

B.COM – COMPUTER – III YEAR - NOTES
WEB PROGRAMMING

| INDEX – UNIT - I | | |
|-------------------------|---|----------------|
| S.no: | Title | Page no: |
| 1. | HTML Programming Introduction | 02 - 03 |
| 2. | Formatting Text-Forms & Formulating Elements | 04 - 12 |
| i. | Formatting Text | 04 - 05 |
| ii. | Formatting Forms | 06 - 08 |
| iii. | Formulating Elements | 09 - 12 |
| 3. | Graphics in HTML Creating Tables & Frames | 13 - 22 |
| i. | Tables | 13 - 19 |
| ii. | Frames | 20 - 22 |
| 4. | Web Design Principles | 23 - 29 |
| 5. | PROGRAMES WITH OUTPUT: | 30 - 45 |
| i. | HTML Elements Formatting | 30 - 30 |
| ii. | HTML Text Formatting | 31 - 33 |
| iii. | HTML Tables | 34 - 39 |
| iv. | HTML Frames | 40 - 41 |
| v. | HTML Forms | 42 - 45 |

HTML Programming Introduction

HTML, which stands for **HyperText Markup Language**, is the predominant [markup language](#) for [web pages](#). HTML is the basic building-blocks of webpages.

HTML is written in the form of [HTML elements](#) consisting of *tags*, enclosed in [angle brackets](#) (like <html>), within the web page content. HTML tags normally come in pairs like <h1> and </h1>. The first tag in a pair is the *start tag*, the second tag is the *end tag* (they are also called *opening tags* and *closing tags*). In between these tags web designers can add text, tables, images, etc.

The purpose of a [web browser](#) is to read HTML documents and compose them into visual or audible web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page.

HTML elements form the building blocks of all websites. HTML allows [images and objects](#) to be embedded and can be used to create [interactive forms](#). It provides a means to create [structured documents](#) by denoting structural [semantics](#) for text such as headings, paragraphs, lists, links, quotes and other items. It can embed [scripts](#) in languages such as [JavaScript](#) which affect the behavior of HTML webpages.

Web browsers can also refer to [Cascading Style Sheets](#) (CSS) to define the appearance and layout of text and other material. The [W3C](#), maintainer of both the HTML and the CSS standards, encourages the use of CSS over explicitly presentational HTML markup.^[1]

What is HTML?

HTML is a language for describing web pages.

- HTML stands for **Hyper Text Markup Language**
- HTML is not a programming language, it is a **markup language**
- A markup language is a set of **markup tags**
- HTML uses **markup tags** to describe web pages

HTML Tags

HTML markup tags are usually called HTML tags

- HTML tags are keywords surrounded by **angle brackets** like <html>
- HTML tags normally **come in pairs** like and

- The first tag in a pair is the **start tag**, the second tag is the **end tag**
 - Start and end tags are also called **opening tags** and **closing tags**
-

HTML Documents = Web Pages

- HTML documents **describe web pages**
- HTML documents **contain HTML tags** and plain text
- HTML documents are also **called web pages**

The purpose of a web browser (like Internet Explorer or Firefox) is to read HTML documents and display them as web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page:

Formatting Text-Forms & Formulating Elements –

HTML uses tags like `` and `<i>` for formatting output, like **bold** or *italic* text.

These HTML tags are called formatting tags

Often `` renders as ``, and `` renders as `<i>`.

However, there is a difference in the meaning of these tags:

`` or `<i>` defines bold or italic text only.

`` or `` means that you want the text to be rendered in a way that the user understands as "important". Today, all major browsers render strong as bold and em as italics. However, if a browser one day wants to make a text highlighted with the strong feature, it might be cursive for example and not bold!

HTML Text Formatting Tags

These are the tags for text formats:

| | |
|--|---|
| <code>text</code> | writes text as bold |
| <code><i>text</i></code> | writes text in italics |
| <code><u>text</u></code> | writes underlined text |
| <code><sub>text</sub></code> | lowers text and makes it smaller |
| <code><sup>text</sup></code> | lifts text and makes it smaller |
| <code><blink>text</blink></code> | guess yourself! (Note: Netscape only.) |
| <code><strike>text</strike></code> | strikes a line through the text |
| <code><tt>text</tt></code> | writes text as on a classic typewriter |
| <code><pre>text</pre></code> | writes text exactly as it is, including spaces. |
| <code>text</code> | usually makes text italic |
| <code>text</code> | usually makes text bold |

HTML "Computer Output" Tags

| Tag | Description |
|---------------------------|----------------------------|
| <code><dfn></code> | Defines a definition term |
| <code><code></code> | Defines computer code text |

| | |
|---------------------------|-----------------------------------|
| <code><samp></code> | Defines sample computer code |
| <code><kbd></code> | Defines keyboard text |
| <code><var></code> | Defines a variable part of a text |
| <code><cite></code> | Defines a citation |
| <code><pre></code> | Defines preformatted text |

HTML Citations, Quotations, and Definition Tags

| Tag | Description |
|---------------------------------|--|
| <code><abbr></code> | Defines an abbreviation |
| <code><acronym></code> | Defines an acronym |
| <code><address></code> | Defines contact information for the author/owner of a document |
| <code><bdo></code> | Defines the text direction |
| <code><blockquote></code> | Defines a long quotation |
| <code><q></code> | Defines a short quotation |
| <code><cite></code> | Defines a citation |
| <code><dfn></code> | Defines a definition term |

Formatting Forms-

HTML forms are used to pass data to a server.

A form can contain input elements like text fields, checkboxes, radio-buttons, submit buttons and more. A form can also contain select lists, textarea, fieldset, legend, and label elements.

The <form> tag is used to create an HTML form:

```
<form>
.
input elements
.
</form>
```

HTML Forms - The Input Element

The most important form element is the input element.

The input element is used to select user information.

An input element can vary in many ways, depending on the type attribute. An input element can be of type text field, checkbox, password, radio button, submit button, and more.

The most used input types are described below.

Text Fields

<input type="text" /> defines a one-line input field that a user can enter text into:

```
<form>
First name: <input type="text" name="firstname" /><br />
Last name: <input type="text" name="lastname" />
</form>
```

How the HTML code above looks in a browser:

First name:
 Last name:

Note: The form itself is not visible. Also note that the default width of a text field is 20 characters.

Password Field

`<input type="password" />` defines a password field:

```
<form>
Password: <input type="password" name="pwd" />
</form>
```

How the HTML code above looks in a browser:

Password:

Note: The characters in a password field are masked (shown as asterisks or circles).

Radio Buttons

`<input type="radio" />` defines a radio button. Radio buttons let a user select ONLY ONE one of a limited number of choices:

```
<form>
<input type="radio" name="sex" value="male" /> Male<br />
<input type="radio" name="sex" value="female" /> Female
</form>
```

How the HTML code above looks in a browser:

- Male
 Female

Checkboxes

`<input type="checkbox" />` defines a checkbox. Checkboxes let a user select ONE or MORE options of a limited number of choices.

```
<form>
<input type="checkbox" name="vehicle" value="Bike" /> I have a
bike<br />
<input type="checkbox" name="vehicle" value="Car" /> I have a
car
</form>
```

How the HTML code above looks in a browser:

- I have a bike
- I have a car
-

Submit Button

`<input type="submit" />` defines a submit button.

A submit button is used to send form data to a server. The data is sent to the page specified in the form's action attribute. The file defined in the action attribute usually does something with the received input:

```
<form name="input" action="html_form_action.asp" method="get">
Username: <input type="text" name="user" />
<input type="submit" value="Submit" />
</form>
```

How the HTML code above looks in a browser:

Username:

If you type some characters in the text field above, and click the "Submit" button, the browser will send your input to a page called "html_form_action.asp". The page will show you the received input.

Formulating Elements-

Elements

Main article: [HTML element](#)

HTML documents are composed entirely of **HTML elements** that, in their most general form have three components: a pair of **tags**, a "start tag" and "end tag"; some **attributes** within the start tag; and finally, any textual and graphical *content* between the start and end tags, perhaps including other nested elements. The **HTML element** is everything between and including the start and end tags. Each **tag** is enclosed in angle brackets.

The general form of an **HTML element** is therefore: `<tag attribute1="value1" attribute2="value2">content</tag>`. Some HTML elements are defined as *empty elements* and take the form `<tag attribute1="value1" attribute2="value2" />`. Empty elements may enclose no content. The name of an HTML element is the name used in the tags. Note that the end tag's name is preceded by a slash character, "/", and that in empty elements the slash appears just before the closing >. If attributes are not mentioned, default values are used in each case.

HTML Elements

An HTML element is everything from the start tag to the end tag:

| Start tag * | Element content | End tag * |
|-------------------------|---------------------|-----------|
| <p> | This is a paragraph | </p> |
| | This is a link | |
| | | |

* The start tag is often called the **opening tag**. The end tag is often called the **closing tag**.

HTML Element Syntax

- An HTML element starts with a **start tag / opening tag**
- An HTML element ends with an **end tag / closing tag**
- The **element content** is everything between the start and the end tag
- Some HTML elements have **empty content**
- Empty elements are **closed in the start tag**

- Most HTML elements can have **attributes**

Tip: You will learn about attributes in the next chapter of this tutorial.

Nested HTML Elements

Most HTML elements can be nested (can contain other HTML elements).

HTML documents consist of nested HTML elements.

HTML Document Example

```
<html>

<body>
<p>This is my first paragraph.</p>
</body>

</html>
```

The example above contains 3 HTML elements.

HTML Example Explained

The <p> element:

```
<p>This is my first paragraph.</p>
```

The <p> element defines a paragraph in the HTML document.
The element has a start tag <p> and an end tag </p>.
The element content is: This is my first paragraph.

The <body> element:

```
<body>
<p>This is my first paragraph.</p>
</body>
```

The `<body>` element defines the body of the HTML document. The element has a start tag `<body>` and an end tag `</body>`. The element content is another HTML element (a `p` element).

The `<html>` element:

```
<html>

<body>
<p>This is my first paragraph.</p>
</body>

</html>
```

The `<html>` element defines the whole HTML document. The element has a start tag `<html>` and an end tag `</html>`. The element content is another HTML element (the `body` element).

Don't Forget the End Tag

Some HTML elements might display correctly even if you forget the end tag:

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

The example above works in most browsers, because the closing tag is considered optional.

Never rely on this. Many HTML elements will produce unexpected results and/or errors if you forget the end tag .

Empty HTML Elements

HTML elements with no content are called empty elements.

`
` is an empty element without a closing tag (the `
` tag defines a line break).

Tip: In XHTML, all elements must be closed. Adding a slash inside the start tag, like `
`, is the proper way of closing empty elements in XHTML (and XML).

HTML Tip: Use Lowercase Tags

HTML tags are not case sensitive: <P> means the same as <p>. Many web sites use uppercase HTML tags.

W3Schools use lowercase tags because the World Wide Web Consortium (W3C) **recommends** lowercase in HTML 4, and **demands** lowercase tags in XHTML.

Graphics in HTML Creating Tables & Frames-

HTML Tables:

TABLES:

Tables are defined with the <table> tag.

A table is divided into rows (with the <tr> tag), and each row is divided into data cells (with the <td> tag). td stands for "table data," and holds the content of a data cell. A <td> tag can contain text, links, images, lists, forms, other tables, etc.

Tables are used on websites for two major purposes:

- The obvious purpose of arranging information in a table
- The less obvious - but more widely used - purpose of creating a page layout with the use of hidden tables.

Using tables to divide the page into different sections is an extremely powerful tool.

Almost all major sites on the web are using invisible tables to layout the pages.

The most important layout aspects that can be done with tables are:

- Dividing the page into separate sections.
An invisible table is excellent for this purpose.
- Creating menus.
Typically with one color for the header and another for the links following in the next lines.
- Adding interactive form fields.
Typically a gray area containing a search option.
- Creating fast loading headers for the page.
A colored table with a text on it loads like a bullet compared to even a small banner.
- Easy alignment of images that have been cut into smaller pieces.

- A simple way to allow text to be written in two or more columns next to each other.

The importance of using tables for these layout purposes can't be overrated. However there are a few things to keep in mind when doing so.

Most important is, that the content of a table is not shown until the entire table is loaded. If you have extremely long pages, you should divide it into two or more tables - allowing the user to start reading the upper content while the rest of the page is loading.

Now let's proceed with learning about the different techniques...

HTML Table Tags

| Tag | Description |
|-------------------------------|---|
| <code><table></code> | Defines a table |
| <code><th></code> | Defines a table header |
| <code><tr></code> | Defines a table row |
| <code><td></code> | Defines a table cell |
| <code><caption></code> | Defines a table caption |
| <code><colgroup></code> | Defines a group of columns in a table, for formatting |
| <code><col /></code> | Defines attribute values for one or more columns in a table |
| <code><thead></code> | Groups the header content in a table |
| <code><tbody></code> | Groups the body content in a table |
| <code><tfoot></code> | Groups the footer content in a table |

HTML `<table>` Tag

[Definition and Usage](#)

The `<table>` tag defines an HTML table.

A simple HTML table consists of the table element and one or more `tr`, `th`, and `td` elements.

The `tr` element defines a table row, the `th` element defines a table header, and the `td` element defines a table cell.

A more complex HTML table may also include caption, col, colgroup, thead, tfoot, and tbody elements.

HTML **<th>** Tag

Definition and Usage

The <th> tag defines a header cell in an HTML table.

An HTML table has two kinds of cells:

- Header cells - contains header information (created with the th element)
- Standard cells - contains data (created with the [td](#) element)

The text in a th element is bold and centered.

The text in a td element is regular and left-aligned.

HTML **<tr>** Tag

Definition and Usage

The <tr> tag defines a row in an HTML table.

A tr element contains one or more [th](#) or [td](#) elements.

HTML **<td>** Tag

Definition and Usage

The <td> tag defines a standard cell in an HTML table.

An HTML table has two kinds of cells:

- Header cells - contains header information (created with the [th](#) element)
- Standard cells - contains data (created with the td element)

The text in a th element is bold and centered.

The text in a td element is regular and left-aligned.

HTML **<caption>** Tag

Definition and Usage

The <caption> tag defines a table caption.

The <caption> tag must be inserted immediately after the <table> tag. You can specify only one caption per table. Usually the caption will be centered above the table.

HTML <colgroup> Tag

Definition and Usage

The <colgroup> tag is used to group columns in a table for formatting.

The <colgroup> tag is useful for applying styles to entire columns, instead of repeating the styles for each cell, for each row.

The <colgroup> tag can only be used inside a table element.

HTML <col> Tag

Definition and Usage

The <col> tag defines attribute values for one or more columns in a table.

The <col> tag is useful for applying styles to entire columns, instead of repeating the styles for each cell, for each row.

The <col> tag can only be used inside a table or a colgroup element.

HTML <thead> Tag

Definition and Usage

The <thead> tag is used to group the header content in an HTML table.

The thead element should be used in conjunction with the [tbody](#) and [tfoot](#) elements.

The tbody element is used to group the body content in an HTML table and the tfoot element is used to group the footer content in an HTML table.

Note: <tfoot> must appear before <tbody> within a table, so that a browser can render the foot before receiving all the rows of data.

Notice that these elements will not affect the layout of the table by default. However, you can use CSS to let these elements affect the table's layout.

HTML <tbody> Tag

Definition and Usage

The <tbody> tag is used to group the body content in an HTML table.

The tbody element should be used in conjunction with the [thead](#) and [tfoot](#) elements.

The thead element is used to group the header content in an HTML table and the tfoot element is used to group the footer content in an HTML table.

Note: <tfoot> must appear before <tbody> within a table, so that a browser can render the foot before receiving all the rows of data.

Notice that these elements will not affect the layout of the table by default. However, you can use CSS to let these elements affect the table's layout.

HTML <tfoot> Tag

Definition and Usage

The <tfoot> tag is used to group the footer content in an HTML table.

The tfoot element should be used in conjunction with the [thead](#) and [tbody](#) elements.

The thead element is used to group the header content in an HTML table and the tbody element is used to group the body content in an HTML table.

Note: <tfoot> must appear before <tbody> within a table, so that a browser can render the foot before receiving all the rows of data.

Notice that these elements will not affect the layout of the table by default. However, you can use CSS to let these elements affect the table's layout.

Table Example

```
<table border="1">
<tr>
<td>row 1, cell 1</td>
<td>row 1, cell 2</td>
</tr>
<tr>
```

```
<td>row 2, cell 1</td>
<td>row 2, cell 2</td>
</tr>
</table>
```

How the HTML code above looks in a browser:

| | |
|---------------|---------------|
| row 1, cell 1 | row 1, cell 2 |
| row 2, cell 1 | row 2, cell 2 |

HTML Tables and the Border Attribute

If you do not specify a border attribute, the table will be displayed without borders. Sometimes this can be useful, but most of the time, we want the borders to show.

To display a table with borders, specify the border attribute:

```
<table border="1">
<tr>
<td>Row 1, cell 1</td>
<td>Row 1, cell 2</td>
</tr>
</table>
```

HTML Table Headers

Header information in a table are defined with the <th> tag.

All major browsers will display the text in the <th> element as bold and centered.

```
<table border="1">
<tr>
<th>Header 1</th>
<th>Header 2</th>
</tr>
<tr>
<td>row 1, cell 1</td>
<td>row 1, cell 2</td>
```

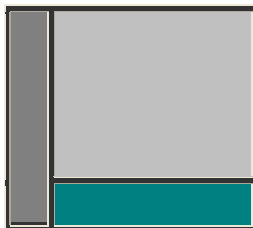
```
</tr>  
<tr>  
<td>row 2, cell 1</td>  
<td>row 2, cell 2</td>  
</tr>  
</table>
```

How the HTML code above looks in your browser:

| Header 1 | Header 2 |
|---------------|---------------|
| row 1, cell 1 | row 1, cell 2 |
| row 2, cell 1 | row 2, cell 2 |

HTML Frames:

Frames can divide the screen into separate windows.



Each of these windows can contain an HTML document.

A file that specifies how the screen is divided into frames is called a frameset.

If you want to make a homepage that uses frames you should:

- make an HTML document with the frameset
- make the normal HTML documents that should be loaded into each of these frames.

When a frameset page is loaded, the browser automatically loads each of the pages associated with the frames.

With frames, you can display more than one HTML document in the same browser window. Each HTML document is called a frame, and each frame is independent of the others.

The disadvantages of using frames are:

- Frames are not expected to be supported in future versions of HTML
- Frames are difficult to use. (Printing the entire page is difficult).
- The web developer must keep track of more HTML documents

The HTML frameset Element

The frameset element holds one or more frame elements. Each frame element can hold a separate document.

The frameset element states HOW MANY columns or rows there will be in the frameset, and HOW MUCH percentage/pixels of space will occupy each of them.

The HTML frame Element

The <frame> tag defines one particular window (frame) within a frameset.

In the example below we have a frameset with two columns.

The first column is set to 25% of the width of the browser window. The second column is set to 75% of the width of the browser window. The document "frame_a.htm" is put into the first column, and the document "frame_b.htm" is put into the second column:

Basic Notes - Useful Tips

Tip: If a frame has visible borders, the user can resize it by dragging the border. To prevent a user from doing this, you can add noresize="noresize" to the <frame> tag.

Note: Add the <noframes> tag for browsers that do not support frames.

Important: You cannot use the <body></body> tags together with the <frameset></frameset> tags! However, if you add a <noframes> tag containing some text for browsers that do not support frames, you will have to enclose the text in <body></body> tags! See how it is done in the first example below.

HTML Frame Tags

| Tag | Description |
|---|--|
| <u><frameset></u> | Defines a set of frames |
| <u><frame /></u> | Defines a sub window (a frame) |
| <u><noframes></u> | Defines a noframe section for browsers that do not handle frames |

HTML **<frameset>** Tag

Definition and Usage

The `<frameset>` tag defines a frameset.

The frameset element holds one or more frame elements. Each frame element can hold a separate document.

The frameset element states HOW MANY columns or rows there will be in the frameset, and HOW MUCH percentage/pixels of space will occupy each of them.

HTML **<frame>** Tag

Definition and Usage

The `<frame>` tag defines one particular window (frame) within a frameset.

Each frame in a frameset can have different attributes, such as border, scrolling, the ability to resize, etc.

HTML **<noframes>** Tag

Definition and Usage

The `<noframes>` tag is used for browsers that do not handle frames.

The noframes element can contain all the elements that you can find inside the body element of a normal HTML page.

The noframes element is most used to link to a non-frameset version of the web site or to display a message to users that frames are required.

The noframes element goes inside the frameset element.

Web Design Principles-

Web design is a broad term used to encompass the way that content (usually hypertext or hypermedia) is delivered to an end-user through the World Wide Web, using a web browser or other web-enabled software to display the content. The intent of web design is to create a website—a collection of online content including documents and applications that reside on a web server/servers. A website may include text, images, sounds and other content, and may be interactive.

1. Precedence (Guiding the Eye)

Good Web design, perhaps even more than other type of design, is about information.

One of the biggest tools in your arsenal to do this is *precedence*. When navigating a good design, the user should be led around the screen by the designer. I call this precedence, and it's about how much visual weight different parts of your design have.

A simple example of precedence is that in most sites, the first thing you see is the logo. This is often because it's large and set at what has been shown in studies to be the first place people look (the top left). This is a good thing since you probably want a user to immediately know what site they are viewing.

But precedence should go much further. You should direct the user's eyes through a sequence of steps. For example, you might want your user to go from logo/brand to a primary positioning statement, next to a punchy image (to give the site personality), then to the main body text, with navigation and a sidebar taking a secondary position in the sequence.

What your user should be looking at is up to you, the Web designer, to figure out.

To achieve precedence you have many tools at your disposal:

- **Position** — Where something is on a page clearly influences in what order the user sees it.
- **Color** — Using bold and subtle colors is a simple way to tell your user where to look.
- **Contrast** — Being different makes things stand out, while being the same makes them secondary.
- **Size** — Big takes precedence over little (unless everything is big, in which case little might stand out thanks to Contrast)
- **Design Elements** — if there is a gigantic arrow pointing at something, guess where the user will look?

[2. Spacing](#)

When I first started designing I wanted to fill every available space up with stuff. Empty space seemed wasteful. In fact the opposite is true.

Spacing makes things clearer. In Web design there are three aspects of space that you should be considering:

- ***Line Spacing***

When you lay text out, the space between the lines directly affects how readable it appears. Too little space makes it easy for your eye to spill over from one line to the next, too much space means that when you finish one line of text and go to the next your eye can get lost. So you need to find a happy medium. You can control line spacing in CSS with the 'line-height' selector. Generally I find the default value is usually too little spacing. Line Spacing is technically called *leading* (pronounced ledding), which derives from the process that printers used to use to separate lines of text in ye olde days — by placing bars of lead between the lines.

- ***Padding***

Generally speaking text should never touch other elements. Images, for example, should not be touching text, neither should borders or tables.

Padding is the space between elements and text. The simple rule here is that you should *always* have space there. There are exceptions of course, in particular if the text is some sort of heading/graphic or your name is [David Carson](#) 😊 But as a general rule, putting space between text and the rest of the world makes it infinitely more readable and pleasant.

- ***White Space***

First of all, white space doesn't need to be white. The term simply refers to empty space on a page (or negative space as it's sometimes called). White space is used to give balance, proportion and contrast to a page. A lot of white space tends to make things seem more elegant and upmarket, so for example if you go to an [expensive architect site](#), you'll almost always see a lot of space. If you want to learn to use whitespace effectively, go through a magazine and look at how adverts are laid out. Ads for big brands of watches and cars and the like tend to have a lot of empty space used as an element of design.

[3. Navigation](#)

One of the most frustrating experiences you can have on a Web site is being unable to figure out where to go or where you are. I'd like to think that for most Web designers, navigation is a concept we've managed to master, but I still find some pretty bad examples out there. There are two aspects of navigation to keep in mind:

Navigation — Where can you go?

There are a few commonsense rules to remember here. Buttons to travel around a site should be easy to find – towards the top of the page and easy to identify. They should look like navigation buttons and be well described. The text of a button should be pretty clear as to where it's taking you. Aside from the common sense, it's also important to make navigation usable. For example, if you have a rollover sub-menu, ensuring a person can get to the sub-menu items without losing the rollover is important. Similarly changing the color or image on rollover is excellent feedback for a user.

Orientation — Where are you now?

There are lots of ways you can orient a user so there is no excuse not to. In small sites, it might be just a matter of a big heading or a 'down' version of the appropriate button in your menu. In a larger site, you might make use of [bread crumb trails](#), sub-headings and a site map for the truly lost.

[4. Design to Build](#)

Life has gotten a lot easier since Web designers transitioned to CSS layouts, but even now it's still important to think about how you are going to build a site when you're still in Photoshop. Consider things like:

- **Can it actually be done?**

You might have picked an amazing font for your body copy, but is it actually a standard HTML font? You might have a design that looks beautiful but is 1100px wide and will result in a horizontal scroller for the majority of users. It's important to know what can and can't be done, which is why I believe all Web designers should also build sites, at least sometimes.

- **What happens when a screen is resizes?**

Do you need repeating backgrounds? How will they work? Is the design centered or left-aligned?

- **Are you doing anything that is technically difficult?**

Even with CSS positioning, some things like vertical alignment are still a bit painful and sometimes best avoided.

- **Could small changes in your design greatly simplify how you build it?**

Sometimes moving an object around in a design can make a big difference in how you have to code your CSS later. In particular, when elements of a design cross over each other, it adds a little complexity to the build. So if your design has, say three elements and each element is completely separate from each other, it would be really easy to build. On the other hand if all three overlap each other, it might still be easy, but will probably be a bit more complicated. You should find a balance between what looks good and small changes that can simplify your build.

- **For large sites, particularly, can you simplify things?**

There was a time when I used to make image buttons for my sites. So if there was a download button, for example, I would make a little download image. In the last year or so, I've switched to using CSS to make my buttons and have never looked back. Sure, it means my buttons don't always have the flexibility I might wish for, but the savings in build time from not having to make dozens of little button images are huge.

5. Typography

Text is the most common element of design, so it's not surprising that a lot of thought has gone into it. It's important to consider things like:

- **Font Choices** — Different types of fonts say different things about a design. Some look modern, some look retro. Make sure you are using the right tool for the job.
- **Font sizes** — Years ago it was trendy to have really small text. Happily, these days people have started to realize that text is meant to be read, not just looked at. Make sure your text sizes are consistent, large enough to be read, and proportioned so that headings and sub-headings stand out appropriately.
- **Spacing** — As discussed above, spacing between lines and away from other objects is important to consider. You should also be thinking about spacing between letters, though on the Web this is of less importance, as you don't have that much control.
- **Line Length** — There is no hard and fast rule, but generally your lines of text shouldn't be too long. The longer they are, the harder they are to read. Small columns of text work much better (think about how a newspaper lays out text).

- **Color** — One of my worst habits is making low-contrast text. It looks good but doesn't read so well, unfortunately. Still, I seem to do it with every Web site design I've ever made, tsk tsk tsk.
- **Paragraphing** — Before I started designing, I loved to justify the text in everything. It made for nice edges on either side of my paragraphs. Unfortunately, justified text tends to create weird gaps between words where they have been auto-spaced. This isn't nice for your eye when reading, so stick to left-aligned unless you happen to have a magic body of text that happens to space out perfectly.

6. Usability

Web design ain't just about pretty pictures. With so much information and interaction to be effected on a Web site, it's important that you, the designer, provide for it all. That means making your Web site design usable.

We've already discussed some aspects of usability – navigation, precedence, and text. Here are three more important ones:

- **Adhering to Standards**
There are certain things people expect, and not giving them causes confusion. For example, if text has an underline, you expect it to be a link. Doing otherwise is not good usability practice. Sure, you can break some conventions, but most of your Web site should do exactly what people expect it to do!
- **Think about what users will actually do**
Prototyping is a common tool used in design to actually 'try' out a design. This is done because often when you actually use a design, you notice little things that make a big difference. ALA had an article a while back called Never Use a Warning When You Mean Undo, which is an excellent example of a small interface design decision that can make life suck for a user.
- **Think about user tasks**
When a user comes to your site what are they actually trying to do? List out the different types of tasks people might do on a site, how they will achieve them, and how easy you want to make it for them. This might mean having really common tasks on your homepage (e.g. 'start shopping', 'learn about what we do,' etc.) or it might mean ensuring something like having a search box always easily accessible. At the end of the

day, your Web design is a tool for people to use, and people don't like using annoying tools!

[7. Alignment](#)

Keeping things lined up is as important in Web design as it is in print design. That's not to say that *everything* should be in a straight line, but rather that you should go through and try to keep things consistently placed on a page. Aligning makes your design more ordered and digestible, as well as making it seem more polished.

You may also wish to base your designs on a specific grid. I must admit I don't do this consciously – though obviously a site like Psdtuts+ does in fact have a very firm grid structure. This year I've seen a few really good articles on grid design including SmashingMagazine's [Designing with Grid-Based Approach](#) & A List Apart's [Thinking Outside The Grid](#). In fact, if you're interested in grid design, you should definitely pay a visit to the aptly named [DesignByGrid.com](#) home to all things gridy.

[8. Clarity \(Sharpness\)](#)

Keeping your design crisp and sharp is super important in Web design. And when it comes to clarity, it's all about the pixels.

In your CSS, everything will be pixel perfect so there's nothing to worry about, but in Photoshop it is not so. To achieve a sharp design you have to:

- Keep shape edges snapped to pixels. This might involve manually cleaning up shapes, lines, and boxes if you're creating them in Photoshop.
- Make sure any text is created using the appropriate anti-aliasing setting. I use 'Sharp' a lot.
- Ensuring that contrast is high so that borders are clearly defined.
- Over-emphasizing borders just slightly to exaggerate the contrast.

[9. Consistency](#)

Consistency means making everything match. Heading sizes, font choices, coloring, button styles, spacing, design elements, illustration styles, photo choices, etc. Everything should be themed to make your design coherent between pages and on the same page.


Keeping your design consistent is about being professional. Inconsistencies in a design are like spelling mistakes in an essay. They just lower the perception of quality. Whatever your design looks like, keeping it consistent will always bring it up a notch. Even if it's a bad design, at least make it a consistent, bad design.

The simplest way to maintain consistency is to make early decisions and stick to them. With a really large site, however, things can change in the design process. When I designed [FlashDen](#), for example, the process took months, and by the end some of my ideas for buttons and images had changed, so I had to go back and revise earlier pages to match later ones exactly.

Having a good set of CSS stylesheets can also go a long way to making a consistent design. Try to define core tags like `<h1>` and `<p>` in such a way as to make your defaults match properly and avoid having to remember specific class names all the time.

PROGRAMES OF UNIT - I

HTML Elements

| PROGRAM | OUTPUT |
|--|---|
| HTML Headings <pre><html> <body> <h1>This is heading 1</h1> <h2>This is heading 2</h2> <h3>This is heading 3</h3> <h4>This is heading 4</h4> <h5>This is heading 5</h5> <h6>This is heading 6</h6> </body> </html></pre> | <p>This is heading 1</p> <p>This is heading 2</p> <p>This is heading 3</p> <p>This is heading 4</p> <p>This is heading 5</p> <hr/> <p>This is heading 6</p> |
| HTML Paragraphs <pre><html> <body> <p>This is a paragraph.</p> <p>This is a paragraph.</p> <p>This is a paragraph.</p> </body> </html></pre> | <p>This is a paragraph.</p> <p>This is a paragraph.</p> <p>This is a paragraph.</p> |
| HTML Links <pre><html> <body> This is a link </body> </html></pre> | <p>This is a link</p> |
| HTML Images <pre><html> <body> </body> </html></pre> |  |

HTML Text Formatting

| | |
|---|---|
| <pre><html> <body> <p>This text is bold</p> <p>This text is strong</p> <p><big>This text is big</big></p> <p><i>This text is italic</i></p> <p><small>This text is small</small></p> <p><tt>This text is teletype</tt></p> <p>This text is emphasized</p> <p><code>Thisiscomputer output</code></p> <p>This is<sub> subscript</sub> and <sup>superscript</sup></p> </body> </html></pre> | <p>This text is bold</p> <p>This text is strong</p> <p>This text is big</p> <p><i>This text is italic</i></p> <p>This text is small</p> <p>This text is teletype</p> <p><i>This text is emphasized</i></p> <p>This is computer output</p> <p>This is _{subscript} and ^{superscript}</p> |
|---|---|

| | |
|---|--|
| <p>HTML "Computer Output" Tags</p> <pre><html> <body> <dfn>Definition term</dfn> <samp>Sample computer code text</samp> <kbd>Keyboard text</kbd> <var>Variable</var> <cite>Citation</cite> </body> </html></pre> | <p><i>Definition term</i> Sample computer code text Keyboard text <i>Variable</i> <i>Citation</i></p> |
| <p>HTML <pre> Tag</p> <pre><html> <body> <pre> Text in a pre element is displayed in a fixed-width font, and it preserves both spaces and line breaks </pre> <p>The pre element is often used to display computer code:</p> <pre>for i = 1 to 10 print i next i </pre> </body> </html></pre> | <p>Text in a pre element is displayed in a fixed-width font, and it preserves both spaces and line breaks</p> <p>The pre element is often used to display computer code:</p> <pre>for i = 1 to 10 print i next i</pre> |
| <p>HTML Citations, Quotations, and Definition Tags</p> <pre><html> <body> The <abbr title="World Health Organization">WHO</abbr> was founded in 1948. Can I get this <acronym title="as soon as possible">ASAP</acronym>? <address> Written by saif4u.webs.com Email</pre> | <p>The WHO was founded in 1948. Can I get this ASAP? <i>Written by saif4u.webs.com</i> Email us <i>Address: Box 564, Disneyland</i> <i>Phone: +12 34 56 78</i> Here is some Hebrew text that should be written from right-to-left!</p> |


```
us</a><br />
```

```
Address: Box 564, Disneyland<br />
```

```
Phone: +12 34 56 78
```

```
</address>
```

```
<bdo dir="rtl">Here is some Hebrew text that  
should be written from right-to-  
left!</bdo><br>
```

```
Here comes a long quotation:
```

```
<blockquote>
```

```
This is a long quotation. This is a long  
quotation. This is a long quotation. This is a  
long quotation. This is a long quotation.
```

```
</blockquote>
```

```
Notice that a browser inserts white space  
before and after a blockquote element. It also  
inserts margins for the blockquote element.
```

```
</body>
```

```
</html>
```

Here comes a long quotation:

This is a long quotation. This is a long quotation. This is a long quotation. This is a long quotation. This is a long quotation. Notice that a browser inserts white space before and after a blockquote element. It also inserts margins for the blockquote element.

HTML Tables

| Program | Output | | | | |
|--|--|-------|---------|---------|-------|
| <pre><html> <body> <table border="1"> <tr> <th>Month</th> <th>Savings</th> </tr> <tr> <td>January</td> <td>\$100</td> </tr> </table> </body> </html></pre> | <table border="1"><thead><tr><th>Month</th><th>Savings</th></tr></thead><tbody><tr><td>January</td><td>\$100</td></tr></tbody></table> | Month | Savings | January | \$100 |
| Month | Savings | | | | |
| January | \$100 | | | | |

HTML <caption> Tag

```

<html>
<body>
<table border="1">
  <caption>Monthly savings</caption>
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$50</td>
  </tr>
</table>
</body>
</html>

```

Monthly savings

| Month | Savings |
|----------|---------|
| January | \$100 |
| February | \$50 |

HTML <colgroup> Tag

```

<html>
<body>
<table width="100%" border="1">
  <colgroup span="2" align="left"></colgroup>
  <colgroup align="right"
  style="color:#0000FF;"></colgroup>
  <tr>
    <th>ISBN</th>
    <th>Title</th>
    <th>Price</th>
  </tr>
  <tr>
    <td>3476896</td>
    <td>My first HTML</td>
    <td>$53</td>
  </tr>
  <tr>
    <td>2489604</td>
    <td>My first CSS</td>
    <td>$47</td>
  </tr>
</table>
</body>
</html>

```

| ISBN | Title | Price |
|---------|---------------|-------|
| 3476896 | My first HTML | \$53 |
| 2489604 | My first CSS | \$47 |

HTML <col> Tag

```

<html>
<body>
<table width="100%" border="1">
  <col align="left" />
  <col align="left" />
  <col align="right" />
  <tr>
    <th>ISBN</th>
    <th>Title</th>
    <th>Price</th>
  </tr>
  <tr>
    <td>3476896</td>
    <td>My first HTML</td>
    <td>$53</td>
  </tr>
  <tr>
    <td>2489604</td>
    <td>My first CSS</td>
    <td>$47</td>
  </tr>
</table>
</body>
</html>

```

| ISBN | Title | Price |
|---------|---------------|-------|
| 3476896 | My first HTML | \$53 |
| 2489604 | My first CSS | \$47 |

HTML <thead> Tag

```

<html>
<head>
<style type="text/css">
thead {color:green}
tbody {color:blue;height:50px}
tfoot {color:red}
</style>
</head>
<body>
<table border="1">
  <thead>
    <tr>
      <th>Month</th>
      <th>Savings</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <td>Sum</td>
      <td>$180</td>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>January</td>
      <td>$100</td>
    </tr>
    <tr>
      <td>February</td>
      <td>$80</td>
    </tr>
  </tbody>
</table>
</body>
</html>

```

| Month | Savings |
|----------|---------|
| Sum | \$180 |
| January | \$100 |
| February | \$80 |

```
</tr>
</tfoot>
<tbody>
<tr>
  <td>January</td>
  <td>$100</td>
</tr>
<tr>
  <td>February</td>
  <td>$80</td>
</tr>
</tbody>
</table>
</body>
</html>
```

HTML <tbody> Tag

```

<html>
<head>
<style type="text/css">
thead { color:green}
tbody { color:blue;height:50px}
tfoot { color:red}
</style>
</head>
<body>
<table border="1">
  <thead>
    <tr>
      <th>Month</th>
      <th>Savings</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <td>Sum</td>
      <td>$180</td>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>January</td>
      <td>$100</td>
    </tr>
    <tr>
      <td>February</td>
      <td>$80</td>
    </tr>
  </tbody>
</table>
</body>
</html>

```

| Month | Savings |
|----------|---------|
| Sum | \$180 |
| January | \$100 |
| February | \$80 |

HTML <tfoot> Tag

```

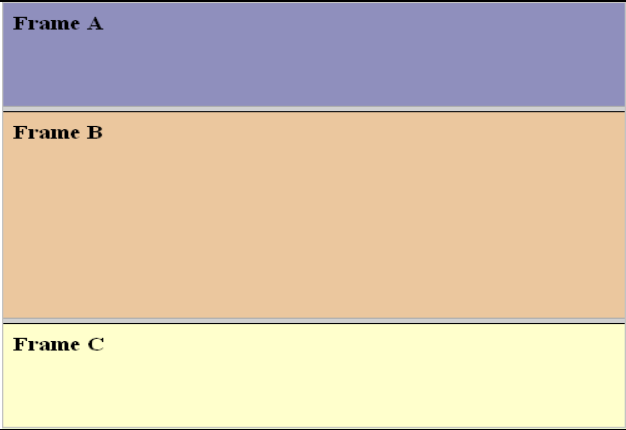
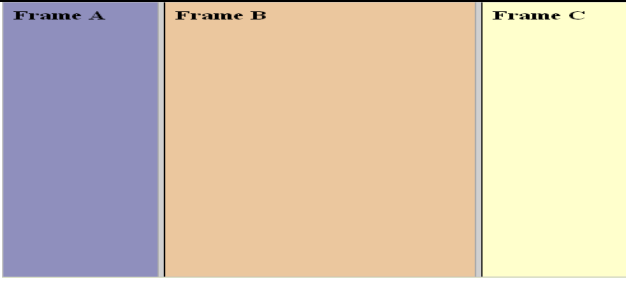
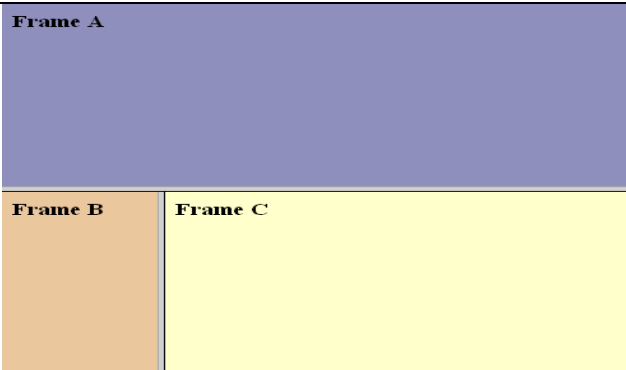
<html>
<head>
<style type="text/css">
thead { color:green}
tbody { color:blue;height:50px}
tfoot { color:red}
</style>
</head>
<body>

```

| Month | Savings |
|----------|---------|
| Sum | \$180 |
| January | \$100 |
| February | \$80 |

```
<table border="1">
<thead>
<tr>
<th>Month</th>
<th>Savings</th>
</tr>
</thead>
<tfoot>
<tr>
<td>Sum</td>
<td>$180</td>
</tr>
</tfoot>
<tbody>
<tr>
<td>January</td>
<td>$100</td>
</tr>
<tr>
<td>February</td>
<td>$80</td>
</tr>
</tbody>
</table>
</body>
</html>
```

HTML Frames:

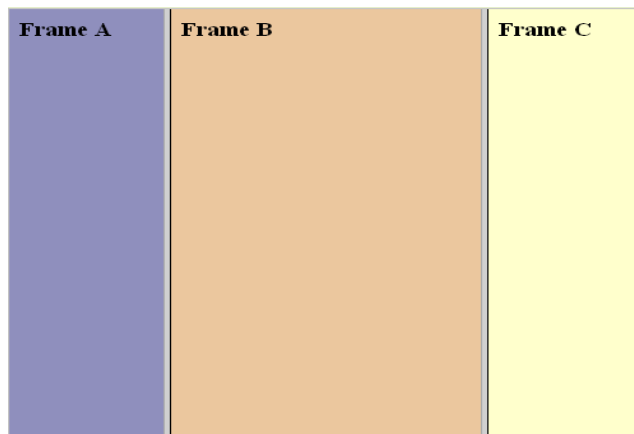
| Program | Output |
|---|--|
| <p>HTML <Horizontal frameset> Tag</p> <pre><html> <frameset rows="25%,50%,25%"> <frame src="frame_a.htm" /> <frame src="frame_b.htm" /> <frame src="frame_c.htm" /> </frameset> </html></pre> |  |
| <p>HTML <Vertical frameset> Tag</p> <pre><html> <frameset cols="25%,50%,25%"> <frame src="frame_a.htm" /> <frame src="frame_b.htm" /> <frame src="frame_c.htm" /> </frameset> </html></pre> |  |
| <p>HTML <frameset in rows & columns> Tag</p> <pre><html> <frameset rows="50%,50%"> <frame src="frame_a.htm" /> <frameset cols="25%,75%"> <frame src="frame_b.htm" /> <frame src="frame_c.htm" /> </frameset> </frameset> </html></pre> |  |

HTML <noframes> Tag

```

<html>
<frameset cols="25%,50%,25%">
  <frame src="frame_a.htm" />
  <frame src="frame_b.htm" />
  <frame src="frame_c.htm" />
</noframes>
  Sorry, your browser does not handle
  frames!
</noframes>
</frameset>
</html>

```

**HTML <navigation frameset> Tag**

```

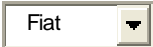

<html>
<frameset cols="120,*">
  <frame src="tryhtml_contents.htm" />
  <frame src="frame_a.htm"
  name="showframe" />
</frameset>
</html>

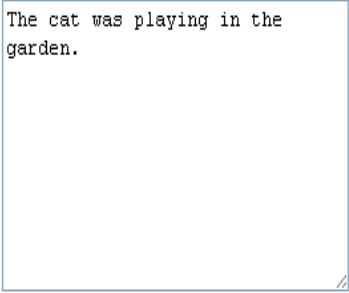
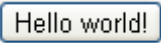
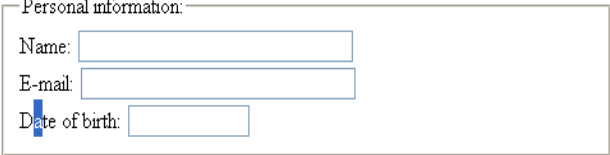
```



HTML Forms:

| Program | Output |
|---|---|
| <p style="text-align: center;">TEXT FIELD</p> <pre><html> <body> <form action=""> First name: <input type="text" name="firstname" /> Last name: <input type="text" name="lastname" /> </form> <p>Note: The form itself is not visible. Also note that the default width of a text field is 20 characters.</p> </body> </html></pre> | <p>First name: <input type="text"/></p> <p>Last name: <input type="text"/></p> <p>Note: The form itself is not visible. Also note that the default width of a text field is 20 characters.</p> |
| <p style="text-align: center;">PASSWORD FIELD</p> <pre><html> <body> <form action=""> Username: <input type="text" name="user" /> Password: <input type="password" name="password" /> </form> <p>Note: The characters in a password field are masked (shown as asterisks or circles).</p> </body> </html></pre> | <p>Username: <input type="text"/></p> <p>Password: <input type="password"/></p> <p>Note: The characters in a password field are masked (shown as asterisks or circles).</p> |
| <p style="text-align: center;">CHECKBOXES</p> <pre><html> <body> <form action=""> <input type="checkbox" name="vehicle" value="Bike" /> I have a bike <input type="checkbox" name="vehicle" value="Car" /> I have a car </form> </body> </html></pre> | <p><input checked="" type="checkbox"/> I have a bike</p> <p><input type="checkbox"/> I have a car</p> |
| <p style="text-align: center;">RADIO BUTTONS</p> <pre><html> <body> <form action=""></pre> | <p><input type="radio"/> Male</p> <p><input type="radio"/> Female</p> |

| | |
|---|---|
| <pre><input type="radio" name="sex" value="male" /> Male <input type="radio" name="sex" value="female" /> Female </form> <p>Note: When a user clicks on a radio-button, it becomes checked, and all other radio-buttons with equal name become unchecked.</p> </body> </html></pre> | <p>Note: When a user clicks on a radio-button, it becomes checked, and all other radio-buttons with equal name become unchecked.</p> |
| <p style="text-align: center;">SIMPLE DROP-DOWN LIST</p> <pre><html> <body> <form action=""> <select name="cars"> <option value="volvo">Volvo</option> <option value="saab">Saab</option> <option value="fiat">Fiat</option> <option value="audi">Audi</option> </select> </form> </body> </html></pre> |  |
| <p style="text-align: center;">Drop-down list with a pre-selected value</p> <pre><html> <body> <form action=""> <select name="cars"> <option value="volvo">Volvo</option> <option value="saab">Saab</option> <option value="fiat" selected="selected">Fiat</option> <option value="audi">Audi</option> </select> </form> </body> </html></pre> |  |
| <p style="text-align: center;">Textarea (a multi-line text input field)</p> <pre><html> <body> <p> This example cannot be edited because our editor uses a textarea for input, and your browser does not allow</pre> | |

| | |
|---|---|
| <p>a textarea inside a textarea.</p> <pre></p> <textarea rows="10" cols="30"> The cat was playing in the garden. </textarea> </body> </html></pre> | <p>This example cannot be edited because our editor uses a textarea for input, and your browser does not allow a textarea inside a textarea.</p>  |
| <p style="text-align: center;">Create a Button</p> <pre><html> <body> <form action=""> <input type="button" value="Hello world!"> </form> </body> </html></pre> |  |
| <p style="text-align: center;">Draw a border around form-data</p> <pre><html> <body> <form action=""> <fieldset> <legend>Personal information:</legend> Name: <input type="text" size="30" /> E-mail: <input type="text" size="30" /> Date of birth: <input type="text" size="10" /> </fieldset> </form> </body> </html></pre> |  |

| Send e-mail from a form | Send e-mail to someone@example.com: |
|---|--|
| <pre> <html> <body> <h3>Send e-mail to someone@example.com:</h3> <form action="MAILTO:someone@example.com " method="post" enctype="text/plain"> Name: <input type="text" name="name" value="your name" /> E-mail: <input type="text" name="mail" value="your email" /> Comment: <input type="text" name="comment" value="your comment" size="50" /> <input type="submit" value="Send"> <input type="reset" value="Reset"> </form> </body> </html> </pre> | <p>Name:</p> <input type="text" value="your name"/> <p>E-mail:</p> <input type="text" value="your email"/> <p>Comment:</p> <input type="text" value="your comment"/> <p><input type="button" value="Send"/> <input type="button" value="Reset"/></p> |

KNOWLEDGE IS POWER
MOHAMMED.SAIFUDDIN SALMAN
AVANTHI DEGREE COLLEGE
<http://www.saif4u.webs.com>
<http://www.saif2010.webs.com>
B.COM COMPUTER – III YEAR
 2011-2012

B.COM – COMPUTER – III YEAR - NOTES WEB PROGRAMMING

INDEX – UNIT - II

| S.no: | Title | Page No: |
|-------|--|----------|
| 1 | Meaning Of DHTML | 02 - 02 |
| 2 | Comparison between HTML and DHTML | 03 - 03 |
| 3 | Static and Dynamic | 04 - 04 |
| 4 | Procedural and Non Procedural | 05 - 05 |
| 5 | Cascading Style Sheets (CSS) | 06 - 06 |
| 6 | Document Object Model (DOM) | 07 - 08 |
| 7 | Different Types of Errors, Runtime Errors, System Errors etc., | 09 - 10 |
| 8 | PROGRAMES WITH OUTPUT: | 11 - 14 |
| i | Dynamic Changes Style | 11 - 11 |
| ii | Text | 11 - 12 |
| iii | Creating Graphics & Multimedia Effects | 13 - 14 |

Meaning Of DHTML

Dynamic HTML (DHTML) is a name for a set of technologies that Web developers can use to create Web pages that update themselves on the fly. In a Web page implemented in DHTML, fonts, positions of elements, graphics might change as you look at it in a browser. Dynamic HTML makes your Web documents more interactive.

Just as in "regular" HTML, when a user downloads a page written in DHTML, the content is stored on the user's computer for viewing in the browser. In DHTML, this content might also contain instructions for how changes should take place in the presentation of the page. Dynamic HTML therefore happens on the client-side. It doesn't involve reconnecting to the server for more information.

Dynamic HTML is used in conjunction with Cascading Style Sheets (CSS). Using coordinate positioning available in CSS, a Web developer can place HTML elements in specific locations on a page. The HTML elements positioned this way can move according to a script (written in, for example, ActiveX using VBScript; or JavaScript) as the user views the page. This positioning can occur not only on the X-Y (flat plane) of the Web page, but in the Z direction, giving the illusion of depth or the stacking of one element on top of another.

There are differences in how the major players--Microsoft and Netscape--implement DHTML. Netscape had proposed a new proprietary element called LAYER to deal with Z-positioning (the LAYER element doesn't seem like it will be used widely). Microsoft agrees to the non-proprietary position of the W³C.

Dynamic HTML relates to the W³C's broader efforts to create an interface called the [Document Object Model \(DOM\)](#). The aim of DOM is to allow programmers to dynamically update the content, structure, and style of documents.

Comparison between HTML and DHTML

<http://www.saif4u.webs.com>

Comparison between HTML and DHTML

HTML

HTML stands for Hyper Text Markup Language, HTML is not a programming language, it is a markup language, A markup language is a set of markup tags HTML uses markup tags to describe web pages

HTML documents describe web pages, contain HTML tags and plain text
HTML documents are also called web pages

DHTML

DHTML is the art of combining HTML, JavaScript, DOM, and CSS

DHTML is Not a Language

DHTML stands for Dynamic HTML.

DHTML is NOT a language or a web standard.

DHTML is a TERM used to describe the technologies used to make web pages dynamic and interactive.

To most people DHTML means the combination of HTML, JavaScript, DOM, and CSS.

Dynamic HTML is a collective term for a combination of new Hypertext Markup Language (HTML) tags and options, that will let you create Web pages more animated and more responsive to user interaction than previous versions of HTML. Much of dynamic HTML is specified in HTML 4.0. Simple examples of dynamic HTML pages would include (1) having the color of a text heading change when a user passes a mouse over it or (2) allowing a user to "drag and drop" an image to another place on a Web page. Dynamic HTML can allow Web documents to look and act like desktop applications or multimedia productions.

Static and Dynamic

DHTML allows scripting languages to change [variables](#) in a web page's definition language, which in turn affects the look and function of otherwise "static" HTML page content, *after* the page has been fully loaded and during the viewing process. Thus the dynamic characteristic of DHTML is the way it functions while a page is viewed, not in its ability to generate a unique page with each page load.

By contrast, a [dynamic web page](#) is a broader concept — any web page generated differently for each user, load occurrence, or specific variable values. This includes pages created by client-side scripting, and ones created by [server-side scripting](#) (such as [PHP](#), [Perl](#), [JSP](#) or [ASP.NET](#)) where the web server generates content before sending it to the client.

- The easiest difference is static HTML once rendered cannot be changed on the other hand dynamic HTML can be changed.
- Static web pages cannot have database access but dynamic pages can be connected to database.
- Using static HTML no interactions persist between user and server but dynamic HTML has capability to interact with the user.
- Static HTML does not require server side coding but dynamic HTML requires server side coding.
- No animation, games can be developed using the static HTML but on the other hand dynamic HTML can perform this task.

Scripting Languages

Scripts that are executed when a document is loaded may be able to modify the document's contents dynamically. The ability to do so depends on the scripting language itself (e.g., the "document.write" statement in the HTML object model supported by some vendors). JavaScript is often used to create dynamic HTML documents. One of the more practical things JavaScript does is allow you to add user events to static pages. For example, JavaScript allows you to embed statements into a page which allow user response to certain common events such as when you hover over a link.

The following example illustrates how scripts may modify a document dynamically. The following script:

```
<SCRIPT type="text/javascript">
    document.write("<b>Date:</b> ")
    document.write(Date())
</SCRIPT>
```

The above script will display the following:

Date: Tue Jun 28 2011 00:48:35 GMT+0300 (Arab Standard Time)

Procedural and Non Procedural

Procedural languages are used in the traditional programming that is based on algorithms or a logical step-by-step process for solving a problem. A **procedural programming language** provides a programmer a means to define precisely each step in the performance of a task. Non-procedural programming languages allow users and professional programmers to specify the results they want without specifying how to solve the problem. examples are FORTRAN, C++, COBOL, ALGOL etc

Procedural languages:

Most widely used are BASIC, COBOL, and C. Uses series of English-like words to write instructions. Often called third-generation language (3GL). Programmer assigns name to sequence of instructions that tells computer what to accomplish and how to do it.

Non Procedural language

Nonprocedural language used for generating reports, performing computations, and updating files

Nonprocedural language—contains English-like instructions that retrieve data

Nonprocedural language that allows access to data in database

Popular 4GL is SQL, query language that allows users to manage data in relational DBMS

Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a [style sheet language](#) used to describe the [presentation semantics](#) (the look and formatting) of a document written in a [markup language](#). Its most common application is to style [web pages](#) written in [HTML](#) and [XHTML](#), but the language can also be applied to any kind of [XML](#) document, including [plain XML](#), [SVG](#) and [XUL](#).

CSS is designed primarily to enable the separation of document content (written in HTML or a similar markup language) from document presentation, including elements such as the [layout](#), [colors](#), and [fonts](#).^[1] This separation can improve content [accessibility](#), provide more flexibility and control in the specification of presentation characteristics, enable multiple pages to share formatting, and reduce complexity and repetition in the structural content (such as by allowing for [tableless web design](#)). CSS can also allow the same markup page to be presented in different styles for different rendering methods, such as on-screen, in print, by voice (when read out by a speech-based browser or [screen reader](#)) and on [Braille](#)-based, [tactile](#) devices. While the author of a document typically links that document to a CSS style sheet, readers can use a different style sheet, perhaps one on their own computer, to override the one the author has specified.

What is CSS

- **CSS** stands for **Cascading Style Sheets**
- Styles define **how to display** HTML elements
- Styles were added to HTML 4.0 **to solve a problem**
- **External Style Sheets** can save a lot of work
- External Style Sheets are stored in **CSS files**

Styles Solved a Big Problem

- HTML was never intended to contain tags for formatting a document.
- HTML was intended to define the content of a document, like:
 - `<h1>This is a heading</h1>`
 - `<p>This is a paragraph.</p>`
- When tags like ``, and color attributes were added to the HTML 3.2 specification, it started a nightmare for web developers. Development of large web sites, where fonts and color information were added to every single page, became a long and expensive process.
- To solve this problem, the World Wide Web Consortium (W3C) created CSS.
- In HTML 4.0, all formatting could be removed from the HTML document, and stored in a separate CSS file.
- All browsers support CSS today.

Document Object Model (DOM)

The **Document Object Model (DOM)** is a [cross-platform](#) and [language](#)-independent convention for representing and interacting with [objects](#) in [HTML](#), [XHTML](#) and [XML](#) documents.^[1] Aspects of the DOM (such as its "Elements") may be addressed and manipulated within the syntax of the programming language in use. The public interface of a DOM is specified in its [application programming interface](#) (API).

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. This is an overview of DOM-related materials here at W3C and around the web.

"Dynamic HTML" is a term used by some vendors to describe the combination of HTML, style sheets and scripts that allows documents to be animated. The W3C has received several submissions from members companies on the way in which the object model of HTML documents should be exposed to scripts. These submissions do not propose any new HTML tags or style sheet technology. The W3C DOM Activity is working hard to make sure interoperable and scripting-language neutral solutions are agreed upon.

What is the DOM

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

The DOM defines a standard for accessing documents like HTML and XML:

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

The DOM is separated into 3 different parts / levels:

- Core DOM - standard model for any structured document
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

The DOM defines the **objects and properties** of all document elements, and the **methods** (interface) to access them.

What is the HTML DOM

The HTML DOM is:

- A standard object model for HTML
- A standard programming interface for HTML
- Platform- and language-independent
- A W3C standard

The HTML DOM defines the **objects and properties** of all HTML elements, and the **methods**(interface) to access them.

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

HTML DOM EXAMPLE:

| PROGRAM | OUTPUT |
|--|--------------|
| <pre><html> <body> <script type="text/javascript"> document.write("Hello World!"); </script> </body> </html></pre> | Hello World! |

What is the XML DOM

The XML DOM defines the **objects and properties** of all XML elements, and the **methods** (interface) to access them.

XML DOM EXAMPLE:

| PROGRAM | OUTPUT |
|--|--|
| <pre><html> <body> <script type="text/javascript" src="loadxmlidoc.js"></script> </head> <body> <script type="text/javascript"> xmlIdoc=loadXMLDoc("books.xml"); document.write(xmlIdoc.getElementsByTagName("title")[0].childNodes[0].nodeValue + " "); document.write(xmlIdoc.getElementsByTagName("author")[0].childNodes[0].nodeValue + " "); document.write(xmlIdoc.getElementsByTagName("year")[0].childNodes[0].nodeValue); </script> </body> </html></pre> | Everyday Italian Giada De Laurentiis 2005 |

Different Types of Errors, Runtime Errors, System Errors etc.,

When creating scripts and web applications, error handling is an important part. If your code lacks error checking code, your program may look very unprofessional and you may be open to security risks.

Errors can roughly be divided into four different types:

- Compile-time errors
- Logical errors
- Run-time errors
- Generated errors

A compile-time error, for example a syntax error, should not cause much trouble as it is caught by the compiler.

A logical error is when a program does not behave as intended, but does not crash. An example could be that nothing happens when a button in a graphical user interface is clicked.

A run-time error is when a crash occurs. An example could be when an operator is applied to arguments of the wrong type. The Erlang programming language has built-in features for handling of run-time errors.

A run-time error can also be emulated by calling `erlang:error(Reason)` or `erlang:error(Reason, Args)` (those appeared in Erlang 5.4/OTP-R10).

A run-time error is another name for an exception of class error.

A generated error is when the code itself calls `exit/1` or `throw/1`. Note that emulated run-time errors are not denoted as generated errors here.

Generated errors are exceptions of classes `exit` and `throw`.

When a run-time error or generated error occurs in Erlang, execution for the process which evaluated the erroneous expression is stopped. This is referred to as a **failure**, that execution or evaluation **fails**, or that the process **fails**, **terminates** or **exits**. Note that a process may terminate/exit for other reasons than a failure.

A process that terminates will emit an **exit signal** with an **exit reason** that says something about which error has occurred. Normally, some information about the error will be printed to the terminal.

Run-time errors

Run-time errors are generated when execution of a particular code statement or function results in an error condition being set. Run-time errors caught by PowerBASIC include Disk access violations (i.e., trying to write data to a full disk), out of bounds array and pointer access, and Memory allocation failures. Array bounds and null-pointer checking is only performed when [#DEBUG ERROR ON](#) is used.

Run-time errors can be trapped; that is, you can cause a designated error-handling subroutine to get control should an error occur. Use the [ON ERROR](#) statement to accomplish this. This routine can "judge" what to do next based on the type of error that occurs. File-system errors (for example, disk full) in particular are prime candidates for run-time error-handling routines. They are the only errors that a thoroughly debugged program should have to deal with.

The [ERROR](#) statement (which simulates run-time errors) can be used to debug your error-handling routines. It allows you to deliberately cause an error condition to be flagged. Avoid using error numbers higher than 240, as they are reserved for use in critical error situations which can never be trapped with ON ERROR. Run-time error values are restricted to the range 1 through 255, and the compiler reserves codes 0 through 150, and 241 through 255 for predefined errors. Attempting to set an error value (with the ERROR statement) outside of the valid range 1 to 255 will result in a run-time [Error 5](#) ("Illegal function call") instead. In addition to the predefined run-time errors, you may also set your own customized run-time error codes in the range 151 through 240. These error codes may be useful to signal specific types of errors in your own applications, ready to be handled by your error trapping code.

In the situation where an undocumented run-time error occurs, the chief suspect is memory corruption. This can typically be caused by writing beyond an array boundary, improper use of pointers, bad Inline Assembly code, etc.

Handling of Run-Time Errors in Erlang

Error Handling Within Processes

It is possible to prevent run-time errors and other exceptions from causing the process to terminate by using catch or try, see the Expressions chapter about [Catch](#) and [Try](#).

Error Handling Between Processes

Processes can monitor other processes and detect process terminations, see the [Processes](#) chapter.

PROGRAMES OF UNIT - II

Dynamic Changes Style

| PROGRAM | OUTPUT |
|---|---|
| <p style="text-align: center;">Change Style of the Current DHTML Element</p> <pre><html> <body> <h1 onclick="this.style.color='red'">Click Me!</h1> </body> </html></pre> | <p style="text-align: center;">Click Me!</p> <p style="text-align: center;">Click Me! (Click on text to change color)</p> |
| <p style="text-align: center;">Change Style of a Specific CSS Element</p> <pre><html> <body> <h1 id="h1" onclick="document.getElementById('h1').style.color=' red'">Click Me!</h1> </body> </html></pre> | <p style="text-align: center;">Click Me!</p> <p style="text-align: center;">Click Me! (Click on text to change color)</p> |

Changing text:

| PROGRAM | OUTPUT |
|---|---|
| <pre><html> <body> <h1 id="header">Old Header</h1> <script type="text/javascript"> document.getElementById("header").innerHTML="N ew Header"; </script> <p>"Old Header" was changed to "New Header"</p> </body> </html></pre> | <p style="text-align: center;">New Header</p> <p style="text-align: center;">"Old Header" was changed to "New Header"</p> |
| <pre><html> <head> <script type="text/javascript"> function nameon() { document.getElementById('h2text').innerHTML="WE LCOME!"; } function nameout() {</pre> | <p style="text-align: center;">Mouse over this text!</p> <p style="text-align: center;">How are you today?</p> <p style="text-align: center;">WELCOME!</p> <p style="text-align: center;">(Click on text to change text)</p> |

| | |
|--|---|
| <pre>document.getElementById('h2text').innerHTML="How are you today?"; } </script> </head> <body> <h2 id="h2text" onmouseout="nameout()" onmouseover="nameon()"> Mouse over this text!</h2> </body> </html></pre> | |
| <pre><html> <head> <script type="text/javascript"> function blinking_header() { if (!document.getElementById('blink').style.color) { document.getElementById('blink').style.color= "red"; } if (document.getElementById('blink').style.color=="red") { document.getElementById('blink').style.color= "black"; } else { document.getElementById('blink').style.color= "red"; } timer=setTimeout("blinking_header()",100); } function stoptimer() { clearTimeout(timer); } </script> </head> <body onload="blinking_header()" onunload="stoptimer()"> <h1 id="blink">Blinking header</h1> </body> </html></pre> | <p style="text-align: center;">Blinking header</p> <p style="text-align: center;">Blinking header</p> <p>(Click on text to blink text in red color)</p> |

Creating Graphics and Multimedia Effects:

| PROGRAM | OUTPUT |
|--|--|
| <pre> <html> <head> <script type="text/javascript"> img2=new Image(); img2.src="landscape3.gif"; function changelImage() { document.getElementById('myImage').src=img2.src; } </script> </head> <body> <p>When you mouse over the image, a new image will appear.</p> <p>The new image appears instantly, because your browser has already loaded the image.</p> </body> </html> </pre> | <p>When you mouse over the image, a new image will appear.</p>  <p>The new image appears instantly, because your browser has already loaded the image</p> |
| <pre> <html> <head> <script type="text/javascript"> function cursor(event) { document.getElementById('trail').style.visibility="visible" ; document.getElementById('trail').style.position="absolute "; document.getElementById('trail').style.left=event.clientX +10; document.getElementById('trail').style.top=event.clientY; } </script> </head> <body onmousemove="cursor(event)"> <h1>Move the cursor over this page</h1> Cursor Text </body> </html> </pre> | <p>Move the cursor over this page</p> <p>(Move your cursor on output window to show cursor text)</p> |

```
<html>
<head>
<script>
function startEQ()
{
richter=5;
parent.moveBy(0,richter);
parent.moveBy(0,-richter);
parent.moveBy(richter,0);
parent.moveBy(-richter,0);
timer=setTimeout("startEQ()",10);
}
function stopEQ()
{
clearTimeout(timer);
}
</script>
</head>
<body>
<form>
<input type="button" onclick="startEQ()" value="Start
an earthquake">
<br />
<br />
<input type="button" onclick="stopEQ()" value="Stop
the earthquake">
</form>
</body>
</html>
```

Shake the window

B.COM – COMPUTER – III YEAR - NOTES
WEB PROGRAMMING

INDEX – UNIT – III AND IV

| S.no: | Title | Page no: |
|-------|--------------------------------------|----------------|
| 1. | VB Scripting Introduction | 02 - 02 |
| 2. | VB Script | 02 - 02 |
| 3. | Basics of VB Script | 03 - 07 |
| 4. | Array Handling | 08 - 10 |
| 5. | User Interaction in VB Script | 11 - 11 |
| 6. | Handling Runtime Errors. | 12 - 16 |
| 7. | PROGRAMES WITH OUTPUT: | 17 - 17 |

VB Scripting Introduction

What You Should Already Know

Before you continue you should have a basic understanding of the following:

- HTML / XHTML

If you want to study these subjects first, find the tutorials on our [Home page](#).

What is VBScript?

- VBScript is a scripting language
- A scripting language is a lightweight programming language
- VBScript is a light version of Microsoft's programming language Visual Basic
- **VBScript is only supported by Microsoft's browsers (Internet Explorer)**

How Does it Work?

When a VBScript is inserted into an HTML document, Internet Explorer browser will read the HTML and interpret the VBScript. The VBScript can be executed immediately, or at a later event.

VBScript only works in Microsoft browsers (Internet Explorer).

VB Script

VBScript (Visual Basic Scripting Edition) is an [Active Scripting](#) language developed by [Microsoft](#) that is modelled on [Visual Basic](#). It is designed as a "lightweight" language with a fast interpreter for use in a wide variety of Microsoft environments. VBScript uses the [Component Object Model](#) to access elements of the environment within which it is running; for example, the FileSystemObject (FSO) is used to [create, read, update and delete files](#).

VBScript has been installed by default in every desktop release of [Microsoft Windows](#) since [Windows 98](#); ^[1] in [Windows Server](#) since [Windows NT 4.0 Option Pack](#); ^[2] and optionally with [Windows CE](#) (depending on the device it is installed on).

A VBScript script must be executed within a host environment, of which there are several provided with Microsoft Windows, including: [Windows Script Host](#) (WSH), [Internet Explorer](#) (IE), and [Internet Information Services](#) (IIS). ^[3] Additionally, the VBScript hosting environment is embeddable in other programs, through technologies such as the Microsoft Script Control (`msscript.ocx`).

Basics of VB Script

VBScript Data Types

What Are VBScript Data Types?

VBScript has only one data type called a **Variant**. A **Variant** is a special kind of data type that can contain different kinds of information, depending on how it's used. Because **Variant** is the only data type in VBScript, it's also the data type returned by all functions in VBScript.

At its simplest, a **Variant** can contain either numeric or string information.

A **Variant** behaves as a number when you use it in a numeric context and as a string when you use it in a string context. That is, if you're working with data that looks like numbers, VBScript assumes that it is numbers and does the thing that is most appropriate for numbers. Similarly, if you're working with data that can only be string data, VBScript treats it as string data. Of course, you can always make numbers behave as strings by enclosing them in quotation marks (" ").

The following table shows the subtypes of data that a **Variant** can contain.

| Subtype | Description |
|--------------------|---|
| Empty | Variant is uninitialized. Value is 0 for numeric variables or a zero-length string ("") for string variables. |
| Null | Variant intentionally contains no valid data. |
| Boolean | Contains either True or False . |
| Byte | Contains integer in the range 0 to 255. |
| Integer | Contains integer in the range -32,768 to 32,767. |
| Currency | -922,337,203,685,477.5808 to 922,337,203,685,477.5807. |
| Long | Contains integer in the range -2,147,483,648 to 2,147,483,647. |
| Single | Contains a single-precision, floating-point number in the range -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values. |
| Double | Contains a double-precision, floating-point number in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values. |
| Date (Time) | Contains a number that represents a date between January 1, 100 to December 31, 9999. |
| String | Contains a variable-length string that can be up to approximately |

| | |
|---------------|---------------------------------|
| | 2 billion characters in length. |
| Object | Contains an object. |
| Error | Contains an error number. |

You can use [conversion functions](#) to convert data from one subtype to another. In addition, the [VarType](#) function returns information about how your data is stored within a **Variant**.

[VBScript Variables](#)

What Is a Variable?

A variable is a convenient placeholder that refers to a computer memory location where you can store program information that may change during the time your script is running. For example, you might create a variable called ClickCount to store the number of times a user clicks an object on a particular Web page. Where the variable is stored in computer memory is unimportant. What's important is that you only have to refer to a variable by name to see its value or to change its value. In VBScript, variables are always of one fundamental data type, [Variant](#).

[VBScript Constants](#)

What Is a Constant?

A [constant](#) is a meaningful name that takes the place of a number or string and never changes. VBScript defines a number of [intrinsic constants](#). You can get information about these intrinsic constants from the [VBScript Language Reference](#).

[VBScript Operators](#)

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. You can use parentheses to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, however, standard operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence.

| Arithmetic | | Comparison | | Logical | |
|-------------|--------|-------------|--------|-------------|--------|
| Description | Symbol | Description | Symbol | Description | Symbol |
| | | | | | |

| | | | | | |
|--------------------------------------|-----|--|----|-------------------------------------|-----|
| Exponentiation | ^ | Equality | = | Logical negation | Not |
| Unary negation | - | Inequality | <> | Logical conjunction | And |
| Multiplication | * | Less than | < | Logical disjunction | Or |
| Division | / | Greater than | > | Logical exclusion | Xor |
| Integer division | \ | Less than or equal to | <= | Logical equivalence | Eqv |
| Modulus arithmetic | Mod | Greater than or equal to | >= | Logical implication | Imp |
| Addition | + | Object equivalence | Is | | |
| Subtraction | - | | | | |
| String concatenation | & | | | | |

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation (&) operator is not an arithmetic operator, but in precedence it does fall after all arithmetic operators and before all comparison operators. The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

Using Conditional Statements

Controlling Program Execution

You can control the flow of your script with conditional statements and looping statements. Using conditional statements, you can write VBScript code that makes decisions and repeats actions. The following conditional statements are available in VBScript:

- [If...Then...Else](#) statement
- [Select Case](#) statement

Looping Through Code

Using Loops to Repeat Code

Looping allows you to run a group of statements repeatedly. Some loops repeat statements until a condition is **False**; others repeat statements until a condition is **True**. There are also loops that repeat statements a specific number of times.

The following looping statements are available in VBScript:

- **Do...Loop**: Loops while or until a condition is **True**.
- **While...Wend**: Loops while a condition is **True**.
- **For...Next**: Uses a counter to run statements a specified number of times.
- **For Each...Next**: Repeats a group of statements for each item in a collection or each element of an array.

VBScript Procedures

inds of Procedures

In VBScript there are two kinds of procedures; the **Sub** procedure and the **Function** procedure.

Sub Procedures

A **Sub** procedure is a series of VBScript statements, enclosed by **Sub** and **End Sub** statements, that perform actions but don't return a value. A **Sub** procedure can take arguments (constants, variables, or expressions that are passed by a calling procedure). If a **Sub** procedure has no arguments, its **Sub** statement must include an empty set of parentheses ().

The following **Sub** procedure uses two intrinsic, or built-in, VBScript functions, **MsgBox** and **InputBox**, to prompt a user for some information. It then displays the results of a calculation based on that information. The calculation is performed in a **Function** procedure created using VBScript. The **Function** procedure is shown after the following discussion.

```
Sub ConvertTemp()
    temp = InputBox("Please enter the temperature in degrees F.", 1)
    MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub
```

Function Procedures

A **Function** procedure is a series of VBScript statements enclosed by the **Function** and **End Function** statements. A **Function** procedure is similar to a **Sub** procedure, but can also return a value. A **Function** procedure can take arguments (constants, variables, or expressions that are passed to it by a calling procedure). If a **Function** procedure has no arguments, its **Function** statement must

include an empty set of parentheses. A **Function** returns a value by assigning a value to its name in one or more statements of the procedure. The return type of a **Function** is always a **VARIANT**.

In the following example, the Celsius function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the ConvertTemp **Sub** procedure, a variable containing the argument value is passed to the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub ConvertTemp()
    temp = InputBox("Please enter the temperature in degrees F.", 1)
    MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub
```

```
Function Celsius(fDegrees)
    Celsius = (fDegrees - 32) * 5 / 9
End Function
```

VBScript Coding Conventions

What Are Coding Conventions?

Coding conventions are suggestions that may help you write code using Microsoft Visual Basic Scripting Edition. Coding conventions can include the following:

- Naming conventions for objects, variables, and procedures
- Commenting conventions
- Text formatting and indenting guidelines

The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of a script or set of scripts so that you and others can easily read and understand the code. Using good coding conventions results in precise, readable, and unambiguous source code that is consistent with other language conventions and as intuitive as possible.

Array Handling

Arrays are a very important concept in VBScript as well as many other languages. An array is a special type of variable which stores a set of related values.

Imagine that you would like to store a list of all the gifts you would like to receive on your wedding day. You want to make a web page that displays a list of all the items. If you were to create a variable for each gift then you might end up having 100 or more variables for gifts alone! However, there is a better solution to this engineering problem.

Instead, you could utilize arrays, which allow you to store many variables(elements) into a super variable (array). Each present would have a position in the array, starting from position 0 and ending with the last gift

vbscript creating an array

We are going to dumb down the example a little bit so that this lesson doesn't get too boring. Let's imagine that we have 4 gifts we want to store in our array. First we need to create an array to store our presents and tell VBScript how big we want our array to be.

As we mentioned, an array's beginning position is 0, so if we specify an array of size 3 that means we can store 4 presents (positions 0, 1, 2 and 3)! This is often confusing for first time VBScript programmers. Below is the correct code to create a VBScript array of size 3.

VBScript Code:

```
<script type="text/vbscript">
Dim myArray(3)
</script>
```

vbscript arrays: storing data

Now that we have created our array we can begin storing information into it. The way to do this is similar to setting the value of a variable, but because an array can hold many values you have to specify the position at which you want the value to be saved.

We have four presents that we need to store and we make sure that we don't store two presents in the same position!

VBScript Code:

```
<script type="text/vbscript">
Dim myArray(3)
myArray(0) = "Clean Underwear"
myArray(1) = "Vacuum Cleaner"
```

```
myArray(2) = "New Computer"
myArray(3) = "Talking Bass"
</script>
```

[vbscript arrays: accessing data](#)

We have all the data stored into the array that we need, now we need to figure out how to get it back out so we can print it to the web page! This step is nearly identical to the storing phase because you have to specify the position of the element you wish to display. For example, if we wanted to print out the value of the present at position 0 in our array you would use the following code:

VBScript Code:

```
<script type="text/vbscript">
Dim myArray(3)
myArray(0) = "Clean Underwear"
myArray(1) = "Vacuum Cleaner"
myArray(2) = "New Computer"
myArray(3) = "Talking Bass"
document.write(myArray(0))
</script>
```

Display:

[vbscript arrays: accessing all data](#)

The above example was a good introduction to accessing elements in an array, but it isn't that helpful for printout out **all** items that might be in an array. If we had 300 items in our array, accessing them one by one would be most time consuming to program.

Below is a piece of code that will automatically go through every element in the array and print it out. The special programming structure this example uses is a For Loop that we will be discussing in greater detail later on in this tutorial.

VBScript Code:

```
<script type="text/vbscript">
Dim myArray(3)
myArray(0) = "Clean Underwear"
myArray(1) = "Vacuum Cleaner"
myArray(2) = "New Computer"
myArray(3) = "Talking Bass"
For Each present In myArray
document.write(present)
document.write("<br />")

```

Next
</script>

Modifying an arrays elements

To modify an arrays elements, refer to the array with the appropriate index number in brackets of the value you want to change and set it to the new value.

Example:

```
<script type="text/vbscript"> Dim colors(4) colors(0) = "green" colors(1) = "blue"
colors(2) = "gray" colors(3) = "orange" document.write("The old value of colors(1): " &
colors(1)) document.write("<br />The old value of colors(3): " & colors(3)) 'change the
value of colors(1) and colors(3) colors(1) = "teal" colors(3) = "violet"
document.write("<br />The new value of colors(1): " & colors(1)) document.write("<br
/>The new value of colors(3): " & colors(3)) </script>
```

Output:

The old value of colors(1): blue The old value of colors(3): orange The new value of colors(1): teal The new value of colors(3): violet

User Interaction in VB Script

The language of **VBScript** is modelled on Visual Basic, and therefore can be reviewed using similar categories: procedures, control structures, constants, variables, **user interaction**, array handling, date/time functions, error handling, mathematical functions, objects, regular expressions, string manipulation, and so on.^[14]

The following are some key points of introduction to the **VBScript** language.

A “procedure” is the main construct in **VBScript** for separating code into smaller modules. **VBScript** distinguishes between a function, which can return a result in an assignment statement, and a subroutine, which cannot. Parameters are positional, and can be passed by value or by reference.

Control structures include the usual iterative and conditional Do Loops, If-Then-Else statements, and Case statements, with some more complex variants, such as Elseif and nested control structures.

As a memory aid in coding, and certainly for readability, there are a large number of constants, such as True and False for logical values, vbOKCancel and vbYesNo for MsgBox codes, vbBlack and vbYellow for color values, vbCR for the carriage return character, and many others.

Variables by default have “Variant” type, but it is possible (and sometimes necessary) to force a particular type (integer, date, etc.) using conversion functions (Cint, CDate, etc.)

User interaction is provided through the functions MsgBox and InputBox which provide a simple dialogue box format for messages and input. Both functions display prompting messages, with the former returning a standard response, and the latter returning one **user**-supplied text or numeric value. For more elaborate GUI **interaction** with controls, **VBScript** can be used in combination with HTML, for example, in an [HTML Application](#). Event-driven forms are not supported as in Visual Basic or Visual Basic for Applications.

Names are not case-sensitive; therefore, for example, MsgBox and msgbox, or FileSystemObject and filesystemobject, are treated as the same name. However, as usual, it is considered a best practice of **VBScript** style to be consistent and to capitalize judiciously.

Handling Runtime Errors

Welcome to *Sesame Script*, the column for beginning script writers. The goal of this column is to teach the very basics of Windows scripting for system administration automation. We'll provide you with the information you'll need to begin reading and understanding scripts and to start modifying those scripts to suit your own needs. If there's anything in particular about scripting you're finding confusing, **let us know**; you're probably not alone.

[What's the Problem?](#)

[Troubleshooting vs. Error Handling](#)

[Error Types](#)

[Don't Stop Now](#)

[Not Complete Denial](#)

[Be Explicit](#)

[That's It?](#)

What's the Problem?

Have you ever worked on a team where one person always seems to be causing problems? If anything ever goes wrong, it can typically be linked to that person in some way. Sometimes it might not even be this person's fault (not entirely, anyway), but this particular person just always turns out to be the scapegoat. Believe it or not, the Scripting Guys have someone like that, too (but Dean doesn't like to talk about it, so we won't mention that it's him). Things happen, mistakes are made, and there's always one person to point a finger at.

The problem with writing scripts is that it's hard to blame anyone else when your script doesn't work the way you want it to. (Well, you *could* blame Dean; he may very well have had something to do with it.) But, most of the time, *you* wrote the script, or modified someone else's script, and now it just doesn't work. What do you do now? You could spend several days contemplating the unfairness of the world and cursing the day you ever heard of scripting, or you could simply do some troubleshooting.

Troubleshooting vs. Error Handling

We're going to talk about troubleshooting and error handling, so we should probably start by explaining what the difference is between the two. Simply put, troubleshooting is figuring out what went wrong; error handling is knowing what could go wrong and dealing with it ahead of time. We're not really going to distinguish between the two in this discussion, we're just going to get you started with a couple of simple techniques you can use to get out of trouble and stay out.

Error Types

There are three types of errors: syntax, runtime, and logic.

Syntax Errors

Syntax errors are usually the result of an incorrectly used keyword. Here are a couple of examples:

```
intNumber = 2
If intNumber = 2 Then
    Wscript.Echo intNumber
Edn If
```

```
intNumber = 2
If intNumber = 2
    Wscript.Echo intNumber
End If
```

In the first example, take a look at the closing End If statement. End is spelled Edn. VBScript doesn't know what Edn means, so it thinks the If statement doesn't have an End If and returns an error. In the second example we left the Then keyword off of the If Then statement. Again, VBScript is expecting a Then but doesn't see one, so it will return an error.

VBScript catches syntax errors before it tries to run the script and displays an error message if any of these errors are found. When you type **cscript test.vbs** at the command prompt, VBScript will look at the script and determine whether all the If's have matching Then's and that all other syntax issues are okay. Only if they are will it try to run the script.

Tip: Because syntax errors are checked before a script runs, a safe way to check for syntax errors is to put **Wscript.Quit** as the first line in your script, like this:

```
Wscript.Quit
intNumber = 2
If intNumber = 2
    Wscript.Quit intNumber
End If
```

You might have a script that is designed to delete files or change properties in Active Directory. Maybe you're still working on your script but you want to check and make sure your syntax is correct before you actually delete or change anything. Adding **Wscript.Quit** to the beginning of your script is an easy way to do this. When you try to run the script, it will be checked for syntax errors. If errors are found, they'll be output to the screen and the script will never run. If there aren't any syntax errors the script will run, but the very first line ends the script and no other lines are executed. You've just safely

tested the syntax of your script without actually running any of the script code.

Runtime Errors

Once you get through the syntax checks with no errors VBScript tries to run the script. At this point you might still get an error, but this time it's a runtime error, meaning the error occurs when the script is running. An example of a runtime error would be trying to connect to a computer that doesn't exist. VBScript has no way of knowing this ahead of time; the script needs to run and actually attempt to connect to the computer before it finds out there is no computer by that name (or it's offline, or whatever). At that point you'll receive an error message letting you know something went wrong.

Logic Errors

Logic errors are usually the most difficult errors to find. With a logic error, VBScript decides the script looks okay, the script runs and completes with no problems, but you don't get the results you expect. Here's an example of how that could happen:

```
intNumber = 2
If intNumber > 0 Then
    Wscript.Echo "Greater than zero"
Elseif intNumber > 1 Then
    Wscript.Echo "Greater than one"
Elseif intNumber > 2 Then
    Wscript.Echo "Greater than two"
End If
```

In this example, we've set intNumber equal to 2. When you go into the If statement, you might expect to receive the message "Greater than one." And when you look at the choices that might make sense: we check to see if intNumber is greater than 0, which it is. Then we check to see if it's greater than 1, and yes, 2 is greater than 1. Finally we check to see if it's greater than 2, which it's not. So our message should be "Greater than one," right? Nope. The way an If-Elseif statement works is that the first If statement will be checked: is intNumber greater than 0? If it is, the message "Greater than zero" will be echoed to the screen, and you'll go straight to the End If statement. The Elseif statements will never happen.

As you can see, there's nothing actually wrong with this script, it does exactly what it's supposed to; it just doesn't do exactly what you *want* it to. You have an error in the logic of your script, or the way the script works.

Don't Stop Now

Have you ever noticed how good people can be at denial? "I'm not going bald, I just have a high forehead." "No, that wasn't my child kicking the back of your seat and

talking all through the movie.” “There are no errors in my script, there’s something wrong with the computer.”

Not Complete Denial

You might not want to completely ignore errors. In the preceding script, you might not want the script to come to a crashing halt when it gets to an invalid date, but you also might not want to just skip right by it. Maybe you want to know that there is invalid information in your data set. But if you put On Error Resume Next in your script and ignore errors, how do you know they happened? Simple: just ask.

VBScript provides an object called the **Err** object. When an error occurs in a VBScript method that method returns an error number that you can access through the Number property of the Err object. So if the Number property is anything except zero, an error has occurred.

Be Explicit

This is an area we discussed in a [previous article](#), but we can’t discuss error handling without mentioning it: the **Option Explicit** statement. Let’s go back to our earlier example:

```
intNumber = 2
If intNumber = 2 Then
    Wscript.Echo intNubmer
End If
```

When you run this script you’re probably expecting it to echo back 2. What it will do instead is echo nothing. If you look closely, you can see that in the Wscript.Echo statement, our variable is spelled incorrectly, inNubmer rather than intNumber. This doesn’t produce an error. VBScript simply thinks you’re using a different variable, which you never actually gave a value so it defaulted to nothing. This type of error can be difficult to spot (even removing an On Error Resume Next statement won’t help you) because there’s no error for VBScript to report. One way to guard against this type of error is to include the Option Explicit statement:

```
Option Explicit

intNumber = 2
If intNumber = 2 Then
    Wscript.Echo intNubmer
End If
```

Now when you run the script you’ll receive an error message:

```
C:\Scripts\test.vbs(3, 1) Microsoft VBScript runtime error: Variable is undefined:
'intNumber'
```

Wait a minute, intNumber isn't the problem, the variable that isn't spelled correctly is intNubmer. Why am I getting an error on the variable I want? Because Option Explicit tells VBScript to display an error for any variable that hasn't been explicitly declared (get it, explicitly, Option Explicit?) with a Dim statement. That means every single variable you use in your script, including object references, must be declared before you can use it, like this:

```
Option Explicit
```

```
Dim intNumber
```

```
intNumber = 2
```

```
If intNumber = 2 Then
```

```
    Wscript.Echo intNubmer
```

```
End If
```

Now when you run your script you get an error message that will actually help you:

```
C:\Scripts\test.vbs(7, 5) Microsoft VBScript runtime error: Variable is undefined:
'intNubmer'
```

As with all good remedies there are side-effects to Option Explicit. As we already saw, we might get errors on variables that are perfectly fine because we forgot to declare them with a Dim statement. If you have a lot of variables in your script this can get very cumbersome. Some people go too far the other direction and leave in Dim statements for variables they don't even use. This, too can clutter up your script and make it hard to read, so try to clean up after yourself. There's nothing like a nice, neat script to get you through the day.

That's It?

There is much more to error handling than the little bit we've shown you here in this article. But we're tired of working now, so this is all you get.

(Oops, our manager just walked by and saw that.) No, really, just kidding, we *never* get tired of working. And we'd *never* leave our devoted readers without making sure we'd provided as much information as we possibly could. But what we *have* done here is give you a quick overview of error-handling, and shown you just a few types of errors you might encounter. For more in-depth information and some intermediate-to-advanced error-handling techniques, take a look at [To Err is VBScript](#). And of course, there's always the handy [Windows 2000 Scripting Guide](#), loaded with information.

PROGRAMES OF UNIT - II

| Program | Output |
|---|----------------------------|
| <pre><html> <body> <script type="text/vbscript"> document.write("This is my first VBScript!") </script> </body> </html></pre> | This is my first VBScript! |
| <pre><html> <body> <script type="text/vbscript"> document.write("Hello World!") </script> </body> </html></pre> | Hello World! |
| <p style="text-align: center;">VBScript Array</p> <pre><html> <body> <script type="text/vbscript"> a=Array(5,10,15,20) document.write(a(3)) </script> </body> </html></pre> | 20 |
| <pre><html> <body> <script type="text/vbscript"> a=Array(5,10,15,20) document.write(a(0)) </script> </body> </html></pre> | 5 |
| <pre><html> <body> <script type="text/vbscript"> a=Array(5,10,15,20) for each x in a document.write(x&" ") next </script> </body> </html></pre> | 5 10 15 20 |

KNOWLEDGE IS POWER

B.COM – COMPUTER – III YEAR – NOTES
WEB PROGRAMMING

INDEX – UNIT - V

| S.no: | Title | Page no: |
|--------------|---|-----------------|
| 1. | Extensible Markup Language (XML) – Introduction | 02 - 04 |
| 2. | Creating XML Documents | 05 - 05 |
| 3. | XML Style Sheets | 06 - 07 |
| 4. | Comparison of XML with other Web Designing Languages | 08 - 09 |
| 5. | XML Document Object Model | 10 - 11 |
| 6. | XML Query Language | 12 - 14 |
| 7. | XML DTD | 15 - 17 |

Comparison of XML with other Web Designing Languages

Comparing XML with HTML

1. XML is designed to carry the data while html is to display the data and focus on how the data looks.
2. XML does not do anything but it is used to structure the data, store the data and transport the data while HTML is used to display the data only.
3. XML is cross platform, hardware and software independent tool to carry data from one source to another destination.
4. XML is self descriptive. The DTD or schema describes what and how to use tags and elements in an xml document.
5. XML does not have predefined tags while HTML has. XML lets you invent your own tags while html gives you predefined tags and you have to use them only.
6. XML is extensible as you can add your own tags to extend the xml data.
7. XML is a complement to HTML not the replacement. Most likely use of xml is to transfer the data between incompatible systems as xml is cross platform. XML has now established a strong foundation in the world of web development.
8. XML can separate the data from html. While working on html, instead of storing data in html you can store the data in a separate xml file to make the html code cleaner and easier. Now you can concentrate on working in html rather than storing data. It also eliminates the need to change in html when xml data is changed.
9. XML tags are case sensitive while html tags are not.
10. Attribute values must always be quoted in xml while its not the case with html.
11. XML elements must be properly nested while html is not so sensitive.

| XML | HTML | CSS |
|--|--|--|
| <p>A set of rules for encoding documents in machine-readable form. The design goals of XML emphasize simplicity, generality, and usability over the Internet.^[6] It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services.</p> | <p>The predominant markup language for web pages. HTML is the basic building-blocks of webpages. The purpose of a web browser is to read HTML documents and compose them into visual or audible web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page. HTML elements form the building blocks of all websites. HTML allows images and objects to be embedded and can be used to create interactive forms. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items.</p> | <p>A style sheet language used to describe the presentation semantics (the look and formatting) of a document written in a markup language. Its most common application is to style web pages written in HTML and XHTML, but the language can also be applied to any kind of XML document, including plain XML, SVG and XUL.</p> |

XML Document Type Definition (DTD):

The oldest schema language for XML is the [Document Type Definition](#) (DTD), inherited from [SGML](#).

DTDs have the following benefits:

- DTD support is ubiquitous due to its inclusion in the XML 1.0 standard.
- DTDs are terse compared to element-based schema languages and consequently present more information in a single screen.
- DTDs allow the declaration of [standard public entity sets](#) for publishing characters.
- DTDs define a *document type* rather than the types used by a namespace, thus grouping all constraints for a document in a single collection.

DTDs have the following limitations:

- They have no explicit support for newer [features](#) of XML, most importantly [namespaces](#).
- They lack expressiveness. XML DTDs are simpler than SGML DTDs and there are certain structures that cannot be expressed with regular grammars. DTDs only support rudimentary datatypes.
- They lack readability. DTD designers typically make heavy use of parameter entities (which behave essentially as textual [macros](#)), which make it easier to define complex grammars, but at the expense of clarity.
- They use a syntax based on [regular expression](#) syntax, inherited from [SGML](#), to describe the schema. Typical XML APIs such as [SAX](#) do not attempt to offer applications a structured representation of the syntax, so it is less accessible to programmers than an element-based syntax may be.

Two peculiar features that distinguish DTDs from other schema types are the syntactic support for embedding a DTD within XML documents and for defining *entities*, which are arbitrary fragments of text and/or markup that the XML processor inserts in the DTD itself and in the XML document wherever they are referenced, like character escapes.

DTD technology is still used in many applications because of its ubiquity.

The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. A DTD can be declared inline in your XML document, or as an external reference.

Why use a DTD?

XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive **from the outside world is valid. You can also use a DTD to verify your own data.**

Valid XML Documents

A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DOCTYPE declaration in the example above, is a reference to an external DTD file. The content of the file is shown in the paragraph below.

XML DTD

The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

If you want to study DTD, you will find our DTD tutorial on our [homepage](#).

The purpose of a DTD (Document Type Definition) is to define the legal building blocks of an XML document.

A DTD defines the document structure with a list of legal elements and attributes.

DTD Newspaper Example

```
<!DOCTYPE NEWSPAPER [  
  
<!ELEMENT NEWSPAPER (ARTICLE+)>  
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>  
<!ELEMENT HEADLINE (#PCDATA)>  
<!ELEMENT BYLINE (#PCDATA)>  
<!ELEMENT LEAD (#PCDATA)>  
<!ELEMENT BODY (#PCDATA)>  
<!ELEMENT NOTES (#PCDATA)>  
  
<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>  
<!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>  
<!ATTLIST ARTICLE DATE CDATA #IMPLIED>  
<!ATTLIST ARTICLE EDITION CDATA #IMPLIED>  
  

```