Ruby Arrays. Ruby arrays are ordered, integer-indexed collections of any object. Each element in an **array** is associated with and referred to by an index. **Array** indexing starts at 0, as in C or Java.

Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.

There are many ways to create or initialize an array. One way is with the *new* class method:

1. `names = Array.new(20)`
2. `nums = Array[](1, 2, 3, 4,5)`
3. `nums = Array[1, 2, 3, 4,5]`
4. `digits = Array(0..9)`

Ex: `nums = Array[](1, 2, 3, 4,5)`

`puts "#{names}"`

o/p:1 2 3 4 5

Array Built-in Methods:

An instance of Array object to call a Array method. As we have seen, following is the way to create an instance of Array object:

```
digits = Array(0..9)
num = digits.at(6)
puts num.length
puts "#{num}"
```

This will produce the following result:

10

6

Methods with Description

array[index] [or] array[start, length] [or]:

array[range] [or] array.slice(index) [or]

array.slice(start, length) [or] array.slice(range)

Returns the element at *index*, or returns a subarray starting at *start* and continuing for *length* elements, or returns a subarray specified by *range*. Negative indices count backward from the end of the array (-1 is the last element). Returns *nil* if the index (or starting index) is out of range.

array.at(index): Returns the element at index. A negative index counts from the end of self. Returns nil if the index is out of range.

array.clear: Removes all elements from array.

array.concat(other_array): Appends the elements in other_array to self.

array.delete(obj) [or]

array.delete(obj) { block }

Deletes items from self that are equal to obj. If the item is not found, returns nil. If the optional code block is given, returns the result of block if the item is not found.

array.delete_at(index)

Deletes the element at the specified index, returning that element, or nil if the index is

array.each { |item| block }

Calls block once for each element in self, passing that element as a parameter.

array.empty?

Returns true if the self array contains no elements.

array.eql?(other)

Returns true if array and other are the same object, or are both arrays with the same

array.last [or] array.last(n)

Returns the last element(s) of self. If array is empty, the first form returns nil.

array.length

Returns the number of elements in self. May be zero.

array.map { |item| block } [or]

array.map! { |item| block } [or]

array.nitems

Returns the number of non-nil elements in self. May be zero.

array.pack(aTemplateString)

Packs the contents of array into a binary sequence according to the directives in

array.pop

Removes the last element from array and returns it, or nil if array is empty.

| | |
|---|---|
| array.push(obj, ...) | Pushes (appends) the given obj onto the end of this array. This expression returns the |
| array.rassoc(key) | Searches through the array whose elements are also arrays. Compares <i>key</i> with the |
| array.reject { item block } | Returns a new array containing the items <i>array</i> for which the block is not <i>true</i> . |
| array.reject! { item block } | Deletes elements from <i>array</i> for which the block evaluates to <i>true</i> . but returns <i>nil</i> if |
| array.replace(other_array) | Replaces the contents of <i>array</i> with the contents of <i>other array</i> , truncating or |
| array.reverse | Returns a new array containing array's elements in reverse order. |
| array.reverse! | Reverses <i>array</i> in place. |
| array.select { item block } | Invokes the block passing in successive elements from array, returning an array |
| array.shift | Returns the first element of <i>self</i> and removes it (shifting all other elements down by |
| array.size | Returns the length of <i>array</i> (number of elements). Alias for length. |
| array.sort [or] array.sort { a,b block } | Returns a new array created by sorting self. |

Array pack directives:

Following table lists pack directives for use with Array#pack.

| Directive | Description |
|-----------|--|
| @ | Moves to absolute position. |
| A | ASCII string (space padded, count is width). |
| a | ASCII string (null padded, count is width). |
| B | Bit string (descending bit order). |
| b | Bit string (ascending bit order). |
| C | Unsigned char. |
| c | Char. |

| | |
|------|--|
| D, d | Double-precision float, native format. |
| E | Double-precision float, little-endian byte order. |
| e | Single-precision float, little-endian byte order. |
| F, f | Single-precision float, native format. |
| G | Double-precision float, network (big-endian) byte order. |
| g | Single-precision float, network (big-endian) byte order. |
| H | Hex string (high nibble first). |
| h | Hex string (low nibble first). |
| I | Unsigned integer. |
| i | Integer. |
| L | Unsigned long. |
| l | Long. |
| M | Quoted printable, MIME encoding (see RFC 2045). |
| m | Base64-encoded string. |
| N | Long, network (big-endian) byte order. |
| n | Short, network (big-endian) byte order. |
| P | Pointer to a structure (fixed-length string). |
| p | Pointer to a null-terminated string. |
| Q, q | 64-bit number. |
| S | Unsigned short. |
| s | Short. |
| U | UTF-8. |
| u | UU-encoded string. |
| V | Long, little-endian byte order. |
| v | Short, little-endian byte order. |
| w | BER-compressed integer \fnm. |
| X | Back up a byte. |
| x | Null byte. |
| Z | Same as a, except that null is added with *. |

Example:

Try following example to pack various data.

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
puts a.pack("A3A3A3") #=> "a b c "
puts a.pack("a3a3a3") #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc") #=> "ABC"
```

This will produce the following result:

```
a b c
abc
ABC
```

Hashes

A Hash is a collection of key-value pairs. To add, fetch, modify, and delete a value from a Hash, you refer to it with a unique key.

A hash is a data structure that stores items by associated keys. This is contrasted against arrays, which stores items by an ordered index. Entries in a hash are often referred to as key-value pairs. This creates an associative representation of data.

Most commonly, a hash is created using symbols as keys and any data types as values. All key-value pairs in a hash are surrounded by curly braces { } and comma separated.

Hashes can be created with two syntaxes. The older syntax comes with a => sign to separate the key and the value.

Hash creation

A hash can be created in two basic ways: with the new keyword or with the hash literal.

```
names = Hash.new
names[1] = "Jane"
names[2] = "Thomas"
puts names
```

The first script creates a hash and adds two key-value pairs into the hash object.

```
names = Hash.new
```

A **store** method can be used to initialize the hash with some values. It can be use instead of the square brackets.

```
names = Hash.new
names.store(1, "Jane")
names.store(2, "Thomas")
names.store(3, "Rebecca")
```

```
puts names
```

```
output: {1=>"Jane", 2=>"Thomas", 3=>"Rebeca"}
```

We create a domains hash with 5 pairs. This time both keys and values are string types.

```
domains = { "de" => "Germany",  
           "sk" => "Slovakia",  
           "hu" => "Hungary",  
           "us" => "United States",  
           "no" => "Norway"  
         }
```

Looping through a hash

There are several methods that can be used to loop through a Ruby hash.

```
stones = { 1 => "garnet", 2 => "topaz",  
          3 => "opal", 4 => "amethyst" }
```

```
stones.each { |k, v| puts "Key: #{k}, Value: #{v}" }
```

```
stones.each_key { |key| puts "#{key}" }
```

```
stones.each_value { |val| puts "#{val}" }
```

```
stones.each_pair { |k, v| puts "Key: #{k}, Value: #{v}" }
```

Hash Methods with Description

hash.[key]

Using a key, references a value from hash. If the key is not found, returns a default

hash.[key]=value

Associates the value given by *value* with the key given by *key*.

hash.clear

Removes all key-value pairs from hash.

hash.delete(key) [or]

array.delete(key) { |key| block }

Deletes a key-value pair from *hash* by *key*. If block is used, returns the result of a block if pair is not found. Compare *delete_if*.

hash.delete_if { |key,value| block }

Deletes a key-value pair from *hash* for every pair the block evaluates to *true*.

hash.each { |key,value| block }

Iterates over *hash*, calling the block once for each key, passing the key-value as a two-

hash.each_key { |value| block }

Iterates over *hash*, calling the block once for each *key*, passing *value* as a parameter.

hash.empty?

Tests whether hash is empty (contains no key-value pairs), returning *true* or *false*.

hash.fetch(key [, default]) [or]

hash.fetch(key) { |key| block }

Returns a value from *hash* for the given *key*. If the *key* can't be found, and there are no other arguments, it raises an *IndexError* exception; if *default* is given, it is returned; if the optional block is specified, its result is returned.

hash.index(value)

Returns the *key* for the given *value* in hash, *nil* if no matching value is found.

hash.indexes(keys)

Returns a new array consisting of values for the given key(s). Will insert the default

hash.indices(keys)

Returns a new array consisting of values for the given key(s). Will insert the default

hash.inspect

Returns a pretty print string version of hash.

hash.invert

Creates a new *hash*, inverting *keys* and *values* from *hash*; that is, in the new hash, the

hash.keys

Creates a new array with keys from *hash*.

hash.length

Returns the size or length of *hash* as an integer.

hash.merge(other_hash) [or]

hash.merge(other_hash) { |key, oldval, newval| block }

Returns a new hash containing the contents of *hash* and *other_hash*, overwriting pairs in hash with duplicate keys with those from *other_hash*.

hash.reject { |key, value| block }

Creates a new *hash* for every pair the *block* evaluates to *true*

| |
|--|
| hash.replace(other_hash) |
| Replaces the contents of <i>hash</i> with the contents of <i>other_hash</i> . |
| hash.select { key, value block } |
| Returns a new array consisting of key-value pairs from <i>hash</i> for which |
| hash.shift |
| Removes a key-value pair from <i>hash</i> , returning it as a two-element array. |
| hash.size |
| Returns the <i>size</i> or length of <i>hash</i> as an integer. |
| hash.sort |
| Converts <i>hash</i> to a two-dimensional array containing arrays of key-value pairs, then |
| hash.store(key, value) |
| Stores a key-value pair in <i>hash</i> . |

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. We will be discussing two iterators here, each and collect. Let's look at these in detail.

Ruby each Iterator:

The each iterator returns all the elements of an array or a hash.

Syntax:

```
collection.each do |variable|
  code
end
```

Executes code for each element in collection. Here, collection could be an array or a ruby hash.

Example:

```
ary = [1,2,3,4,5]
ary.each do |i|
  puts i
end
```

This will produce the following result:

```
1
2
3
4
5
```

Ruby collect Iterator:

The collect iterator returns all the elements of a collection.

Syntax:

```
collection = collection.collect
```

The collect method need not always be associated with a block. The collectmethod returns the entire collection, regardless of whether it is an array or a hash.

Example:

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

```
b = Array.new
```

```
b = a.collect
```

```
puts b
```

This will produce the following result:

```
1
2
3
4
5
```

NOTE: The collect method is not the right way to do copying between arrays. There is another method called a clone, which should be used to copy one array into another array.

For example, this code produces an array b containing 10 times each value in a.

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

```
b = a.collect{|x| 10*x}
```

```
puts b
```

This will produce the following result:

```
10
20
30
40
50
```

Pattern Matching

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings using a specialized syntax held in a pattern.

A regular expression literal is a pattern between slashes or between arbitrary delimiters followed by %r as follows:

Syntax:

```
/pattern/
/pattern/im # option can be specified
%r!/usr/local! # general delimited regular expression
```

Example:

```
line1 = "Cats are smarter than dogs";
```

```
line2 = "Dogs also like meat";
```

```
if ( line1 =~ /Cats(.*)/ )
```



```

puts "Line1 contains Cats"
end
if ( line2 =~ /Cats(.*)/ )
  puts "Line2 contains Dogs"
end

```

This will produce the following result:

Line1 contains Cats

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:

| Modifier | Description |
|----------|--|
| i | Ignore case when matching text. |
| o | Perform #{ } interpolations only once, the first time the regexp literal is evaluated. |
| x | Ignores whitespace and allows comments in regular expressions |
| m | Matches multiple lines. recognizing newlines as normal characters |
| u,e,s,n | Interpret 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. |

Regular-expression 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 |
| [...] | 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 |
| 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 |
| (?-imx) | Temporarily toggles off i, m, or x options within a regular |
| (?: 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 |
| \z | Matches end of string. |
| \G | Matches point where last match finished. |
| \b | Matches word boundaries when outside brackets. Matches |
| \B | Matches nonword boundaries. |

| | |
|---------------------------|--|
| <code>\n. \t. etc.</code> | Matches newlines. carriage returns. tabs. etc. |
| <code>\1...\9</code> | Matches nth grouped subexpression. |
| <code>\10</code> | Matches nth grouped subexpression if it matched already. |

Regular-expression Examples:

Literal characters:

| Example | Description |
|---------------------|--|
| <code>/rubv/</code> | Match "rubv". |
| <code>¥</code> | Matches Yen sign. Multibyte characters are supported in Ruby 1.9 |

Character classes:

| Example | Description |
|----------------------------|--|
| <code>/[Rr]ubv/</code> | Match "Rubv" or "rubv" |
| <code>/rub[ve]/</code> | Match "rubv" or "rube" |
| <code>/[aeiou]/</code> | Match any one lowercase vowel |
| <code>/[0-9]/</code> | Match any digit: same as <code>/[0123456789]/</code> |
| <code>/[a-z]/</code> | Match any lowercase ASCII letter |
| <code>/[A-Z]/</code> | Match any uppercase ASCII letter |
| <code>/[a-zA-Z0-9]/</code> | Match any of the above |
| <code>/[^aeiou]/</code> | Match anything other than a lowercase vowel |
| <code>/[^0-9]/</code> | Match anything other than a digit |

Special Character Classes:

| Example | Description |
|------------------|--|
| <code>/./</code> | Match any character except newline |
| <code>./m</code> | In multiline mode <code>.</code> matches newline, too |
| <code>^d/</code> | Match a digit: <code>/[0-9]/</code> |
| <code>^D/</code> | Match a nondigit: <code>/[^0-9]/</code> |
| <code>^s/</code> | Match a whitespace character: <code>/[\t\r\n\f]/</code> |
| <code>^S/</code> | Match nonwhitespace: <code>/[^ \t\r\n\f]/</code> |
| <code>^w/</code> | Match a single word character: <code>/[A-Za-z0-9_]/</code> |
| <code>^W/</code> | Match a nonword character: <code>/[^A-Za-z0-9_]/</code> |

Repetition Cases:

| Example | Description |
|----------------|--|
| /rubv?/ | Match "rub" or "rubv": the v is optional |
| /rubv*/ | Match "rub" plus 0 or more vs |
| /rubv+/ | Match "rub" plus 1 or more vs |
| ^d{3}/ | Match exactly 3 digits |
| ^d{3,}/ | Match 3 or more digits |
| ^d{3,5}/ | Match 3, 4, or 5 digits |

RUBY ON RAILS:

Ruby on Rails is a web application framework written in Ruby, a dynamic programming language. Ruby on Rails uses the Model-View-Controller (MVC) architecture pattern to organize application programming.

- ❖ A *model* in a Ruby on Rails framework maps to a table in a database.
- ❖ A controller is the component of Rails that responds to external requests from the web server to the application, and responds to the external request by determining which view file to render.
- ❖ A view in the default configuration of Rails is an erb file. It is typically converted to output html at run-time.
- ❖ It is a dynamic, general-purpose object-oriented programming language.
- ❖ Combines syntax inspired by Perl, also influenced by Eiffel and Lisp.
- ❖ Supports multiple programming paradigms, including functional, object oriented, imperative and reflective.
- ❖ Has a dynamic type system and automatic memory management
- ❖ Ruby is a Meta programming language.

Sample Ruby Code:

```
puts "Bhaskar"
```

```
# Output "Bhaskar"
```

```
puts "Bhaskar".upcase
```

```
# Output "Bhaskar" in upprecase
```

```
10.times do
```

```
puts "Bhaskar".upcase
```

```
end
```

```
# Output "Bhaskar" 10 times
```

Sample Ruby Code: Class

Class Employee: defining three attributes for a Employee; name, age,

```
class Employee # must be capitalized
```

```
attr_accessor :name, :age, :position
```

```
# The initialize method is the constructor
```

```
def initialize(name, age, position)
```

```
  @name = name
```

```
  @age = type
```

```
  @position = color end
```

New Employee

Creating an instance of the Employee class:

```
a = Employee.new("JAY", "23", "Test Engineer")
```

```
b = Employee.new("SAM", "24", "Test Engineer")
```

Method

To be able to describe employees, we add a method to the employee class:

```
def describe
```

```
  @name + " is of " + @age + " years"
```

```
  + " working as "
```

```
  + @position+ ".\n"
```

```
end
```

Calling Method:

To get the description of Employee, we can call Employee with the describe method attached:

```
emp = a.describe  
puts emp  
or:
```

```
puts a.describe
```

Rails:

- ❖ Rails is an open source Ruby framework for developing database-backed web applications
- ❖ The Rails framework was extracted from real-world web applications. Thus it is an easy to use and cohesive framework that's rich in functionality
- ❖ All layers in Rails are built to work together and uses a single language from top to bottom
- ❖ Everything in Rails (templates to control flow to business logic) is written in Ruby, except for configuration files

What is so special about Rails?

Other frameworks use extensive code generation, which gives users a one-time productivity boost but little else, and customization scripts let the user add customization code in only a small number of carefully selected points

- Meta programming replaces these two primitive techniques and eliminates their disadvantages.
- Ruby is one of the best languages for Meta programming, and Rails uses this capability well.

Scaffolding:

You often create temporary code in the early stages of development to help get an application up quickly and see how major components work together. Rails automatically create much of the scaffolding you'll need.

Convention over configuration:

- ❖ Most Web development frameworks for .NET or Java forces to write pages of configuration code, instead Rails doesn't need much configuration. The total configuration code can be reduced by a factor of five or more over similar Java frameworks just by following common conventions.
- ❖ Naming your data model class with the same name as the corresponding database table
- ❖ 'id' as the primary key name

Active Record framework:

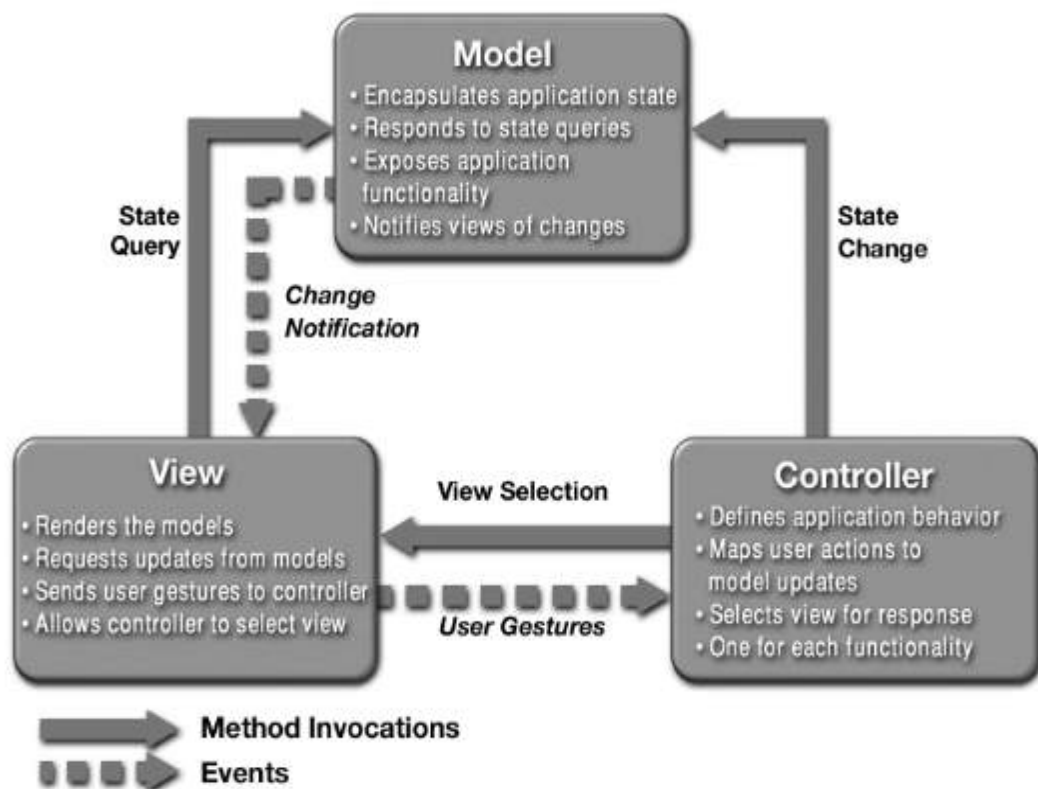
- ❖ Saves objects to the database.
- ❖ Discovers the columns in a database schema and automatically attaches them to domain objects using Meta programming.

Action Pack:

- ❖ Views and controllers have a tight interaction, in rails they are combined in Action Pack
- ❖ Action pack breaks a web request into view components and controller components
- ❖ So an action usually involves a controller request to create, read, update, or delete (CRUD) some part of the model, followed by a view request to render a page

MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE:

The MVC design pattern separates the component parts of an application MVC pattern allows rapid change and evolution of the user interface and controller separate from the data model.



Model:

- ❖ Contains the data of the application
 - Transient
 - Stored (eg Database)
- ❖ Enforces "business" rules of the application
 - Attributes
 - Work flow

View:

- ❖ Provides the user interface
- ❖ Dynamic content rendered through templates
- ❖ Three major types
 - Ruby code in erb (embedded ruby) templates
 - xml.builder templates
 - rjs templates (for javascript, and thus ajax)

Controller:

- ❖ Perform the bulk of the heavy lifting
- ❖ Handles web requests
- ❖ Maintains session state
- ❖ Performs caching
- ❖ Manages helper modules