

Dr. UMayal RamAnathan College for Women, Karaikudi-03

Accredited with B+ Grade by NAAC

Affiliated to Alagappa University

(Run by Dr. Alagappa Chettiar Educational Trust)

Karaikudi – 630 003

Study Material

2020 – 2021

7BCEE1B

Web Design

Name of the Staff : Ms. R. Valaiyapathy

Department : Computer Science

Course Outcome:

Web Design	7BCEE1B	CO1	To provide the learners to understanding the basic concepts and terminology of HTML programming in general.
		CO2	Ability to understand how to use CSS in webpage, how to add the elements are discussed.
		CO3	Formats based on the requirements of the problem.
		CO4	To learn about the user defined functions

**III YEAR – V SEMESTER
COURSE CODE: 7BCEE1B
ELECTIVE COURSE-I (B)–WEB DESIGN**

Unit I

Introduction to HTML: Markup Languages – editing HTML – common tags – header – text styling – linking – images – formatting text – special characters, horizontal rules and line breaks – unordered list – nested and ordered list – tables and formatting – forms – linking – frames.

Unit II

Cascading Style Sheets:

Introduction – Inline styles – Embedded Style Sheets – Conflicting Style – Linking External Style Sheets – Positioning Elements – Backgrounds – Element Dimension – Box Model and Text Flow – Media Types – Building a Dropdown menu

Unit III

Java Script: introduction – control structures – if structure – while structure – assignment

operators – increment and decrement operators – for structure – switch structure – do/while structure – break and continue statement – logical operators

Unit IV

Java Script Functions: Programmer defined functions – function definitions – duration of identifiers – scope rules – recursion – recursion vs iteration – global functions

Java Script Arrays: Arrays – declaring and allocating arrays – references and reference parameters – passing arrays to functions – sorting arrays – searching arrays – multiple-subscripted arrays

Java Script Objects: Math object – String object – Date object – Boolean and Number Object – document object – window object.

Unit V

Document Object Model (DOM): Modeling a document – Traversing and modifying a DOM

Tree – DOM collections and Dynamic styles.

Events: Registering Event Handlers – onload - onmousemove, the event Object and this – on mouseover and on mouseout – onfocus and onblur – form processing with onsubmit and onreset – event bubbling and other events.

XML: Basics – structuring Data – XML Name Spaces – Document Type Definations – W3C XML schema documents – XML Vocabularies – XSLT

Text Book:

1. -Internet and World Wide Web – How to Program, H.M.Deitel, P.J.Deital, T.R.Nieto, Pearson Education Asia – Addison Wesley Longman Pte Ltd.

Book for Reference:

1. -Special edition using HTML, Mark R Brown and Jerry Honeycutt, Third edition



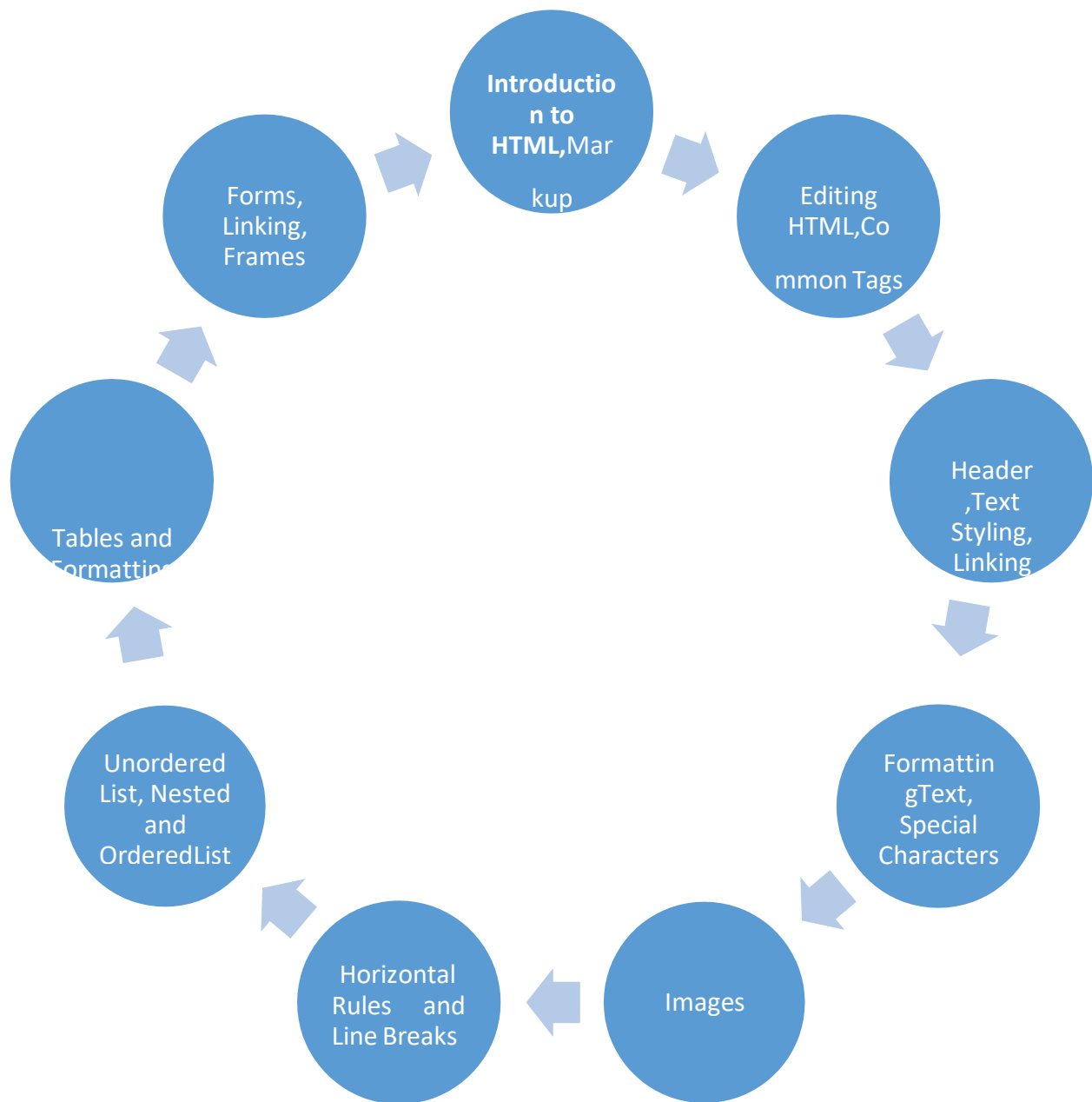
Course Outcome:

Semester	Course Title	Course Code	Course Outcome
----------	--------------	-------------	----------------

V	Web Design	7BCEE1B	CO1	Understand the principles of creating an effective web page, including an in-depth consideration of information architecture. Become familiar with graphic design principles that relate to web design and learn how to implement theories into practice.
			CO2	Learn the language of the web: HTML and CSS. Learn CSS grid layout and flex box.
			CO3	Learn techniques of responsive web design, including media queries. Develop skills in digital imaging (Adobe Photoshop.)
			CO4	Develop basic programming skills using Java script. Be able to embed social media content into web pages.
			CO5	Develop skills in analyzing the usability of a web site. Understand how to plan and conduct user research related to web usability.

Unit I

Mind Map



Introduction to HTML:

This language provides the format for specifying simple logical structure and links in a hypertext document. As a markup language, special formatting commands are placed in the text describing how the final version should appear. These formatted documents are interpreted by a Web browser which uses the HTML code to format the page being displayed. Although most professionals use special authoring tools to write HTML documents and to manage sites, developers of e-commerce sites and applications need to know the nitty-gritty detail of HTML, and this is what you will study.

HTML has had several versions over the years. "HTML 2.0" was the first standard HTML specification which was published in 1995. HTML 4.01 was a major version of HTML and it was published in late 1999. Though HTML 4.01 version is widely used but currently we are having HTML 5

version which is an extension to HTML 4.01, and this version was published in 2012. This course will take you through website creation using HTML5.

Markup Languages

HTML pages are created by tagging textual information with HTML markup. HTML markup consists of tags, which appear inside angled brackets < and >

An example of an HTML tag is , which causes text to appear in bold. only notes where text should begin to appear in bold, while the tag marks the end of the emboldening. Most HTML tags have a corresponding end tag, which is specified by the name of the tag preceded by the / character. So, to create the text:

Internet Commerce is great!

The text is marked up as:

```
<B>Internet Commerce is great!</B>
```

Another example of an HTML tag is <I>, which causes text to appear in italic. In HTML 4.01, the <I> tag was used to render text in italics. However, this is not necessarily the case with HTML5. Style sheets can be used to format the text inside the <I> element. This will be demonstrated later.

Note that tags are not case-sensitive. In other words, or are the same tag, both specifying bold text.

Nesting HTML Tags

Text may be both bold and italicised. This is done by using both the and <I> tags. When doing so, it is important to remember not to overlap HTML tags. In other words:

```
<B><I>Internet Commerce is great!</I></B>
```

is **correct**, but

```
<B><I>Internet Commerce is great!</B></I>
```

is **wrong**.

Overlapping tags is a common mistake. Although Web browsers are usually smart enough to work out what is meant, it can lead to problems. Furthermore, for an HTML page to be considered valid HTML, it must contain no overlapping tags.

To Do:

Read the section on "HTML Tags" in your textbooks.

Editing Html

This section covers the creation of an HTML page. You will need a Web browser and a text editor. Use any text editor you wish to, but the following Activity descriptions will use Notepad++. Notepad++ is a free Windows editor that also supports several programming languages. For example, you will notice that HTML keywords are highlighted in different colors.

1. Open your Web browser. This sections' goal is to create a Web document that can be opened with your browser.
2. Open Notepad++. It can be found by selecting Start, then All Programs, then Notepad++.
3. Type the following text into Notepad++: your name and the module number (CSC5003). Save this file as start.txt.
4. Now load start.txt into the browser by dragging start.txt onto your browser.
5. The browser should now display the text contained in **start.txt**. (If it does not, make sure that you have saved **start.txt** and that this is the file you are opening).
6. Once you have displayed start.txt, return to Notepad. Add the text "Internet Commerce", and save the file again.
7. Return to the Web browser and reload the document (by using either by using the Refresh or Reload toolbar buttons, or by selecting File/Open once again).
8. If you are able to see the new piece of text, you have successfully used Notepad to create your first Web page.

Getting started with HTML

This Activity adds HTML tags to start.txt.

1. Open your file **start.txt** in Notepad.
2. Mark up the text "Internet Commerce" so that appears in bold. Do this by placing the tag in front of the text, and at the end of the text, as shown below:
Internet Commerce
3. Save the file as **start.html**, since it contains some HTML formatting. Save the file with this new name (using Save As). Note that saving it as **start.htm** is also accepted. Other than the obvious, the letter "L," there's not much of a difference between the two extensions. Most, if not all, web browsers and servers will treat a file with an HTM extension exactly as it would a file with an HTML extension, and vice versa.
4. Load **start.html** in the Web browser. **Internet Commerce** should now appear in bold.
5. Return to Notepad and add more text, some of it in bold and others in italics. (Remember <I> is the tag for italics) Save the document and reload it.

Common Tags

Although a number of HTML tags have been introduced that markup how text should be displayed in a browser, a correct HTML document must always include certain structural tags. These tags are<HTML>, <HEAD>, <BODY> and <TITLE>.

The <HTML> tag should be placed around the document's contents; this tells the browser that the whole document is written in HTML. Like a person, all HTML documents have only one head and

one body. All the text of the HTML document should be inside either the head or the body. Roughly, the <HEAD> holds information about the document itself, and the <BODY> holds the information that should be displayed. The document's <TITLE> is given in the <HEAD>. The title is shown at the very top of the browser (i.e. in the title bar) —not in the browser window itself.

The standard structure of an HTML document is:

```
<HTML>
<HEAD>
<TITLE>Text to appear in the title bar of the browser</TITLE>
</HEAD>
<BODY>The text to appear in the main browser window.
</BODY>
</HTML>
```

This format should always be used when writing HTML documents.

Note: students are often confused about the use of the <BODY> tag, and they often include multiple body tags. This can lead to problems later on, so make sure to use only one <BODY> tag.

To Do: Read the section on HTML document structure in your textbooks.

Structuring your HTML document

In this Activity you will convert your file that contains a few HTML tags into a correctly structured HTML document. Open start.htm in Notepad.

1. Add the <HTML> tag on the first line of the file (before anything else).
2. Add the </HTML> end tag on the last line of the file (after everything else).
3. Add the document header by adding a <HEAD> tag on the line underneath the <HTML> tag and the </HEAD> tag on the line beneath that.
4. Between the opening and closing <HEAD> tags, add the <TITLE> and </TITLE> tags.
5. Enter the text "My first Web page" between the <TITLE> tags.
6. Underneath the </HEAD> tag, create the body of the document by entering the <BODY> tag.
7. At the bottom of the document, add the </BODY> tag just before the </HTML> tag.
8. Save the file.

If you have problems correctly formatting the file.

You are probably thinking that it looks the same as the previous document. However, if you look closely at the title bar you should see that it now displays the words "My first Web page". The main difference, however, is that the browser now has to do a lot less work to do, since the document informs it of the HTML's structure.

Loading your HTML file on Tomcat

The previous chapter guided you through tomcat installation. Let us launch the start.html file using the tomcat webserver. Make sure that your tomcat server has been started. Save start.html in the folder my apps that you created within the web apps folder. Load **start.html** in your browser by typing <http://localhost:8888/myapps/start.html>

Header

The DOCTYPE declaration defines the document type to be HTML. In HTML5 this is written as `<!DOCTYPE html>`. The `<!DOCTYPE>` declaration helps the browser to display a web page correctly. There are different document types on the web. To display a document correctly, the browser must know both type and version. The doctype declaration is not case sensitive. All cases are acceptable:

Another set of HTML tags are the headings tags. These are `<H1>`, `<H2>`, `<H3>`, `<H4>`, `<H5>` and `<H6>`. The text surrounded by the `<H1>` tag is displayed in a very large font size. Text surrounded by the `<H2>` tag is displayed in a slightly smaller font size, and so on down to the `<H6>` heading tag. You can use these tags to provide your page with a standard outline format. For example, the page heading might be displayed using the `<H1>` tag, a section heading using `<H2>` and a sub-section heading using `<H3>` and so on. Use HTML headings for headings only. Don't use headings to make text **BIG** or **bold**.

Search engines use your headings to index the structure and content of your web pages. It is important to use headings to show the document structure. Browsers automatically add some empty space (a margin) before and after each heading.

Headings

1. Load format.htm in MS-Notepad.
2. Within the `<head>` tags, add `<meta charset="UTF-8">`. It does not matter whether it is below or after the `<title>` tag.
3. Set up the page heading "Formatting text" and place the `<H1>` heading tags around it, in other words, `<H1>Formatting text</H1>`.
4. Reload format.html in your browser. You will notice that the effect of the `<H1>` tag is to display the text not only in an enlarged font size but also to include extra space above and below it. So you do not need a `
` or `<P>` tag as well.
5. Return to Notepad and use the `<H2>` tag to create a sub-heading for the page, "Paragraphs and line breaks".
6. Add `<hr>` between `_This'` and `_is'`.
7. Reload the document in your browser to check the HTML and you should have an output.

Text Styling

The HTML **style** attribute is used to add styles to an element, such as color, font, size, and more.

Example I

am Red I

am Blue

I am Rio

The HTML Style Attribute

Setting the style of an HTML element, can be done with the **style** attribute.

The HTML **style** attribute has the following syntax:

```
<tagname style="property:value;">
```

You will learn more about CSS later in this tutorial.

The **property** is a CSS property. The **value** is a CSS value.

Background Color

The CSS **background-color** property defines the background color for an HTML element.

Example

Set the background color for a page to powderblue:

```
<body style="background-color:powderblue;">
```

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

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

```
</body>
```

Example

Set background color for two different elements:

```
<body>
```

```
<h1 style="background-color:powderblue;">This is a heading</h1>
```

```
<p style="background-color:tomato;">This is a paragraph.</p>
```

```
</body>
```

Text Color

The CSS **color** property defines the text color for an HTML element:

Example

```
<h1 style="color:blue;">This is a heading</h1>
```

```
<p style="color:red;">This is a paragraph.</p>
```

Fonts

The CSS **font-family** property defines the font to be used for an HTML element:

Example

```
<h1 style="font-family:verdana;">This is a heading</h1>
```

```
<p style="font-family:courier;">This is a paragraph.</p>
```

Text Size

The CSS **font-size** property defines the text size for an HTML element:

Example

```
<h1 style="font-size:300%;">This is a heading</h1>  
<p style="font-size:160%;">This is a paragraph.</p>
```

Text Alignment

The CSS `text-align` property defines the horizontal text alignment for an HTML element:

Example

```
<h1 style="text-align:center;">Centered Heading</h1>  
<p style="text-align:center;">Centered paragraph.</p>
```

Linking

The `<link>` tag defines the relationship between the current document and an external resource. The `<link>` tag is most often used to link to external style sheets or to add a [favicon](#) to your website.

The `<link>` element is an empty element, it contains attributes only.

Example

Link to an external style sheet:

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

Attributes

Attribute	Value	Description
<code>crossorigin</code>	anonymous use-credentials	Specifies how the element handles cross-origin requests
<code>href</code>	<i>URL</i>	Specifies the location of the linked document
<code>hreflang</code>	<i>language_code</i>	Specifies the language of the text in the linked document
<code>media</code>	<i>media_query</i>	Specifies on what device the linked document will be displayed
<code>referrerpolicy</code>	no-referrer no-referrer-when-downgrade origin origin-when-cross-origin unsafe-url	Specifies which referrer to use when fetching the resource

rel	alternate author dns-prefetch help icon license next pingback preconnect prefetch preload prerender prev search stylesheet	Required. Specifies the relationship between the current document and the linked document
sizes	<i>HeightxWidth</i> any	Specifies the size of the linked resource. Only for rel="icon"
title		Defines a preferred or an alternate stylesheet
type	<i>media_type</i>	Specifies the media type of the linked document

Global Attributes

The `<link>` tag also supports the [Global Attributes in HTML](#).

Event Attributes

The `<link>` tag also supports the [Event Attributes in HTML](#).

```
link {
  display: none;
}
```

Images

The `` tag is used to embed an image in an HTML page.

Images are not technically inserted into a web page; images are linked to web pages.

The `` tag creates a holding space for the referenced image.

The `` tag has two required attributes:

- src - Specifies the path to the image
- alt - Specifies an alternate text for the image, if the image for some reason cannot be displayed

Note: Also, always specify the width and height of an image. If width and height are not specified, the page might flicker while the image loads.

Tip: To link an image to another document, simply nest the `` tag inside an `<a>` tag (see example below).

Example

How to insert an image:

```

```

Attributes

Attribute	Value	Description
alt	<i>text</i>	Specifies an alternate text for an image
crossorigin	anonymous use-credentials	Allow images from third-party sites that allow cross-origin access to be used with canvas
height	<i>pixels</i>	Specifies the height of an image
ismap	ismap	Specifies an image as a server-side image map
loading	eager lazy	Specifies whether a browser should load an image immediately or to defer loading of images until some conditions are met
longdesc	<i>URL</i>	Specifies a URL to a detailed description of an image
referrerpolicy	no-referrer no-referrer-when-downgrade origin origin-when-cross-origin unsafe-url	Specifies which referrer information to use when fetching an image
sizes	<i>sizes</i>	Specifies image sizes for different page layouts
src	<i>URL</i>	Specifies the path to the image
srcset	<i>URL-list</i>	Specifies a list of image files to use in different situations
usemap	<i>#mapname</i>	Specifies an image as a client-side image map
width	<i>pixels</i>	Specifies the width of an image

Global Attributes

The `` tag also supports the [Global Attributes in HTML](#).

Event Attributes

The `` tag also supports the [Event Attributes in HTML](#).

Example Global Attributes

The `` tag also supports the [Global Attributes in HTML](#).

Event Attributes

The `` tag also supports the [Event Attributes in HTML](#).

Formatting Text

HTML contains several elements for defining text with a special meaning.

This text is bold

This is _{subscript} and ^{superscript}

HTML Formatting Elements

Formatting elements were designed to display special types of text:

- `` - Bold text
- `` - Important text
- `<i>` - Italic text
- `` - Emphasized text
- `<mark>` - Marked text
- `<small>` - Smaller text
- `` - Deleted text
- `<ins>` - Inserted text
- `<sub>` - Subscript text
- `<sup>` - Superscript text

HTML `` and `` Elements

The HTML `` element defines bold text, without any extra importance.

Example

```
<b>This text is bold</b>
```

The HTML `` element defines text with strong importance. The content inside is typically displayed in bold.

Example

```
<strong>This text is important!</strong>
```

HTML `<i>` and `` Elements

The HTML `<i>` element defines a part of text in an alternate voice or mood. The content inside is typically displayed in italic.

Tip: The `<i>` tag is often used to indicate a technical term, a phrase from another language, a thought, a ship name, etc.

Example

```
<i>This text is italic</i>
```

The HTML `` element defines emphasized text. The content inside is typically displayed in italic.

Tip: A screen reader will pronounce the words in `` with an emphasis, using verbal stress.

Example

```
<em>This text is emphasized</em>
```

HTML `<small>` Element

The HTML `<small>` element defines smaller text:

Example

```
<small>This is some smaller text.</small>
```

HTML `<mark>` Element

The HTML `<mark>` element defines text that should be marked or highlighted:

Example

```
<p>Do not forget to buy <mark>milk</mark> today.</p>
```

HTML `` Element

The HTML `` element defines text that has been deleted from a document. Browsers will usually strike a line through deleted text:

Example

```
<p>My favorite color is <del>blue</del> red.</p>
```

HTML `<ins>` Element

The HTML `<ins>` element defines a text that has been inserted into a document. Browsers will usually underline inserted text:

Example

<p>My favorite color is blue <ins>red</ins>.</p>

HTML <sub> Element

HTML <sub> element defines subscript text. Subscript text appears half a character below the normal line, and is sometimes rendered in a smaller font. Subscript text can be used for chemical formulas, like H₂O:

Example

<p>This is _{subscripted} text.</p>

HTML <sup> Element

The HTML <sup> element defines superscript text. Superscript text appears half a character above the normal line, and is sometimes rendered in a smaller font. Superscript text can be used for footnotes, like WWW^[1]:

Example

<p>This is ^{superscripted} text.</p>

HTML Text Formatting Elements

Tag	Description
	Defines emphasized text
<small>	Defines smaller text
<sub>	Defines subscripted text
<sup>	Defines superscripted text

<u><ins></u>	Defines inserted text
<u></u>	Defines deleted text
<u><mark></u>	Defines marked/highlighted text

Special Characters,

Some characters are reserved in HTML and they have special meaning when used in HTML document. For example, you cannot use the greater than and less than signs or angle brackets within your HTML text because the browser will treat them differently and will try to draw a meaning related to HTML tag.

Many mathematical, technical, and currency symbols, are not present on a normal keyboard.

To add such symbols to an HTML page, you can use the entity name or the entity number (a decimal or a hexadecimal reference) for the symbol.

Example

Display the euro sign, €, with an entity name, a decimal, and a hexadecimal value:

`<p>I will display €</p>`

`<p>I will display €</p>`

`<p>I will display €</p>`

Will display as:

I will display €

I will display €

I will display €

HTML processors must support following five special characters listed in the table that follows.

Symbol	Description	Entity Name	Number Code
"	quotation mark	"	"
'	apostrophe	'	'
&	ampersand	&	&
<	less-than	<	<

Symbol	Description	Entity Name	Number Code
>	greater-than	>	>

Horizontal Rules

The `<hr>` tag creates a horizontal line in an HTML page. The `<hr>` element can be used to separate content.

The HTML `<meta>` element is also meta data. It can be used to define the character set, and other information about the HTML document. Other meta elements that can be used are `<style>` and `<link>`.

Line Breaks

The tag `
` is used to start a new line. `
` is a standalone tag, that means there is no closing `</BR>` tag. Note that `
` does not place a line space between the two lines. To do that you need to use the `<P>` paragraph tag. Do not forget to add the end tag `</p>` although most browsers will display HTML correctly even if you forget the end tag. The tag `<pre>` defines preformatted text. The text inside a `<pre>` element is displayed in a fixed-width font (usually Courier), and it preserves both spaces and line breaks.

HTML Tip - How to View HTML Source

Have you ever seen a Web page and wondered "Hey! How did they do that?" To find out, right-click in the page and select "View Page Source" (in Chrome) or "View Source" (in IE), or similar in another browser. This will open a window containing the HTML code of the page.

Formatting text

Paragraphs and line breaks

Users of HTML are sometimes surprised to find that HTML gives them little control over the way that a page is displayed. It should be remembered that HTML was developed as a means of marking up the structure of a document not as a way of determining its presentation.

Formatting text to appear on a Web page is therefore different from formatting text to appear in a printed document.

This

is

cool.

Using headings, horizontal rules and meta tags

Paragraphs, Line Breaks and Preformatting

In this Activity you will use the <P> and
 tags to create line breaks in text. We will also demonstrate the use of <pre>.

1. Load Notepad and begin a new HTML document.
2. Enter the usual structural HTML tags. Set the title to "Formatting text".
3. Within the body type in the following text exactly as it appears below. Note how 'This is cool' has been typed. Do not use any HTML tags to format it at this stage.

Users of HTML are sometimes surprised to find that HTML gives them little control over the way that a page is displayed. It should be remembered that HTML was developed as a means of marking up the structure of a document not as a way of determining its presentation. Formatting text to appear on a Web page is therefore different from formatting text to appear in a printed document.

This

is

Cool.

4. Save the document as **format.html** in your myapps folder and load it in your browser to view it. Note that 'This is cool' is displayed without the line breaks.
5. Resize your browser and watch how the text is reformatted to fit in the resized browser window.
6. Return to Notepad and make the changes as shown in Figure 3.1.
7. Save the file again and load it in your browser to check your HTML. Resize the browser and watch how the document is reformatted for the resized window.

Lists

In this Activity you will create a series of lists to practice your HTML list-building skills.

1. Load format.html in Notepad.
2. Underneath the text, create three lists as follows:
 - a. List one should be a circled bulleted (i.e. unordered) list, using square bullets, giving the days of the week.
 - b. List two should be a numbered list of the months of the year. Make the numbers lowercase roman numerals.
 - c. List three should be a definition list of the four seasons.
3. Save the file and view it in your Web browser to ensure that it displays as desired.
4. Reload format.html in Notepad and create a new bulleted list showing the four seasons. Within each season create a numbered sub list of the appropriate months of the year.
5. Save the file and load it in your Web browser to examine the document.

Unordered List

- Apple 1. Apples
- Oranges 2. Oranges
- Bananas 3. Bananas

The two examples above are lists. The list on the left uses bullets to mark the list elements, and is known as an **unordered list**. The list on the right uses numbers to mark the list elements and is known as an **ordered list**.

HTML lists consist of a list tag and list element tags.

In an **unordered list**, the list tag is and the list element tag is . Note that although the list element end tag was optional in previous versions of HTML, it no longer is. The list end tag is also not optional.

To create an **unordered list** as in the above example, use the following HTML.

```
<UL>
<LI>Apples</LI>
<LI>Oranges</LI>
<LI>Bananas</LI>
</UL>
```

Note that it is useful to indent the tags on the page to keep track of the level of indentation. To add more list elements, add extra list element tags containing the elements within the tags.

Nested and Ordered List

A **style** attribute can be added to an unordered list, to define the style of the marker:

Type	Description
type="1"	The list items will be numbered with numbers (default)
type="A"	The list items will be numbered with uppercase letters
type="a"	The list items will be numbered with lowercase letters
type="I"	The list items will be numbered with uppercase roman numbers
type="i"	The list items will be numbered with lowercase roman numbers

For example,

```
<OL type = "i">
```

```
<LI>Apples</LI>
<LI>Oranges</LI>
<LI>Bananas</LI>
</OL>
```

The **description** list, is different: it has neither bullets nor numbers. The description list tag is `<DL>` and the list elements consist of a term and its definition. The term is marked by `<DT>` tags and the definition by `<DD>` tags. An example use definition lists is the glossary definition that appears below.

```
<DL>
<DT>HTML </DT>
<DD> Hypertext Markup Language; the format of Web documents </DD>
</DL>
```

Lists can be nested (lists inside lists). For example,

```
<OL type = "i">
<LI>Apples</LI>
<LI>Oranges</LI>
<LI>Bananas</LI>
<ul>
<li> small bananas </li>
<li> big bananas </li>
</ul>
</OL>
```

Tables and Formatting

What is a Table?

A table is a grid organized into columns and rows, much like a spreadsheet. An example table is shown below. This table consists of sixteen cells organized into rows and columns. But before beginning to use tables in website design, we should consider the role that they fill.

Why do We Use Tables?

Tables were initially developed as a method to organize and display data in columns and rows. This chapter discusses such tables. However, tables later became a tool for Web page layout, and as such provide a possible solution for structured navigation. Frames may also be used to provide structured navigation. However, the use of tables over frames is preferred for this purpose, as earlier Web browsers (e.g. Netscape ver.1.0) do not support frames.

To Do

Read up about tables in your textbooks.

Creating a Data Table

Work through Activity 1 in order to understand how tables are created. Bear in mind that rarely is anything achieved which satisfies all of the stated requirements in the first pass. The key to developing perfect Web pages relies on that old adage: "*Learn from your mistakes!*"

Therefore, as long as a start is made, and mistakes are seen as a learning experience, then the design process will eventually succeed.

Please feel free to experiment at any time. If you make mistakes but manage to correct them, take encouragement from this.

Creating a Table

The objective of this Activity is to create a timetable for CSC5003 students to be displayed on a Web page as shown below:

CSC503 timetable

	Monday	Tuesday	Wednesday	Thursday	Friday
6-7pm	look at website	free	Implementation	free	free
7-8pm	take some notes	free	Implementation	free	free

1. Begin a new Web page in your text editor. The header is shown below. When entering the text, try to spot the deliberate mistake and correct it as necessary.

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>
```

```
HTML Table Design
```

```
</HEAD>
```

```
</TITLE>
```

```
<BODY>
```

```
</BODY >
```

```
</HTML >
```

The correct code is given at the chapter's end.

1. Begin a new Web page in your text editor. The header is shown below. When entering the text, try to spot the deliberate mistake and correct it as necessary.

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>
HTML Table Design
</HEAD>
</TITLE>
<BODY>
</BODY >
</HTML >
```

The correct code is given at the chapter's end.

2. Save your file as tab_ex1.html

3. The next stage is to open the table. To open and close a table, use respectively the <TABLE> and </TABLE> tags within the document's BODY.

```
<HEAD>
<TITLE>
HTML Table Design
</HEAD>
</TITLE>
<BODY>
<TABLE>
</TABLE>
</BODY >
</HTML >
```

4. Save the file and load it in your browser. At first you will notice a blank screen as the table is not visible. A border and rows may be added to make the table visible. If you do not specify a border for the table, it will be displayed without borders. When adding a border, its size can be defined in pixels, for example: <TABLE border=10 style= -width: 80%|| >. Notice the use of the width attribute to set the table to a width of 80% of the screen's size (this can also be defined in pixels). However, it is worth noting that the border attribute is on its way out of the HTML standard! It is better to use CSS by first creating a <style> tag within the <head> tag then leave using only the style attribute within the table tag. `__td` stands for `__tabular data` and `__th` stands for `__tabular header`.

```
<style>
table, th, td {
border: 1px solid black;
}
```

</style>

.....

<TABLE style= -width: 80%| >

5. The <TR> tag is used to add rows. Each row is composed of several data cells. Their dimensions can be defined using width and height attributes: <TD width=25% height=20 bgcolor="darkred"> Notice that the cell's colour can also be defined. Try to create the table below before you look at the solution code under Discussion and Answers at the end of the chapter.

red cell	light blue cell
----------	-----------------

Table with one row and two columns

6. Reopen the file tab_ex1.html in your text editor and make the following amendments to <TABLE> and <tr> tags. Note the <CENTER> tag centers the table horizontally and it also centers the text within each cell in the row.

<TABLE style= "width: 80%" align = "center">

<tr align = "center">

7. Save this file as tab_ex2.html and view it in your browser. It should look as below.

red cell	light blue cell
----------	-----------------

Table with centered text

8. We can see that the text is still not given any specific font. HTML tag is deprecated in version 4.0, onwards (hence it is not supported in HTML5) and now all fonts are set by using CSS. Try to assign the Comic Sans MS font by making the following addition to the style section. Save the file as tab_ex4.html.

font-family: Comic Sans MS;

This sets all the text in in each cell to have the same font. What if you want to have different fonts in each cell? To do this, you can use the <p style > tag within each <TD> tag. Modify your <TD> tags to the following:

<TD width=25% height=70 bgcolor="red"><p style="font-family: verdana">red cell </p></td>

<TD width=75% bgcolor="lightblue"><p style="font-family: Comic Sans MS"> light blue cell </p></td>

9. To add a caption to a table use the <caption> tag within the <table> body. This caption appears on top of the table. Add the caption —Tabling| to your table thus:

<caption> Tabling </caption>

10. Save the file as tab_ex3.html and view it in your browser. It should look as below.

Tabling

red cell	light blue cell
----------	-----------------

Table with caption and text with different font

11. In order to meet the objective of this Activity — that is, to create a timetable for CSC5003 — use the HTML code in the next page. Save this as tab_ex4.html. One extra HTML tag needs to be introduced: the TH tag, which inserts a table header cell. It is similar to the TD tag and has the same attributes (i.e. align, bgcolor, height etc.). However, TH is used to set the cell's text apart from the rest of the table's text, usually setting it bold and slightly larger. Now that you have completed Activity 1, you should have a good idea of how to create a basic data table.

```
<HTML><HEAD><TITLE>HTML Table Design </TITLE><style>table, th, td { border: 1px solid
black; } </style></HEAD><BODY><TABLE style= "width: 80%" align = "center"><caption>CSC503
timetable </caption><tr ><td width=50%></td><th width = 150>Monday </th><th width =
150>Tuesday</th><th width = 150>Wednesday </th><th width = 150>Thursday</th><th width =
150>Friday</th></tr><tr ><td >6-7pm </td><td >Look at website</td><td >free </td><td
>Implementation </td><td >free </td><td >free </td></tr><tr ><td
>7-8pm </td><td >Take some notes</td><td >free </td><td >Implementation </td><td >free
</td><td >free </td></tr></TABLE></BODY >
</HTML >
```

Here are instructions on how to organise and display data in a table:

1. Insert the <TABLE> tag and decide on the table's dimensions (if required)
2. Add a row using the <TR> tag
3. In the newly created row, insert a cell <TD> with the necessary dimensions and other attributes
4. Add the data to be displayed
5. Terminate the data cell </TD>
6. Repeat steps 3-5 as necessary
7. Terminate the row </TR>
8. Repeat steps 2-7 until all the necessary rows have been added
9. Terminate the table </TABLE>

To Do

Look up the basic table structure in your textbooks and on the Internet. Draw up a list of the tags for your own use and reference.

Check your list against this one:

HTML tag

Comments

<TABLE></TABLE>

Table definition and end tag

<CAPTION></CAPTION>

Caption definition and end tag

<TR></TR>

Row definition and end tag



<TD></TD>

Cell definition and end tag

HTML Color Table

This Activity's objective is to write the HTML code to display the following table. Feel free to add more colors.

Some HTML Colors

Colour	Name	hexidecimal	RGB value
	Salmon	FA8072	250-128-114
	Gold	FFD700	255-215-0

To Do

Read up on 'Spanning Rows and Columns' and 'Table Appearance and Colours' in your textbooks. Add the new tags to your list of table related tags.

Using Tables in Page Design

Tables are useful for laying out text and images on in Web page. Before continuing with instructions on how to do this, let us first consider why there is a need to manage layout.

It is important to realise that it is not a monitor's absolute size that is usually of interest, but rather its screen **resolution**.

While a Web browser can manage to layout a document at any resolution, different resolutions do effect the layout and presentation of an HTML document. Resolution is measured in picture elements, called **pixels**. Typical monitor resolutions are 640x480, 800x600, 1024x870, 1280x1024 and 1600x1200.

Resolution and monitor size are independent of one another: a large monitor can have a low resolution, while a small monitor may have a high resolution. Resolution is determined by the hardware,

the user, and the video card driver installed on the computer. A single monitor may have a choice of resolutions.

There is also the issue of a browser's 'live space'. Live space refers to the browser area the Web page is displayed in. This can vary from user to user, as the toolbars and status bars the user chooses to have displayed in the browser will reduce or increase the live space available to a Web page.

It is for all these reasons that the dichotomy between **fixed** and **flexible** Web page design has occurred.

By default, all Web pages are designed with flexibility in mind. Flexibility can be defined as a Web page's ability to resize and adapt to the available resolution, monitor and window sizes. Such an approach has both advantages and disadvantages.

Advantages:

- **Default Setting:** therefore no new tags are needed — the Web page fills entire space.
- **Philosophical:** flexibility is the philosophy of the Web i.e. it should be accessible by the greatest number of users.
- **Realistic:** resolutions, monitor and window sizes are always different. Keeping a Web page flexible allows it to be viewed on many available formats.

Disadvantages:

- **Uncomfortable:** reading text on large monitors is uncomfortable as the lines are too long.
- **Unpredictability:** the designer often cannot predict how a Web page will appear under varying resolutions and live space sizes.
- **Coherence:** on small monitors, everything may not appear correctly

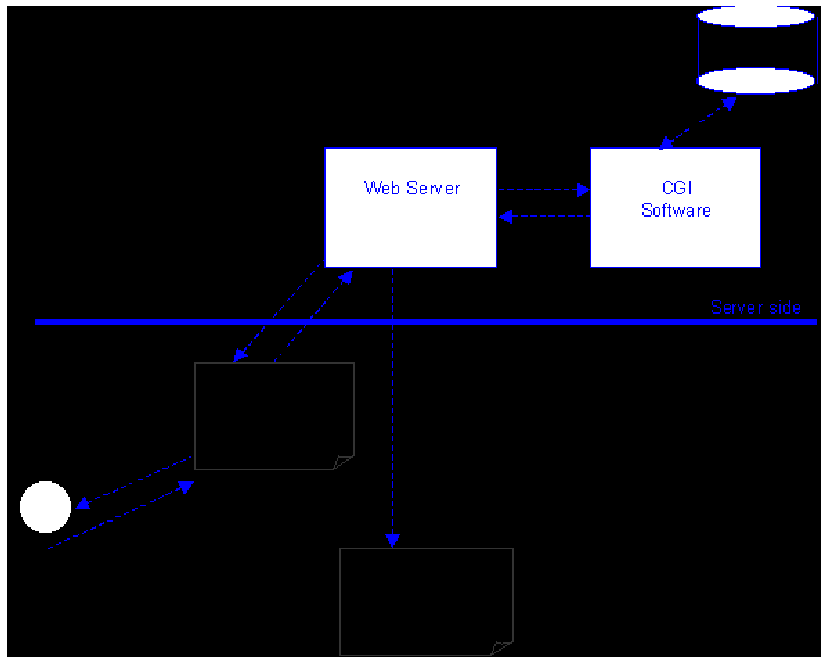
Forms

Forms are best learnt using a hands on approach. To become proficient with HTML forms you need to create many, sorting out the problematic nuances as you go along. Therefore, the main content of the unit is a series of sections: the first is a short introduction to HTML forms; the second discusses each form element, and involves some textbook study. (You may find it more convenient to postpone activities until you have covered all the form elements).

This introduction covers the main form elements. It also explains the process that occurs when a form is submitted. The main elements of forms are: Text fields; Password fields; Text areas; Radio buttons; Check boxes; Menu buttons and scrolling lists; Submit and reset buttons; and file picker. HTML5 defines a number of new input types that can be used in forms. Examples are Email address fields; web address fields; numbers as spin boxes and sliders; date pickers; search boxes; color pickers; form validation; and required fields.

Processing Forms

Although forms could simply be used to display information, HTML provides them in order to supply a way for the user to interact with a Web server. The most widely used method to process the data submitted through a form is to send it to server-side software typically written in a scripting language, although any programming language can be used. The figure below outlines the kind of processing that takes place.



1. The user retrieves a document containing a form from a Web server.
2. The user reads the Web page and interacts with the form it contains.
3. Submitting the form sends the form data to the server for processing.
4. The Web server passes the data to a CGI programs.
5. The CGI software may use database information or store data in a server-side database
6. The CGI software may generate a new Web page for the server to return to the user.
7. The user reads the new Web document and may interact with it.

Typically, form data is sent to a server (or to an email address) as a sequence of pairs, each pair being made up of a name and an associated value. The method that this data uses to arrive at its destination depends on the data encoding. Normally the pairs will be sent as binary-encoded characters, making them straightforward to process by software, and easy to read by humans. For example, an on-line store selling used computer parts might use a form when ordering second-hand disk drives; the form would send to the server for processing information identifying the manufacturer, the model name, and maybe quote price thus:

manufacturer=syquest&model=e3135&price=45

This text represents a sequence of three name/value pairs. The names are **manufacturer**, **model** and **price**, and their associated values are *syquest*, *ez135* and *45*. There is nothing special about the names chosen or the way values are written, except that what is sent depends entirely on what the CGI software expects. If it expected **maker**, **item**, and **cost**, then the data from submitting the form would have to be:

```
maker=syquest&item=ez135&cost=45
```

Quite simply, whatever the processing software expects determines what the HTML form must provide. Often the same person or team develops both form and CGI software, so this is usually of little concern.

Because of the standard way in which the server-side software that process form data is supplied with data, such software is usually referred to as a Common Gateway Interface (CGI) script. Quite often CGI scripts on Unix servers are written in a language called Perl, but languages such as Python are becoming popular; when complex or fast processing is required, C, C++ or Java may use.

To avoid server side programming when developing forms, and to avoid depending on scripts that may require considerable study, we will mostly use a different method of processing form information: email. In fact, it is very useful to submit form data to an email address, particularly in situations when the data should be seen by a human before being processed by software.

Creating Forms

This section explores the main elements found on HTML forms. This is done in a manner matching the way many people develop forms — a little bit at a time. The discussion of each form element involves both reading from your textbook as well as a practical activity. (Some of the activities will take considerable time.) You may prefer to postpone doing Activities 1-7 until you have reached the end of the section on submit and reset buttons.

Starting a Form

All forms start with the `<FORM>` tag and end with `</FORM>`. All other form objects go between these two tags.

The form tag has two main properties: **METHOD** and **ACTION**.

METHOD refers to **post** or **get**. The **post** attribute will send the information from the form as a text document. The **get** attribute is used mostly with search engines, and will not be discussed. We will generally set **METHOD="post"**.

ACTION usually specifies the location of the CGI script that will process the form data. We are not using CGI scripts, and are instead setting this attribute to an imaginary email address (which causes the form data to be emailed to that address).

```
ACTION="mailto:put.your@email.address.here"
```

Putting these together gives us:

```
<FORM METHOD="post" ACTION="mailto:put.your@email.address.here"></FORM>
```

To Do

Read about Forms in your textbooks.

Scripts, and are instead setting this attribute to an imaginary email address (which causes the form data to be emailed to that address).

```
ACTION="mailto:put.your@email.address.here"
```

Putting these together gives us:

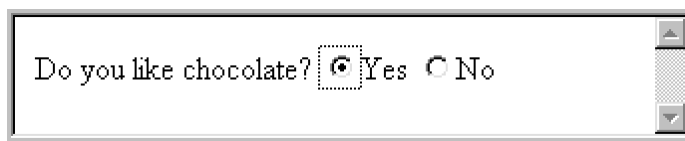
```
<FORM METHOD="post" ACTION="mailto:put.your@email.address.here"></FORM>
```

To Do

Read about Forms in your textbooks.

Radio Buttons

Radio buttons are often used on questionnaires to indicate a person's opinion, or their likes and dislikes. They can also be used for 'yes' or 'no' responses. Radio buttons should be used when only one answer from a set of possible answers may be chosen. This is best illustrated by example:

A screenshot of a web browser window showing a form. The form contains the text "Do you like chocolate?" followed by two radio buttons. The first radio button is selected and is labeled "Yes", and the second radio button is unselected and is labeled "No".

This is achieved with:

Do you like chocolate?

```
<input type="radio" name="chocolate" value="yes">Yes
```

```
<input type="radio" name="chocolate" value="no">No
```

The tag and its attributes are as follows.

<input> is the tag used for most of the form objects.

type="radio" sets the object to a radio button.

name="orbits" labels the entire set of radio buttons. This makes identifying the data easier. For example if the radio button was for the question 'Do you own an automobile?', you might set **name="automobile"**

VALUE=... With a set of radio buttons for one question, it is not enough to provide a name only. We need to give each radio button a value so that the form-processing software (CGI script or email) can determine which radio button has been selected. This is where the **value** attribute comes in. Each of the values above is set to the answer for that radio button. This information is sent with the data when the form is submitted. Hence, the form in the above figure would submit **orbits=strongly_agree**.

CHECKED

This has not been used in the above example. If it were included, the button is set as if it had been clicked. This is useful if one choice should be made the default.

To Do

Read about Radio Buttons in your textbooks.

Radio Buttons

In an HTML document, add these numbered questions and create radio buttons for them.

1. Do you like playing computer games? Yes / No
2. How much did you enjoy the 'Star Wars' Trilogy? I enjoyed it / I did not enjoy it / I have not seen it
3. Do you have an email address? Yes / No
4. Chocolate is delicious? strongly agree / agree / neutral / disagree / strongly disagree
5. Then create four more questions of your own choice that use radio buttons.

Checkboxes are one of the simplest objects that can be placed on a form. These input elements can only be selected and de-selected. Unlike radio buttons, more than one can be selected at a time.

For example, when signing up for a free e-mail account with GMail [<http://www.gmail.com>] or Hotmail [<http://www.hotmail.com>], a user may well have to fill in a series of forms. One of them is often an interests form.

```
<INPUT TYPE="checkbox" NAME="autos" VALUE="yes">Autos<BR>
```

```
<INPUT TYPE="checkbox" NAME="business" VALUE="yes">Business & Finance<BR>
```

```
<INPUT TYPE="checkbox" NAME="movies" VALUE="yes">Movies<BR>
```

```
<INPUT TYPE="checkbox" NAME="music" VALUE="yes">Music<BR>
```

```
<INPUT TYPE="checkbox" NAME="computing" VALUE="yes">Computers & Internet<BR>
```

```
<INPUT TYPE="checkbox" NAME="science" VALUE="yes">Science<BR>
```

```
<INPUT TYPE="checkbox" NAME="sports" VALUE="yes">Sports<BR>
```

<INPUT> is the tag used for most of the form objects. **type="checkbox"** sets the object to a checkbox.

name is used to supply a name to the checkbox.

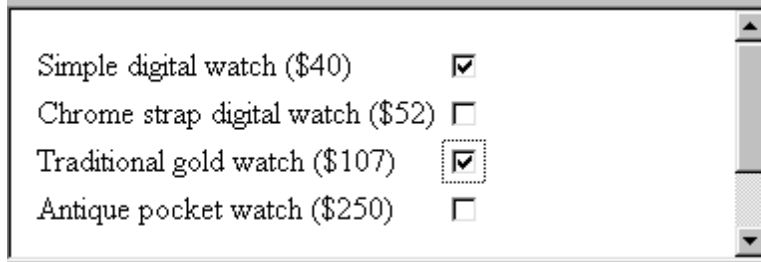
VALUE="yes" if the item is checked, this is the value that will be associated with the name when the form is submitted for processing. Hence, in the above example, **yes** will be associated with each of the name values **movies**, **science** and **sports**.

CHECKED

This has not been used in the above examples. If it were included as an attribute of the tag, the check box would be set as if it had been clicked by the user. This is useful if one or more options are to be offered as defaults.

Check Boxes

Imagine you are developing a website that sells various types of watch. You want to allow customers to select what they want to buy with a set of check boxes, as in the figure below.



Simple digital watch (\$40)	<input checked="" type="checkbox"/>
Chrome strap digital watch (\$52)	<input type="checkbox"/>
Traditional gold watch (\$107)	<input checked="" type="checkbox"/>
Antique pocket watch (\$250)	<input type="checkbox"/>

Linking

Anchors

To link to another file use the `link` tag.

The term URL is the location of the file to be linked to. It could be on a hard or floppy disk — as in

`a:\filename.html` or `c:\my documents\week01\filename.html` — on the same Web server, or on another Web server, as in <http://www.fortunecity.com/username/filename.html>

Simple hypertext links

In this Activity you will create four new Web pages: `index.html`, `filetwo.html`, `filethree.html` and `filefour.html`. You will then link them together using relative URLs.

1. Open Notepad and type in the HTML code shown below. (You may find it easier to cut and paste the code from your Web browser into Notepad rather than enter it yourself.)

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>File name</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<h2>File name</h2>
```

```
<p>
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

```
</P>
```

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse

molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit present luptatum zzril delenit augue dui dolore te feugait nulla facilisi.

```
<P>
<A HREF="index.html">Homepage</A><BR>
<A HREF="filetwo.html">Filetwo</A><BR>
<A HREF="filethree.html">Filethree</A><BR>
<A HREF="filefour">Filefour</A><BR></BODY>
</HTML>
```

2. Save this file as **index.htm**. Save the file a further three times using the file names from the list above. Each time also revise file name in the HTML title and body.
3. You should now have four unique files that link to each other. Test these files in your browser by first opening index.html.
4. Now add the following in index.htm to create a hyperlink to the University of Cape Town website.

```
<P>
<A HREF="http://www.uct.ac.za">University of Cape Town</A>
```

5. Save the file and test it in your browser.

Linking to Email Addresses & other Non-Web Links

The previous examples have used the HTTP protocol to inform the browser to load a Web page when you click a hyperlink. Various other protocols may be used. For example, to create a link to an email address use the 'mailto' protocol. Note that this depends on the user's email programme being correctly configured, and so may not always work. However, this feature is commonly used on the Web to contact the webmaster or to get more information from sites.

The following anchor tag creates a mail link:

```
<A HREF="mailto:username@domainname">Email user</A>
```

You can test this by providing your own email address. This was tested and works for a Gmail address.

A subject line for the email message can also be provided:

```
<A HREF="mailto:username@address.com?SUBJECT=e-mail from a friend">user</A>
```

Other protocols that can be used include: ftp://, news://, telnet:// and gopher://. These protocols often require other software besides the Web browser, and, as with emails, if the software is not correctly installed and configured the links will not work.

Linking to Sections within Documents

Anchor tags can be used to link to a specific location within an HTML file (even within the same HTML file).

Firstly, a location must be defined using the tag: `` with xxx being the location name. A link to the location is created using the tag `link`. If the location is in another document, its file name too must be included as well: `link`

Linking to sections within a document

This Activity sets up links to sections within the documents created by Activity 8

1. Open Notepad and load **index.html**.
2. Copy the following text twice into the body of the file above the hyperlinks in order to create a long file. Rename section two to section three when copying it for the second time.

Section two

`<P>`

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat..

`<P>`

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duiis dolore te feugait nulla facilisi.

`<P>`

3. Now define the location section two by amending the text in the following way.

`section two`

4. Define the location section three in the same way. Save the file.
5. Save this file as filetwo.html, overwriting the previous file.
6. Re-open index.html and add the following hyperlinks to the top of the `<body>` section:
 - Section Two
 - Section Three
 - File Two: Section Two
 - File Two: Section Three
7. Ensure that the links work by reloading index.htm in your Web browser.

Targeting Windows

In modern Web browsers, anchor tags can specify target windows using the target window attribute.

```
<A HREF="URL" TARGET="New_Window"></A>
```

This specifies where the contents of a selected hyperlink should be displayed. It is commonly used with frames or multiple browser windows, allowing the link to be opened in a specified frame or a new browser window. Windows may have names defined for them, but the underscore should not be used for the first character of any target defined in your documents. Such names are reserved for four special target names:

<code>_blank</code>	The browser always loads a target="_blank" linked document in a newly opened, unnamed window.
<code>_self</code>	This target value is the default for all <A> tags that do not specify a target. It causes the target document to be loaded and displayed in the same frame window as the source document.
<code>_parent</code>	This one is useful for framed sites to create navigation links back to the parent window.
<code>_top</code>	<code>_top</code> forces a break out of a framed site, or to take over the browser window. That means, for example, that if a site has been linked to from within a framed site, clicking on the link only brings up the site inside the frame. The <code>_top</code> target attribute forces the link to take over the

entire browser window.

Frames.

The basic frames are deprecated in HTML5 and are out of use. The frameset tag and its helper tags frame/noframes are removed from the modern HTML5 standard. The basic frames had this markup:

```
<FRAMESET COLS="20%,80%">
<FRAME src=left.html>
<FRAME src=right.html>
</FRAMESET>
```

This sets up the frameset to consist of two frames in two columns. The first frame takes 20% of the browser window and the second 80%. A file called left.html will appear in one frame and a file called right.html in the other. In order to view the framed page in your browser you will need to create these two pages.

HTML5 incorporates the inline frame element, **iframe**, which is a HTML page embedded into the current page.

iframes Elements

Web page with one or multiple frames

Create a web page with a single Frame

This Activity sets up a webpage with a single frame.

1. Open a text editor, such as Notepad, and enter the <HEAD> and <BODY> portions of an HTML page.

2. Use the HTML code below to lay out the page with one frame that embeds the UCT website:

```
<iframe src="http://www.uct.ac.za/"></iframe>
```

Save this page as frame1.html and load it in your web browser.

Create a web page with a multiple frames

This Activity sets up a webpage with two frames.

1. Open a text editor, such as Notepad, and enter the <HTML> and <BODY> portions of an HTML page.

```
<iframe src="right.html"></iframe>
<iframe src="left.html"></iframe>
```

Save this page as frame2.html.

2. Begin another document and enter the following code. Enter the following code:

```
<BODY>This is the left frame</BODY>
```

Save this file as left.html in the same folder as file2.html.

3. Begin another document and enter the following code. Save this as right.html. Note that the bgcolor tag was deprecated in HTML5 and now we use the CSS style.

```
<head>
<style>
BODY
{
background-color:blue;
}
</style>
</head>
<BODY>
This is the right frame with a blue background
</BODY>
```

4. Now, load frame2.html in your Web browser. You should see that both frames, with the two files loaded in them as appropriate.

Set Width and Height

Use the height and width attributes to specify the size.

The attribute values are specified in pixels by default, but they can also be in percent (like "80% "), for example:

```
<iframe src="right.html" width = 200 height = 100></iframe>
```

Remove the Border

By default, an iframe has a black border around it.

To remove the border, add the style attribute and use the CSS border property:

```
<iframe src="right.html" style = "border:none"></iframe>
```

Use iframe as a Target for a Link

An iframe can be used as the target frame for a link.

Target iframe

1. In this activity you will create a link that, when clicked, will open the UCT website in a frame.

2. Open frame2.html and make the following changes.

```
<iframe src="right.html" name = "right"></iframe>
<p><a href="http://www.uct.ac.za/" target="right">Go to UCT</a></p>
<p>Click on the link above to open UCT website in the right frame</p>
```

3. Save the file and load in in your browser. Click on ‘_Got to UCT’ and the UCT website should load within the right frame.

Advantages and disadvantages of iframes

The major disadvantages of using iframes are:

- Frames can make the production of a website complicated, although current software addresses this problem.
 - It is easy to create badly constructed websites using frames. The most common mistake is to include a link that creates duplicate Web pages displayed within a frame.
 - Search engines that reference a Web page only give the address of that specific document. This means that search engines might link directly to a page that was intended to be displayed within a frameset.
 - Users have become so familiar with normal navigation using tables, the back button, and so on, that navigating through a site that uses frames can be a problem.
-
- The use of too many frames can put a high workload on the server. A request for, say, ten files, each 1 Kilobyte in size, requires a greater workload than a request for a single 10 Kilobyte file.

The advantages of HTML5 iframes include:

- The main advantage of frames is that it allows the user to view multiple documents within a single Web page.
- It is possible to load pages from different servers in a single frameset.

To Do

Research iframes and their uses. Discuss the following topics with other students on the forum:

- The merits of designing a Web page with and without iframes.
- The alternatives to using iframes.

Try to cite some examples of good and poor website design using frames of your own.

Summary:

HTML is the universal markup language for the Web. HTML lets you format text, add graphics, create links, input forms, frames and tables, etc., and save it all in a text file that any browser can read and display.

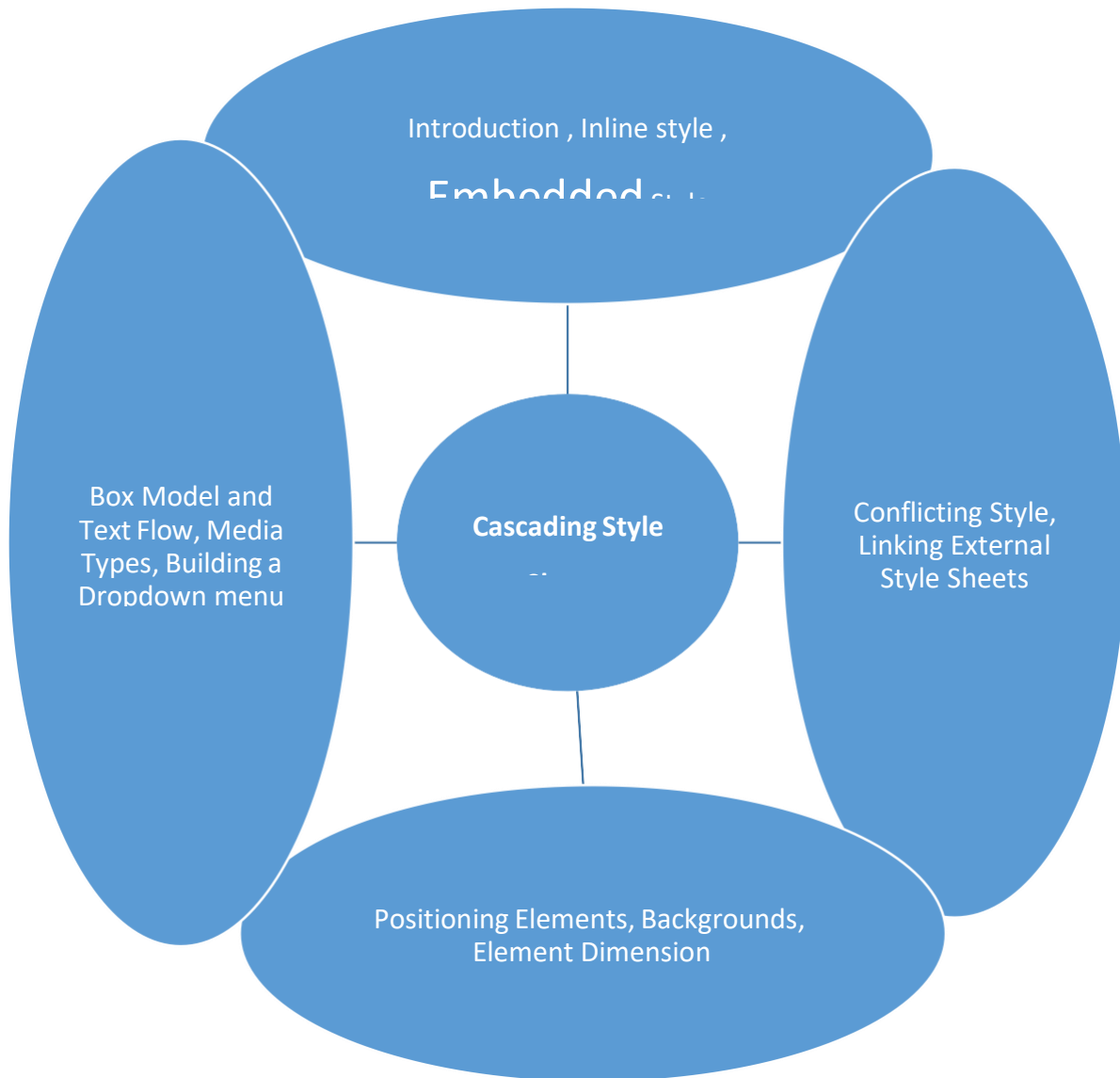
Questions:

1. What is HTML?
2. What is a Tag in HTML?
3. What is the key difference between HTML Elements and Tags?
4. If you want to display some HTML data in a table in tabular format, which HTML tags will you use?
5. What are Attributes in HTML?

6. What is an Anchor tag in HTML?
7. What are Lists in HTML?

UNIT II

Mind Map



Cascading Style Sheets:Introduction

The cascading style sheet standard supplies very powerful tools to control Web page formatting. For instance, consider a university with many departments — each with their own individual design criteria — that is producing a website. It is possible to create a hierarchy of style sheets that allows each department's website to maintain formatting consistency with all the other university sites, while allowing each department to deviate from the format where needed.

The style sheet standard supported by modern browsers is called cascading style sheets, or CSS. CSS files contain a set of rules for the formatting of HTML documents. An example is given below:

```
<html>
<head>
<title>UCT MSc IT Example 1 on style sheets</title>
<style>
BODY {
font-family : "times new roman; margin-left : 20%;
margin-right: 20%; text-align : justify; background : ivory; color : black;
}
P {
text-indent : 2cm;
}
</style>
</head>
```

A style sheet is a collection of rules that describe the format of the HTML tags. In the above example there are six rules describing the format of the BODY tag, and one rule for the P tag. There are two ways:

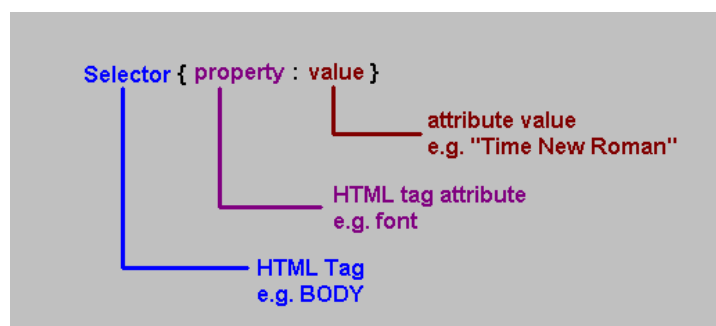
to write style sheets: the technically easier rule-based approach, and an approach that procedurally constructs a style sheet — such an approach is outside the scope of this unit, but feel welcome (if you any spare time) to visit various sites on this topics such as this one [<http://csgs6k1.uwaterloo.ca/~dmg/dsssl/tutorial/tutorial.html>] or search on Google [<http://www.google.com>] for more

Below is a description of the rules used in the above example.

Body - *bcolor* set to "ivory", left and right margins indented relatively by 20%, font set to "Times New Roman" and text colour set to black.

P to indent the first line by and absolute value of two centimeters.

There are three parts to a style sheet rule,



Advantages of Style Sheets

1. **Multiple Styles** - A single document can be presented in multiple styles by using multiple style sheets.
2. **Re-styling** - The use of style sheets (which are separate to the HTML files) allows the quick re-styling of any document, without modifying the original HTML.
3. **Document maintenance** - The ability to re-style many documents allows us to easily make changes to the appearance of many Web pages without separately editing each one.
4. **Consistency** - Style sheets guarantee consistency throughout website.

5. **Optimal file size** - The smaller the files the faster the download. Using style sheets can help minimize file sizes, since, for example, every *< font >*tag, is defined in one place in a style sheet, rather than in multiple places in the HTML file.
6. **Style and structure** - When first developed, HTML was only concerned with document markup and not with the document's formatting. This eventually changed, with more and more functionality being added to HTML to allow for formatting. With the introduction of style sheets, the HTML document is again concerned only with structural document markup—all formatting is now placed in the style sheet.

Disadvantages of Style Sheets

1. **Browser dependency** - Style sheets format things slightly differently on different browsers. Unfortunately, browsers have different support for HTML and style sheets. Newer browsers have largely converged on HTML support so that HTML documents look the same across different browsers. The state of style sheet support is somewhat worse, largely because style sheets are newer than HTML.
2. **Old Browsers** - Some very old browsers (such Netscape Navigator 2) do not support style sheets. All in all, style sheets have only minor disadvantages, and should be used when developing websites.

To Do

Read up on style sheets on your text books. Can you see any advantages or disadvantages that have not been presented here? Also use the Internet to find out more about the usage of style sheets.

Important Note about Rules

Notice in the given example how each rule is separated by a semi-colon (;). If your code is not working, ensure that the semi-colons are present.

There are three ways to apply the above CSS rules to an HTML file:

1. add them in-line to the HTML file itself, attached directly to the relevant HTML tag.
2. embed the rules into the HTML file
3. link the CSS file to the HTML file

Inline styles

In-Line styles are added to individual tags and are usually avoided. Like the FONT tag they clog up HTML documents, making them larger and increasing their download times.

An example of an in-line style is given below:

```
<P style="text-indent: 2cm; color:darkred;">
```

This paragraph has been formatted using the in-line style command.

```
</P><P>
```

This paragraph has not been formatted using the in-line style command.

```
>/P>
```

Embedded Style Sheets

This method avoids duplication within a single HTML document. However, it still has its drawbacks: every Web page on your site needs this embedded style sheet inserted; consequently any updates to the style sheet have to be made to every HTML document that has the style sheet embedded in it. We have already used embedded style sheets as they are the simplest to implement, here is another example:

```
<html>
```

```
<head>
```

```
<title> University of Cape Town, Example on embedded style sheets</title>
```

```
<style> BODY {
```

```
font-family : "times new roman; margin-left : 20%;
```

```
margin-right: 20%; text-align : justify; background : ivory; color :
```

```
darkred;
```

```
}
```

```
P {
```

```
text-indent : 2cm;
```

```
}
```

```
h1,h2,h3{ color:red; margin-left:2cm
```

```
}
```

```
</style>
```

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

This gives you all the advantages of style sheets: by changing a single value in one file the format

change is propagated to all of the HTML documents linked to the style sheet. The style sheet is written just as an embedded style sheet is, but, instead of inserting it in an HTML file, it is saved as a separate

file (usually with a .css extension). Each HTML document then imports the CSS file. There are two ways to import a file:

Linking it

```
< link rel="stylesheet" href=" ../pathname//stylesheet_filename.css" type="tex
```

Importing it

```
<style>
```

```
@import (http://pathname/stylesheet.css);
```

```
</style>
```

Conflicting Style

CSS properties end up competing with one another in where they need to appear in the cascade hierarchy. To resolve such conflicts, you have CSS Cascade. **The CSS Cascade is a way for browsers to resolve conflicting CSS declarations.**

1. Origin Precedence

(a) When in Conflict

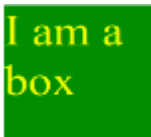
Remember that, an HTML page is read sequentially from top to bottom. When there are two different specified styles for an element, **the declaration at the bottom wins**. For external stylesheets, consider it to be at the position of the declaration of the stylesheet. Because of this, for example, **one should always declare the user stylesheet after the *Bootstrap* stylesheet** so that we can override any bootstrap styles we want.

(b) When not in Conflict

When there are two different declarations for the same element but they focus on different attributes, the styles simply merge, and the element displays properties of both the declarations. Infact, **whenever two declarations are not in conflict with each other, they merge.**

```
<head>
  <style>
    .box{
      color: blue;
      background-color: green;
      width: 150px;
      height: 150px;
    }
    .box{
      color: yellow;
      font-size: 3em;
    }
  </style>
</head>
<body>
  <div class="box">I am a box</div>
</body>
```

There are 2 style declarations. The text-color is in conflict, but other styles are not, so, what will be the output?



Output

According to the rules, the colour, which is in conflict, is received from the bottom declaration. The non-conflicting ones, like background colour, are received from the upper declaration.

2. Inheritance

A child element always inherits the styles of the parent element, unless declared otherwise separately. Even here, if the declarations are not in conflict, they simply merge.

```
<head>
  <style>
    .parent{
      color: blue;
      background-color: green;
      width: 500px;
      height: 500px;
    }
    .box{
      font-size: 3em;
      background-color: red;
      border: 1px solid black;
      margin: 10px;
    }
    .box4{
      color: yellow;
      font-size: 1em;
    }
  </style>
</head>
<body>
  <div class="parent">
    <div class="box">I am a box</div>
    <div class="box">I am a box</div>
    <div class="box">I am a box</div>
    <div class="box box4">I am a box</div>
  </div>
</body>
```

What will happen in the code above? This includes the application of both, the principles of origin as well as inheritance.

As we can see, the text color is inherited from the parent element. The background color is received from the styles of the box. There are 2 conflicting declarations for “*box4*”, both of which have the same specificity (we will come to specificity in the next paragraph). So, the bottom declaration wins, and the font-size becomes *1em*.



As we can see, the text color is inherited from the parent element. The background color is received from the styles of the box. There are 2 conflicting declarations for “*box4*”, both of which have the same specificity (we will come to specificity in the next paragraph). So, the bottom declaration wins, and the font-size becomes *1em*.

3. Specificity

When two different selectors are used but both select the same element, the precedence is decided by specificity rules. We can think of the specificity of an element as a 4-digit number, and the greater number wins. The number is found in the following way:-

1. The thousands digit denotes the presence of inline styles.
2. The hundreds digit denotes the number of IDs specified in the declaration.
3. The tens digit denotes the number of classes specified in the declaration.
4. The one's digit denotes the number of HTML tags specified in the declaration.

Now, try to figure out what will happen in the following code:

```
<head>
  <style>
    .parent{
      color: blue;
      background-color: green;
      width: 500px;
      height: 500px;
    }
    .box4.box{
      color: yellow;
      font-size: 1em;
    }
    #b{
      color: pink;
    }
    div > .box{
      font-size: 3em;
      background-color: red;
      border: 1px solid black;
      margin: 10px;
    }
  </style>
</head>
<body>
```

Let's find the specificity scores for each element:

First box:

There are two selectors for the first box. One is using `-div > .box`", and the other is the inline style.

Let's compare their scores:

Using the rules above, the selector using inline styles gets a score of 1000.

The selector using `-div > .box`" gets a score of 0011.

So, selector using inline style wins (since $1000 > 11$), and the conflicting style, i.e., font-color, should be purple. The other styles should simply merge.

Second box:

Again there are two selectors here, namely `-#b`" and `-div > .box`".

`"#b`" gets a score of 0100, and `-div > .box`" has score of 0011.

Again, as $100 > 11$, the ID selector wins, although there are no conflicting declarations here.

Third box: There is only one selector `-div > .box`”.

Fourth box: The `-div > .box`” is common for this as well. The other one has two classes, and therefore has a score of 0020.

Again, as $20 > 11$, `-.box4.box`” wins. Let's have a look at the output.



Note: You may find the inline style very powerful, and want to use it every time. However, using inline styles is bad practice. The CSS styles should all be at one place, thus one should refrain from using inline styles. Inline-styles are used the majority of the time when you want to quickly check how the element looks with the style.

Linking External Style Sheets

CSS allows us to link external style sheets to our files. This helps us make changes to CSS separately and improves the page load time. External files are specified in `<link>` tag inside `<head>` of the document.

Syntax

The syntax for including external CSS is as follows.

```
<link rel="stylesheet" href="#location">
```

Example: The following examples illustrate how CSS files are embedded &miuns;

HTML file

```
<!DOCTYPE html>

<html>

<head>

<linkrel="stylesheet"type="text/css"href="style.css">
```

```
</head>

<body>

<h2>Demo Text</h2>

<div>

<ul>

<li>This is demo text.</li>

<li>This is demo text.</li>

<li>This is demo text.</li>

<li>This is demo text.</li>

<li>This is demo text.</li>

</ul>

</div>

</body>

</html>
```

CSS file

```
h2 {
  color: red;
}

div {
  background-color: lightcyan;
}
```

Output

This gives the following output



Example: HTML file

```
<!DOCTYPE html>

<html>

<head>

<linkrel="stylesheet" type="text/css" href="style.css">

</head>

<body>

<h2>Demo Heading</h2>

<p>This is demo text. This is demo text. This is demo text. This is demo text. This is demo text. This is demo text. This is demo text. This is demo text. </p>

</body>

</html>
```

CSS file

```
p {
    background: url("https://www.tutorialspoint.com/images/QAicon.png");
    background-origin: content-box;
    background-size: cover;
    box-shadow: 0 0 3px black;
    padding: 20px;
    background-origin: border-box;
```

Output

This gives the following output –



Positioning Elements

The **position** property specifies the type of positioning method used for an element (static, relative, fixed, absolute or sticky).

The position Property

The **position** property specifies the type of positioning method used for an element.

There are five different position values:

- **static**
- **relative**
- **fixed**
- **absolute**
- **sticky**

Elements are then positioned using the top, bottom, left, and right properties. However, these properties will not work unless the **position** property is set first. They also work differently depending on the position value.

position: static;

HTML elements are positioned static by default.

Static positioned elements are not affected by the top, bottom, left, and right properties.

An element with **position: static;** is not positioned in any special way; it is always positioned according to the normal flow of the page:

This <div> element has position: static;

Here is the CSS that is used:

Example

```
div.static {  
  position: static;  
  border: 3px solid #73AD21;  
}
```

position: relative;

An element with **position: relative;** is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

This <div> element has position: relative;

Here is the CSS that is used:

Example

```
div.relative {  
  position: relative;  
  left: 30px;  
  border: 3px solid #73AD21;  
}
```

position: fixed;

An element with **position: fixed**; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

A fixed element does not leave a gap in the page where it would normally have been located.

Notice the fixed element in the lower-right corner of the page. Here is the CSS that is used:

Example

```
div.fixed {  
  position: fixed;  
  bottom: 0;  
  right: 0;  
  width: 300px;  
  border: 3px solid #73AD21;  
}
```

This <div> element has **position: fixed**;

position: absolute;

An element with **position: absolute**; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).

However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

Note: Absolute positioned elements are removed from the normal flow, and can overlap elements.

Here is a simple example:

This <div> element has position: relative;

This <div> element has position: absolute;

Here is the CSS that is used:

Example

```
div.relative {  
  position: relative;  
  width: 400px;  
  height: 200px;  
  border: 3px solid #73AD21;  
}
```

```
div.absolute {  
  position: absolute;  
  top: 80px;  
  right: 0;  
  width: 200px;  
  height: 100px;  
  border: 3px solid #73AD21;  
}
```

position: sticky;

An element with **position: sticky;** is positioned based on the user's scroll position.

A sticky element toggles between **relative** and **fixed**, depending on the scroll position. It is positioned relative until a given offset position is met in the viewport - then it "sticks" in place (like position:fixed).

Note: Internet Explorer does not support sticky positioning. Safari requires a `-webkit-` prefix (see example below). You must also specify at least one of **top**, **right**, **bottom** or **left** for sticky positioning to work.

In this example, the sticky element sticks to the top of the page (**top: 0**), when you reach its scroll position.

Example

```
div.sticky {  
  position: -webkit-sticky; /* Safari */  
  position: sticky;  
  top: 0;  
  background-color: green;  
  border: 2px solid #4CAF50;  
}
```

Positioning Text in an Image

How to position text over an image:

Backgrounds

The CSS background properties are used to add background effects for elements.



In these chapters, you will learn about the following CSS background properties:

- **background-color**
- **background-image**
- **background-repeat**
- **background-attachment**
- **background-position**
- **background** (shorthand property)

CSS background-color

The `background-color` property specifies the background color of an element.

Example

The background color of a page is set like this:

```
body {  
  background-color: lightblue;  
}
```

With CSS, a color is most often specified by:

- a valid color name - like "red"
- a HEX value - like "#ff0000"
- an RGB value - like "rgb(255,0,0)"

Look at [CSS Color Values](#) for a complete list of possible color values.

Other Elements

You can set the background color for any HTML elements:

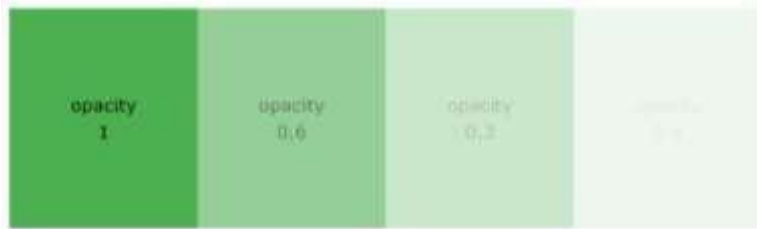
Example

Here, the `<h1>`, `<p>`, and `<div>` elements will have different background colors:

```
h1 {  
  background-color: green;  
}  
  
div {  
  background-color: lightblue;  
}  
  
p {  
  background-color: yellow;  
}
```

Opacity / Transparency

The `opacity` property specifies the opacity/transparency of an element. It can take a value from 0.0 - 1.0. The lower value, the more transparent:



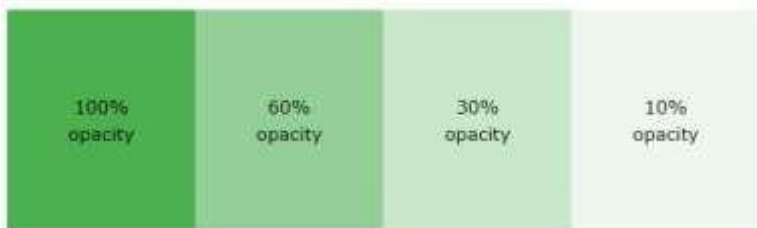
Example

```
div {  
  background-color: green;  
  opacity: 0.3;  
}
```

Note: When using the `opacity` property to add transparency to the background of an element, all of its child elements inherit the same transparency. This can make the text inside a fully transparent element hard to read.

Transparency using RGBA

If you do not want to apply opacity to child elements, like in our example above, use **RGBA** color values. The following example sets the opacity for the background color and not the text:



You learned from our [CSS Colors Chapter](#), that you can use RGB as a color value. In addition to RGB, you can use an RGB color value with an **alpha** channel (RGBA) - which specifies the opacity for a color.

An RGBA color value is specified with: `rgba(red, green, blue, alpha)`. The *alpha* parameter is a number between 0.0 (fully transparent) and 1.0 (fully opaque).

Tip: You will learn more about RGBA Colors in our [CSS Colors Chapter](#).

Example

```
div {  
  background: rgba(0, 128, 0, 0.3) /* Green background with 30% opacity */  
}
```

The CSS Background Color Property

Property	Description
background-color	Sets the background color of an

Element Dimension

The CSS **height** and **width** properties are used to set the height and width of an element.

The CSS **max-width** property is used to set the maximum width of an element.

This element has a height of 50 pixels and a width of 100%.

CSS Setting height and width

The **height** and **width** properties are used to set the height and width of an element.

The height and width properties do not include padding, borders, or margins. It sets the height/width of the area inside the padding, border, and margin of the element.

CSS height and width Values

The **height** and **width** properties may have the following values:

- **auto** - This is default. The browser calculates the height and width
- **length** - Defines the height/width in px, cm etc.
- **%** - Defines the height/width in percent of the containing block
- **initial** - Sets the height/width to its default value
- **inherit** - The height/width will be inherited from its parent value

CSS height and width Examples

This element has a height of 200 pixels and a width of 50%

Example

Set the height and width of a <div> element:

```
div {  
  height: 200px;  
  width: 50%;  
  background-color: powderblue;  
}
```

This element has a height of 100 pixels and a width of 500 pixels.

Example

Set the height and width of another <div> element:

```
div {  
  height: 100px;  
  width: 500px;  
  background-color: powderblue;  
}
```

Note: **Remember that the `height` and `width` properties do not include padding, borders, or margins! They set the height/width of the area inside the padding, border, and margin of the element!**

Setting max-width

The `max-width` property is used to set the maximum width of an element.

The `max-width` can be specified in *length values*, like px, cm, etc., or in percent (%) of the containing block, or set to none (this is default. Means that there is no maximum width).

The problem with the `<div>` above occurs when the browser window is smaller than the width of the element (500px). The browser then adds a horizontal scrollbar to the page.

Using `max-width` instead, in this situation, will improve the browser's handling of small windows.

Tip: Drag the browser window to smaller than 500px wide, to see the difference between the two divs!

This element has a height of 100 pixels and a max-width of 500 pixels.

Note: If you for some reason use both the `width` property and the `max-width` property on the same element, and the value of the `width` property is larger than the `max-width` property; the `max-width` property will be used (and the `width` property will be ignored).

Example

This `<div>` element has a height of 100 pixels and a max-width of 500 pixels:

```
div {  
  max-width: 500px;  
  height: 100px;  
  background-color: powderblue;  
}
```

All CSS Dimension Properties

Property	Description
----------	-------------

height	Sets the height of an element
max-height	Sets the maximum height of an element
max-width	Sets the maximum width of an element
min-height	Sets the minimum height of an element
min-width	Sets the minimum width of an element
width	Sets the width of an element

Box Model and Text Flow

The CSS Box Model

In CSS, the term "box model" is used when talking about design and layout.

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



Explanation of the different parts:

- **Content** - The content of the box, where text and images appear
- **Padding** - Clears an area around the content. The padding is transparent
- **Border** - A border that goes around the padding and content
- **Margin** - Clears an area outside the border. The margin is transparent

The box model allows us to add a border around elements, and to define space between elements.

Example

Demonstration of the box model:

```
div {  
  width: 300px;  
  border: 15px solid green;  
  padding: 50px;  
  margin: 20px;  
}
```

Width and Height of an Element

In order to set the width and height of an element correctly in all browsers, you need to know how the box model works.

Important: When you set the width and height properties of an element with CSS, you just set the width and height of the **content area**. To calculate the full size of an element, you must also add padding, borders and margins.

Example

This `<div>` element will have a total width of 350px:

```
div {  
  width: 320px;  
  padding: 10px;  
  border: 5px solid gray;  
  margin: 0;  
}
```

Here is the calculation:

```
320px (width)  
+ 20px (left + right padding)  
+ 10px (left + right border)  
+ 0px (left + right margin)  
= 350px
```

The total width of an element should be calculated like this:

Total element width = width + left padding + right padding + left border + right border + left margin + right margin

The total height of an element should be calculated like this:

Total element height = height + top padding + bottom padding + top border + bottom border + top margin + bottom margin

Exercise:

Set the width of the <div> element to "200px".

```
<style>
[ ] {
[ ]
}
</style>
<body>
<div>
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
</div>
</body>
```

Media Types

Introduced Media Types

The **@media** rule, introduced in CSS2, made it possible to define different style rules for different media types.

Examples:

You could have one set of style rules for computer screens, one for printers, one for handheld devices, one for television-type devices, and so on.

Unfortunately these media types never got a lot of support by devices, other than the print media type.

CSS3 Introduced Media Queries

Media queries in CSS3 extended the CSS2 media types idea: Instead of looking for a type of device, they look at the capability of the device.

Media queries can be used to check many things, such as:

- width and height of the viewport
- width and height of the device
- Orientation (is the tablet/phone in landscape or portrait mode?)
- resolution

Using media queries are a popular technique for delivering a tailored style sheet to desktops, laptops, tablets, and mobile phones (such as iPhone and Android phones).

Browser Support

The numbers in the table specifies the first browser version that fully supports the `@media` rule.

Property					
@media	21.0	9.0	3.5	4.0	9.0

Media Query Syntax

A media query consists of a media type and can contain one or more expressions, which resolve to either true or false.

```
@media not |only mediatype and (expressions) {  
    CSS-Code;  
}
```

The result of the query is true if the specified media type matches the type of device the document is being displayed on and all expressions in the media query are true. When a media query is true, the corresponding style sheet or style rules are applied, following the normal cascading rules.

Unless you use the not or only operators, the media type is optional and the `all` type will be implied.

You can also have different stylesheets for different media:

```
<link rel="stylesheet" media="mediatype and|not|only (expressions)"  
href="print.css">CSS3 Media Types
```

Value	Description
all	Used for all media type devices
print	Used for printers
screen	Used for computer screens, tablets, smart-phones etc.
speech	Used for screenreaders that "reads" the page out loud

Media Queries Simple Examples

One way to use media queries is to have an alternate CSS section right inside your style sheet.

The following example changes the background-color to lightgreen if the viewport is 480 pixels wide or wider (if the viewport is less than 480 pixels, the background-color will be pink):

Example

```
@media screen and (min-width: 480px) {  
  body {  
    background-color: lightgreen;  
  }  
}
```

The following example shows a menu that will float to the left of the page if the viewport is 480 pixels wide or wider (if the viewport is less than 480 pixels, the menu will be on top of the content):

Example

```
@media screen and (min-width: 480px) {  
  #leftsidebar { width: 200px; float: left; }  
  #main { margin-left: 216px; }  
}
```

Building a Dropdown menu

HTML Use any element to open the dropdown content, e.g. a ``, or a `<button>` element.

Use a container element (like `<div>`) to create the dropdown content and add whatever you want inside of it.

Wrap a `<div>` element around the elements to position the dropdown content correctly with CSS.

CSS The `.dropdown` class uses `position:relative`, which is needed when we want the dropdown content to be placed right below the dropdown button (using `position:absolute`).

The `.dropdown-content` class holds the actual dropdown content. It is hidden by default, and will be displayed on hover (see below). Note the `min-width` is set to 160px. Feel free to change this. **Tip:** If you want the width of the dropdown content to be as wide as the dropdown button, set the `width` to 100% (and `overflow:auto` to enable scroll on small screens).

Instead of using a border, we have used the CSS `box-shadow` property to make the dropdown menu look like a "card".

The `:hover` selector is used to show the dropdown menu when the user moves the mouse over the dropdown button.

Create a dropdown box that appears when the user moves the mouse over an element.

Example

```
<style>
.dropdown {
  position: relative;
  display: inline-block;
}
.dropdown-content {
  display: none;
  position: absolute;
  background-color: #f9f9f9;
  min-width: 160px;
  box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);
  padding: 12px 16px;
  z-index: 1;
}
.dropdown:hover .dropdown-content {
  display: block;
}
</style>
<div class="dropdown">
  <span>Mouse over me</span>
  <div class="dropdown-content">
    <p>Hello World!</p>
  </div>
</div>
```

Dropdown Menu

Create a dropdown menu that allows the user to choose an option from a list:

This example is similar to the previous one, except that we add links inside the dropdown box and style them to fit a styled dropdown button:

Example

```
<style>
/* Style The Dropdown Button */
.dropbtn {
```

```
background-color: #4CAF50;
color: white;
padding: 16px;
font-size: 16px;
border: none;
cursor: pointer;
}
/* The container <div> - needed to position the dropdown content */
.dropdown {
position: relative;
display: inline-block;
}
/* Dropdown Content (Hidden by Default) */
.dropdown-content {
display: none;
position: absolute;
background-color: #f9f9f9;
min-width: 160px;
box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);
z-index: 1;
}
/* Links inside the dropdown */
.dropdown-content a {
color: black;
padding: 12px 16px;
text-decoration: none;
display: block;
}
/* Change color of dropdown links on hover */
.dropdown-content a:hover {background-color: #f1f1f1}
/* Show the dropdown menu on hover */
.dropdown:hover .dropdown-content {
display: block;
}
```

/ Change the background color of the dropdown button when the dropdown content is shown */*

```
.dropdown:hover .dropbtn {  
  background-color: #3e8e41;  
}  
</style>  
<div class="dropdown">  
  <button class="dropbtn">Dropdown</button>  
  <div class="dropdown-content">  
    <a href="#">Link 1</a>  
    <a href="#">Link 2</a>  
    <a href="#">Link 3</a>  
  </div>  
</div>
```

Right-aligned Dropdown Content

Left

Right

If you want the dropdown menu to go from right to left, instead of left to right, add **right: 0;**

Example

```
.dropdown-content {  
  right: 0;  
}
```

Dropdown Image

How to add an image and other content inside the dropdown box.

Hover over the image:

```
<!DOCTYPE html>  
<html>  
<head>  
<style>  
.dropdown {  
  position: relative;  
  display: inline-block;  
}  
.dropdown-content {
```



```

    display: none;
    position: absolute;
    background-color: #f9f9f9;
    min-width: 160px;
    box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);
    z-index: 1;
}
.dropdown:hover .dropdown-content {
    display: block;
}
.desc {
    padding: 15px;
    text-align: center;
}
</style>
</head>
<body>
<h2>Dropdown Image</h2>
<p>Move the mouse over the image below to open the dropdown content.</p>
<div class="dropdown">

<div class="dropdown-content">

<div class="desc">Beautiful Cinque Terre</div>
</div>
</div>
</body>
</html>

```

Dropdown Navbar

How to add a dropdown menu inside a navigation bar.

```

<!DOCTYPE html>
<html>
<head>
<style>

```

```
ul {  
  list-style-type: none;  
  margin: 0;  
  padding: 0;  
  overflow: hidden;  
  background-color: #333;  
}  
li {  
  float: left;  
}  
li a, .dropbtn {  
  display: inline-block;  
  color: white;  
  text-align: center;  
  padding: 14px 16px;  
  text-decoration: none;  
}  
li a:hover, .dropdown:hover .dropbtn {  
  background-color: red;  
}  
li.dropdown {  
  display: inline-block;  
}  
.dropdown-content {  
  display: none;  
  position: absolute;  
  background-color: #f9f9f9;  
  min-width: 160px;  
  box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);  
  z-index: 1;  
}  
.dropdown-content a {  
  color: black;  
  padding: 12px 16px;
```

```

    text-decoration: none;
    display: block;
    text-align: left;
}
.dropdown-content a:hover {background-color: #f1f1f1;}
.dropdown:hover .dropdown-content {
    display: block;
}
</style>
</head>
<body>
<ul>
<li><a href="#home">Home</a></li>
<li><a href="#news">News</a></li>
<li class="dropdown">
<a href="javascript:void(0)" class="dropbtn">Dropdown</a>
<div class="dropdown-content">
<a href="#">Link 1</a>
<a href="#">Link 2</a>
<a href="#">Link 3</a>
</div>
</li>
</ul>
<h3>Dropdown Menu inside a Navigation Bar</h3>
<p>Hover over the "Dropdown" link to see the dropdown menu.</p>
</body></html>

```

Summary:

CSS stands for Cascading Style Sheets. It is a language designed to specify the overall appearance of WebPages as well as the appearance and structure of the text and elements such as images and buttons on WebPages and their layout.

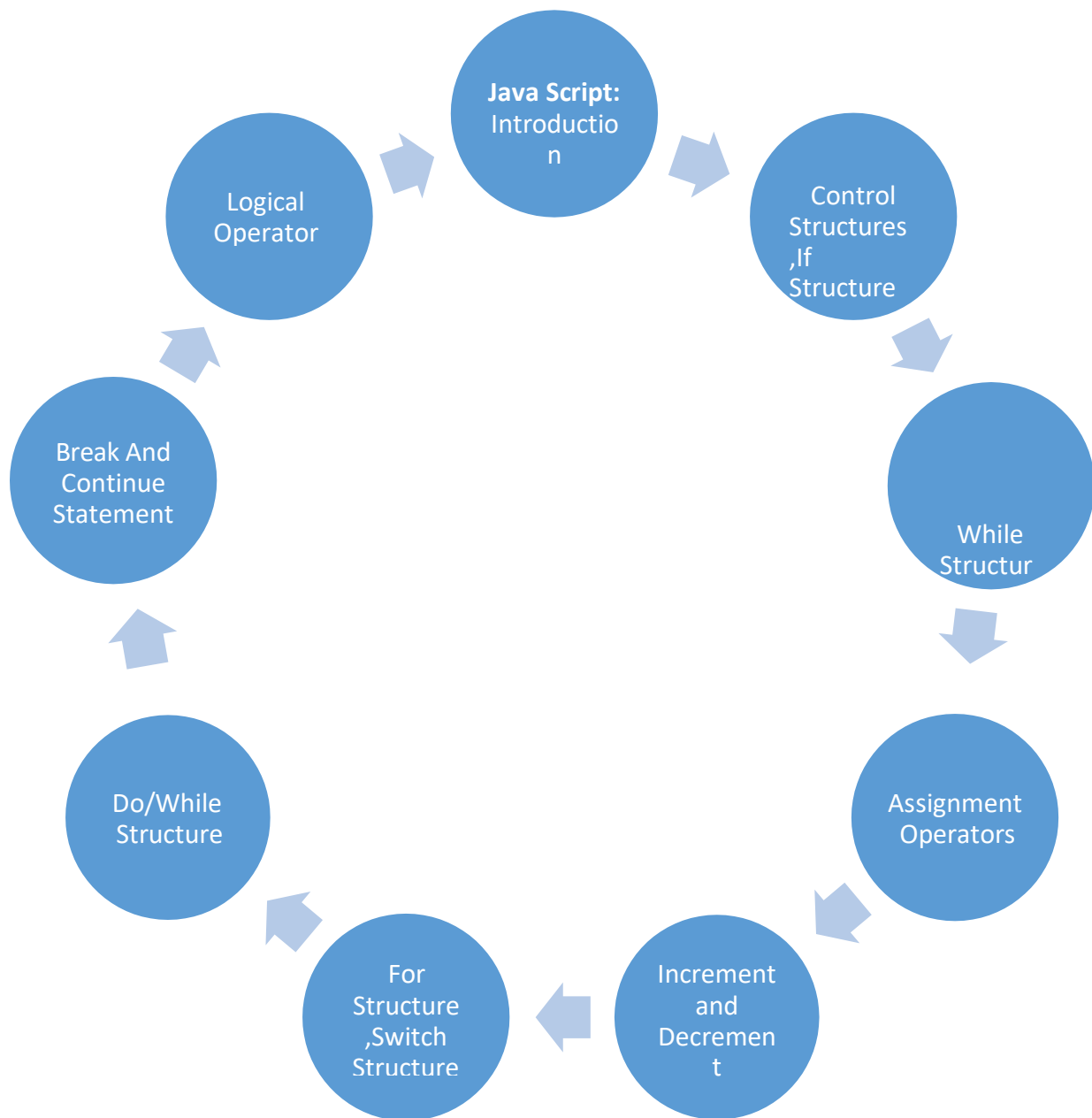
Questions:

1. What are Inline styles?

2. Explain Embedded Style Sheets?
3. Define Conflicting Style?
4. Explain about Linking External Style Sheets?
5. What are the Positioning Elements available in CSS? Explain it.
6. Explain CSS Backgrounds?
7. Define Element Dimension?
8. Describe Box Model and Text Flow?
9. Define Media Types?
10. Explain Building a Dropdown menu?

Unit III

Mind Map

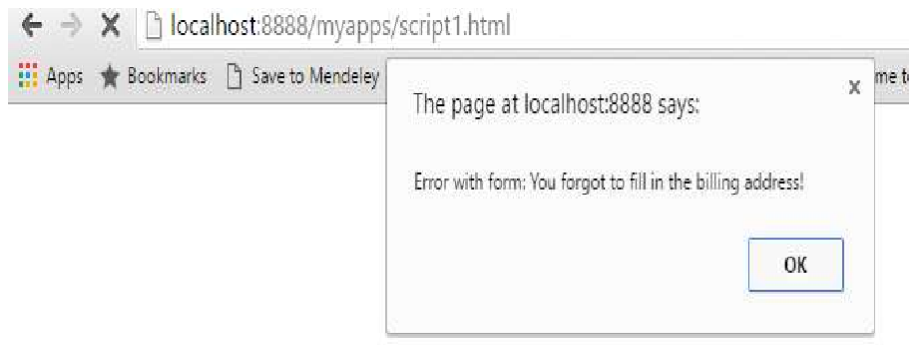


Java Script: Introduction

Web browsers were originally designed to interpret HTML with two primary purposes: to render documents marked up in HTML to an acceptable level of quality, and, crucially, to be able to follow hyperlinks to resources. As the Web grew, so did the demand for more sophisticated Web content. Among many other extensions, graphics, forms, and tables were added to the HTML standard. With the exception of forms, there is nothing in HTML that supports interaction with the user. Given the ubiquity of Web browsers, and the effort which millions of ordinary people have put into learning to use them, they provide an almost universal starting point for interacting with complex systems, particularly

commercial, Internet based systems. Hence the need for sophisticated interaction facilities within Web browsers.

The main means for providing interactivity within HTML documents is the JavaScript programming language. HTML documents can include JavaScript programs that are interpreted (i.e. run) by the Web browser displaying the Web document. In a real sense, JavaScript allows a Web document to interact with its environment — that is, with the browser that is displaying it. Ultimately, it lets the Web document become more interactive, to the user's benefit. For example, the following message could be given to a user when they submit a form with a missing field:



The above message can be shown with the following JavaScript code.

```
<SCRIPT>  
window.alert('Error with form: You forgot to fill in the billing address!')  
</SCRIPT>
```

The JavaScript code is contained within the `<SCRIPT>` and `</SCRIPT>` tags. Everything between those tags must conform to the JavaScript standard (the standard itself is an ECMA International standard, called ECMAScript). The above statement is an instruction to the browser requesting that an alert box display the message "Error with form: You forgot to fill in the billing address!".

This unit will later cover another way to include JavaScript in HTML documents. It is worth noting for now that the `<SCRIPT>` tag can include a language attribute to ensure the browser interprets the enclosed commands as JavaScript, since other languages have, in the past, been used (such as VBScript, which is no longer used in new websites, and is supported by very few browsers). For simplicity, we will use the attribute's default value (of JavaScript) by omitting the attribute from the `<SCRIPT>` tag.

Control Structures

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
- Use **else** to specify a block of code to be executed, if the same condition is false
- Use **else if** to specify a new condition to test, if the first condition is false
- Use **switch** to specify many alternative blocks of code to be executed

If Structure

Use the **if** statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax

```
if (condition) {  
  
    // block of code to be executed if the condition is true  
  
}
```

Note that **if** is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

Example

Make a "Good day" greeting if the hour is less than 18:00.

```
if (hour < 18) {  
    greeting = "Good day";  
}
```

The result of greeting will be:

Good day

The else Statement

Use the **else** statement to specify a block of code to be executed if the condition is false.

```
if (condition) {  
  
    // block of code to be executed if the condition is true  
  
} else {  
  
    // block of code to be executed if the condition is false  
  
}
```

Example

If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

```
if (hour < 18) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

The result of greeting will be:

Good day

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {  
    // block of code to be executed if condition1 is true  
} else if (condition2) {  
    // block of code to be executed if the condition1 is false and condition2 is true  
} else {  
    // block of code to be executed if the condition1 is false and condition2 is false  
}
```

Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
if (time < 10) {  
    greeting = "Good morning";  
} else if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```


The result of greeting will be:

Grade

Do/While Structure

While Structure

Loops can execute a block of code as long as a specified condition is true.

The While Loop

The `while` loop loops through a block of code as long as a specified condition is true.

Syntax

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

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

Example

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

If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The Do While Loop

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

Syntax

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

Example

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

Example

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

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y

<code>*=</code>	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	<code>x %= y</code>	<code>x = x % y</code>
<code>**=</code>	<code>x **= y</code>	<code>x = x ** y</code>

The **addition assignment** operator (`+=`) adds a value to a variable.

Assignment

```
let x = 10;
```

```
x += 5;
```

Assignment operators are fully described in the [JS Assignment](#) chapter.

Increment and Decrement Operators

The JavaScript Increment and Decrement Operators useful to increase or decrease the value by 1. For instance, Incremental operator `++` used to increase the existing variable value by 1 (`x = x + 1`). The decrement operator `--` is used to decrease or subtract the existing value by 1 (`x = x - 1`).

For Structure

Loops can execute a block of code a number of times.

The syntax for both the increment and decrement operators in JavaScript is

- Increment Operator : `++x` or `x++`
- Decrement Operator: `--x` or `x--`

Example

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title> Increment and Decrement Operators in JavaScript </title>
```

```

</head>
<body>
<script>
var x = 10, y = 20;
document.write("<b>----INCREMENT OPERATOR EXAMPLE---- </b>");
document.write("<br \> Value of x : "+ x); //Original Value
document.write("<br \> Value of x : "+ x++); // Using increment Operator
document.write("<br \> Value of x : "+ x + "<br \>"); //Incremented value
-----
document.write("<br \><b>----DECREMENT OPERATOR EXAMPLE---- </b>");
document.write("<br \> Value of y : "+ y); //Original Value
document.write("<br \> Value of y : "+ y--); // using decrement Operator
document.write("<br \> Value of y : "+ y); //decremented value
</script>
</body>
</html>

```

JavaScript Prefix and Postfix

If you observe the above syntax, we can assign the JavaScript increment and decrement operators either before operand or after the operand. When ++ or -- is used before operand like: ++x, --x then we call it as prefix, if ++ or -- is used after the operand like: x++ or x-- then we called it as postfix.

Let's explore the JavaScript prefix and postfix

1. ++i (Pre increment): It will increment the value of i even before assigning it to the variable i.
2. i++ (Post-increment): The operator returns the variable value first (i.e, i value) then only i value will incremented by 1.
3. --i (Pre decrement): It decrements the value of i even before assigning it to the variable i.
4. i-- (Post decrement): The [JavaScript](#) operator returns the variable value first (i.e., i value), then only i value decrements by 1.

JavaScript Prefix and Postfix Example

This example will show you, How to use JavaScript Increment and Decrement Operators as the Prefix and Postfix in JavaScript

```

<!DOCTYPE html>
<html>
<head>
<title>javascript prefix and Postfix </title>

```

```
</head>
<body>
<script>
var x = 10, y = 20, a = 5, b = 4;
document.write("<b>----PRE INCREMENT OPERATOR EXAMPLE---- </b>");
document.write("<br \> Value of X : " + x); //Original Value
document.write("<br \> Value of X : "+ (++x)); // Using increment Operator
document.write("<br \> Value of X Incremented: " + x + "<br \>"); //Incremented value
-----
document.write("<br \><b>----POST INCREMENT OPERATOR EXAMPLE --- </b>");
document.write("<br \> Value of Y : "+ y); //Original Value
document.write("<br \> Value of Y : "+ y++); // Using increment Operator
document.write("<br \> Value of Y Incremented: "+ y + "<br \>"); //Incremented value
-----
document.write("<br \><b>----PRE DECREMENT OPERATOR EXAMPLE --- </b>");
document.write("<br \> Value of A : "+ a); //Original Value
document.write("<br \> Value of A : "+ --a); // using decrement Operator
document.write("<br \> Value of A Decrementd: "+ a + "<br \>"); //decremented value
-----
document.write("<br \><b>----POST DECREMENT OPERATOR EXAMPLE---- </b>");
document.write("<br \> Value of B : "+ b); //Original Value
document.write("<br \> Value of B : "+ b--); // using decrement Operator
document.write("<br \> Value of B Decrementd: "+ b + "<br \>"); //decremented value
</script>
</body>
</html>
```

```
----PRE INCREMENT OPERATOR EXAMPLE----
Value of X : 10
Value of X : 11
Value of X Incremented: 11

----POST INCREMENT OPERATOR EXAMPLE----
Value of Y : 20
Value of Y : 20
Value of Y Incremented: 21

----PRE DECREMENT OPERATOR EXAMPLE----
Value of A : 5
Value of A : 4
Value of A Decrementd: 4

----POST DECREMENT OPERATOR EXAMPLE----
Value of B : 4
Value of B : 4
Value of B Decrementd: 3
```

JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

Instead of writing:

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

You can write:

```
for (let i = 0; i < cars.length; i++) {
  text += cars[i] + "<br>";
}
```

Different Kinds of Loops

JavaScript supports different kinds of loops:

- `for` - loops through a block of code a number of times
- `for/in` - loops through the properties of an object
- `for/of` - loops through the values of an iterable object
- `while` - loops through a block of code while a specified condition is true

- `do/while` - also loops through a block of code while a specified condition is true

The For Loop

The `for` loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {  
  
    // code block to be executed  
  
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example

```
for (let i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}
```

From the example above, you can read:

Statement 1 sets a variable before the loop starts (`let i = 0`).

Statement 2 defines the condition for the loop to run (`i` must be less than 5).

Statement 3 increases a value (`i++`) each time the code block in the loop has been executed.

Statement 1

Normally you will use statement 1 to initialize the variable used in the loop (`let i = 0`).

This is not always the case, JavaScript doesn't care. Statement 1 is optional.

You can initiate many values in statement 1 (separated by comma):

Example

```
for (let i = 0, len = cars.length, text = ""; i < len; i++) {  
    text += cars[i] + "<br>";  
}
```

And you can omit statement 1 (like when your values are set before the loop starts):

Example

```
let i = 2;  
let len = cars.length;  
let text = "";  
for (; i < len; i++) {  
    text += cars[i] + "<br>";  
}
```

Statement 2

Often statement 2 is used to evaluate the condition of the initial variable.

This is not always the case, JavaScript doesn't care. Statement 2 is also optional.

If statement 2 returns true, the loop will start over again, if it returns false, the loop will end.

If you omit statement 2, you must provide a **break** inside the loop. Otherwise the loop will never end. This will crash your browser. Read about breaks in a later chapter of this tutorial.

Statement 3

Often statement 3 increments the value of the initial variable.

This is not always the case, JavaScript doesn't care, and statement 3 is optional.

Statement 3 can do anything like negative increment (`i--`), positive increment (`i = i + 15`), or anything else.

Statement 3 can also be omitted (like when you increment your values inside the loop):

Example

```
let i = 0;
let len = cars.length;
let text = "";
for (; i < len; ) {
  text += cars[i] + "<br>";
  i++;
}
```

Loop Scope

Using `var` in a loop:

Example

```
var i = 5;
for (var i = 0; i < 10; i++) {
  // some code
}
// Here i is 10
```

Using `let` in a loop:

Example

```
let i = 5;
for (let i = 0; i < 10; i++) {
  // some code
}
// Here i is 5
```

In the first example, using `var`, the variable declared in the loop redeclares the variable outside the loop. In the second example, using `let`, the variable declared in the loop does not redeclare the variable outside the loop.

When `let` is used to declare the `i` variable in a loop, the `i` variable will only be visible within the loop.

Switch Structure

The **JavaScript switch statement** is used to execute one code from multiple expressions. It is just like else if statement that we have learned in previous page. But it is convenient than *if..else..if* because it can be used with numbers, characters etc.

The signature of JavaScript switch statement is given below.

1. `switch(expression){`
2. `case value1:`
3. `code to be executed;`
4. `break;`
5. `case value2:`
6. `code to be executed;`
7. `break;`
8. `.....`
- 9.
10. `default:`
11. `code to be executed if above values are not matched;`
12. `}`

Let's see the simple example of switch statement in javascript.

1. `<script>`
2. `var grade='B';`
3. `var result;`
4. `switch(grade){`
5. `case 'A':`
6. `result="A Grade";`

7. break;
8. case 'B':
9. result="B Grade";
10. break;
11. case 'C':
12. result="C Grade";
13. break;
14. default:
15. result="No Grade";
16. }
17. document.write(result);
18. </script>

Output of the above example

B

Break and Continue Statement

The `break` statement "jumps out" of a loop.

The `continue` statement "jumps over" one iteration in the loop.

The Break Statement

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch()` statement.

The `break` statement can also be used to jump out of a loop:

Example

```
for (let i = 0; i < 10; i++) {  
  if (i === 3) { break; }  
  text += "The number is " + i + "<br>";  
}
```

In the example above, the `break` statement ends the loop ("breaks" the loop) when the loop counter (i) is 3.

The Continue Statement

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 3:

Example

```
for (let i = 0; i < 10; i++) {  
  if (i === 3) { continue; }  
  text += "The number is " + i + "<br>";  
}
```

JavaScript Labels

To label JavaScript statements you precede the statements with a label name and a colon:

label:

statements

The `break` and the `continue` statements are the only JavaScript statements that can "jump out of" a code block.

Syntax:

`break labelname;`

`continue`

`labelname;`

The `continue` statement (with or without a label reference) can only be used to **skip one loop iteration**.

The `break` statement, without a label reference, can only be used to **jump out of a loop or a switch**.

With a label reference, the break statement can be used to **jump out of any code block**:

Example

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];  
list: {  
  text += cars[0] + "<br>";  
  text += cars[1] + "<br>";  
  break list;  
  text += cars[2] + "<br>";  
  text += cars[3] + "<br>";  
}
```

A code block is a block of code between { and }.

Logical Operators

JavaScript Logical Operators

Operator	Description
&&	logical and
	logical or
!	logical not

Logical operators are fully described in the [JS Comparisons](#) chapter.

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Type operators are fully described in the [JS Type Conversion](#) chapter.

Summary:

At the end of this chapter you will be able to:

- Explain the differences between JavaScript and Java;
- Write HTML files using some basic JavaScript tags and objects.

Control Structures can be considered as **the building blocks of computer programs**. They are commands that enable a program to -take decisions, following one path or another. A program is usually not limited to a linear sequence of instructions since during its process it may bifurcate, repeat code or bypass sections.

Questions:

1. Explain in briefly control structures?
2. Explain about Loop structure?
3. Explain about operators in JS?
4. Define break and continue statement?

Unit IV Mind Map



Java Script Functions:

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

JavaScript functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

There are mainly two advantages of JavaScript functions.

1. **Code reusability:** We can call a function several times so it save coding.
2. **Less coding:** It makes our program compact. We don't need to write many lines of code each time to perform a common task.

JavaScript Function Syntax

The syntax of declaring function is given below.

1. `function functionName([arg1, arg2, ...argN]){`
2. `//code to be executed`
3. `}`

JavaScript Functions can have 0 or more arguments.

Example

```
function myFunction(p1, p2) {  
  return p1 * p2; // The function returns the product of p1 and p2  
}
```

FunctionDefinitions

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas: **(parameter1, parameter2, ...)**

The code to be executed, by the function, is placed inside curly brackets: {}

```
function name(parameter1, parameter2, parameter3) {
```

```
  // code to be executed
```

```
}
```

Function **parameters** are listed inside the parentheses () in the function definition.

Function **arguments** are the **values** received by the function when it is invoked.

Inside the function, the arguments (the parameters) behave as local variables.

A Function is much the same as a Procedure or a Subroutine, in other programming languages.

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Function Return

When JavaScript reaches a `return` statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

Example

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

```
let x = myFunction(4, 3); // Function is called, return value will end up in x
```

```
function myFunction(a, b) {  
  return a * b; // Function returns the product of a and b  
}
```

The result in x will be:

12

Why Functions?

You can reuse code: Define the code once, and use it many times.

You can use the same code many times with different arguments, to produce different results.

Example

Convert Fahrenheit to Celsius:

```
function toCelsius(fahrenheit) {  
  return (5/9) * (fahrenheit-32);  
}  
document.getElementById("demo").innerHTML = toCelsius(77);
```

The () Operator Invokes the Function

Using the example above, `toCelsius` refers to the function object, and `toCelsius()` refers to the function result.

Accessing a function without `()` will return the function object instead of the function result.

Example

```
function toCelsius(fahrenheit) {  
  return (5/9) * (fahrenheit-32);  
}  
document.getElementById("demo").innerHTML = toCelsius;
```

Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

Example

Instead of using a variable to store the return value of a function:


```
let x = toCelsius(77);
let text = "The temperature is " + x + " Celsius";
```

You can use the function directly, as a variable value:

```
let text = "The temperature is " + toCelsius(77) + " Celsius";
```

Local Variables

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables can only be accessed from within the function.

Example

```
// code here can NOT use carName
```

```
function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}
```

```
// code here can NOT use carName
```

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

JavaScript Function Methods

Method	Description
apply()	It is used to call a function contains this value and a single array of arguments.
bind()	It is used to create a new function.
call()	It is used to call a function contains this value and an argument list.
toString()	It returns the result in a form of a string.

Duration of Identifiers

An identifier is a **sequence of characters in the code that identifies a variable, function, or property**. In JavaScript, identifiers are case-sensitive and can contain Unicode letters, \$, _ , and digits (0-9), but may not start with a digit.

Identifiers can be a combination of letters, numbers, special symbols, etc. But it must not extend **31 characters**. Hence, the maximum possible length of an identifier is 31 characters.

An identifier is a **name that is given to entities like variables, functions, class, etc**. Keywords cannot be used as identifier names. For example, `//invalid const new = 5; // Error!`

The JavaScript syntax defines two types of values:

- Fixed values
- Variable values

Fixed values are called **Literals**.

Variable values are called **Variables**.

JavaScript Literals

The two most important syntax rules for fixed values are:

1. **Numbers** are written with or without decimals:

10.50

1001

2. **Strings** are text, written within double or single quotes:

"John Doe"

'John Doe'

```
// How to create variables:  
var x;  
let y;
```

```
// How to use variables:
```

```
x = 5;  
y = 6;  
let z = x + y;
```

JavaScript Variables

In a programming language, **variables** are used to **store** data values.

JavaScript uses the keywords `var`, `let` and `const` to **declare** variables.

An **equal sign** is used to **assign values** to variables.

In this example, x is defined as a variable. Then, x is assigned (given) the value 6:

```
let x;  
x = 6;
```

JavaScript Operators

JavaScript uses **arithmetic operators** (`+` `-` `*` `/`) to **compute** values:

```
(5 + 6) * 10
```

JavaScript uses an **assignment operator** (`=`) to **assign** values to variables:

```
let x, y;
```

```
x = 5;
```

$$y = 6;$$

JavaScript Expressions

An expression is a combination of values, variables, and operators, which computes to a value. The computation is called an evaluation.

For example, `5 * 10` evaluates to 50:

```
5 * 10
```

Expressions can also contain variable values:

```
x * 10
```

The values can be of various types, such as numbers and strings.

For example, `"John" + " " + "Doe"`, evaluates to `"John Doe"`:

```
"John" + " " + "Doe"
```

JavaScript Keywords

JavaScript **keywords** are used to identify actions to be performed.

The `let` keyword tells the browser to create variables:

```
let x, y;  
x = 5 + 6;  
y = x * 10;
```

The `var` keyword also tells the browser to create variables:

```
var x, y;  
x = 5 + 6;  
y = x * 10;
```

In these examples, using `var` or `let` will produce the same result.

You will learn more about `var` and `let` later in this tutorial.

JavaScript Comments

Not all JavaScript statements are "executed".

Code after double slashes `//` or between `/*` and `*/` is treated as a **comment**.

Comments are ignored, and will not be executed:

```
let x = 5; // I will be executed
```

```
// x = 6; I will NOT be executed
```

JavaScript Identifiers / Names

Identifiers are JavaScript names.

Identifiers are used to name variables and keywords, and functions.

The rules for legal names are the same in most programming languages.

A JavaScript name must begin with:

- A letter (A-Z or a-z)
- A dollar sign (\$)
- Or an underscore (_)

Subsequent characters may be letters, digits, underscores, or dollar signs.

Note

Numbers are not allowed as the first character in names.

This way JavaScript can easily distinguish identifiers from numbers.

JavaScript is Case Sensitive

All JavaScript identifiers are **case sensitive**.

The variables `lastName` and `lastname`, are two different variables:

```
let lastname, lastName;  
lastName = "Doe";  
lastname = "Peterson";
```

JavaScript and Camel Case

Historically, programmers have used different ways of joining multiple words into one variable name:

Hyphens:

first-name, last-name, master-card, inter-city.

Hyphens are not allowed in JavaScript. They are reserved for subtractions.

Underscore:

first_name, last_name, master_card, inter_city.

Upper Camel Case (Pascal Case):

FirstName, LastName, MasterCard, InterCity.

Lower Camel Case:

JavaScript programmers tend to use camel case that starts with a lowercase letter:

firstName, lastName, masterCard, interCity.

JavaScript Character Set

JavaScript uses the **Unicode** character set.

Unicode covers (almost) all the characters, punctuations, and symbols in the world.

Scope Rules

Scope determines the accessibility (visibility) of variables.

JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

Block Scope

Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: `let` and `const`.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a `{ }` block cannot be accessed from outside the block:

Example

```
{  
  let x = 2;  
}  
// x can NOT be used here
```

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a `{ }` block can be accessed from outside the block.

Example

```
{  
  var x = 2;  
}  
// x CAN be used here
```

Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

Example

```
// code here can NOT use carName
```

```
function myFunction() {  
  let carName = "Volvo";  
  // code here CAN use carName  
}
```

```
// code here can NOT use carName
```

Local variables have **Function Scope**:

They can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

Function Scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared with `var`, `let` and `const` are quite similar when declared inside a function.

They all have **Function Scope**:

```
function myFunction() {  
  var carName = "Volvo"; // Function Scope  
}
```

```
function myFunction() {
  let carName = "Volvo"; // Function Scope
}
function myFunction() {
  const carName = "Volvo"; // Function Scope
}
```

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

Example

```
let carName = "Volvo";
// code here can use carName

function myFunction() {
  // code here can also use carName
}
```

A global variable has **Global Scope**:

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

Global variables can be accessed from anywhere in a JavaScript program.

Variables declared with `var`, `let` and `const` are quite similar when declared outside a block. They all have **Global Scope**:

```
var x = 2;    // Global scope
let x = 2;    // Global scope
const x = 2;  // Global scope
```

JavaScript Variables

In JavaScript, objects and functions are also variables.

Scope determines the accessibility of variables, objects, and functions from different parts of the code.

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable `carName`, even if the value is assigned inside a function.

Example

```
myFunction();

// code here can use carName

function myFunction() {
  carName = "Volvo";
}
```

Strict Mode

All modern browsers support running JavaScript in "Strict Mode".
You will learn more about how to use strict mode in a later chapter of this tutorial.

In "Strict Mode", undeclared variables are not automatically global.

Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the `var` keyword belong to the window object:

Example

```
var carName = "Volvo";  
// code here can use window.carName
```

Global variables defined with the `let` keyword do not belong to the window object:

Example

```
let carName = "Volvo";  
// code here can not use window.carName
```

Warning

Do NOT create global variables unless you intend to.

Your global variables (or functions) can overwrite window variables (or functions).
Any function, including the window object, can overwrite your global variables and functions.

The Lifetime of JavaScript Variables

The lifetime of a JavaScript variable starts when it is declared.

Function (local) variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser window (or tab).

Function Arguments

Function arguments (parameters) work as local variables inside functions.

Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Example

Use recursion to add all of the numbers up to 10.

```
publicclassMain{  
  
publicstaticvoidmain(String[]args){  
  
int result =sum(10);  
  
System.out.println(result);  
  
}  
  
publicstaticintsum(int k){if(k  
  
>0){  
  
return k +sum(k -1);  
  
}  
  
}else{  
  
return0  
  
}  
  
}
```

Example Explained

When the `sum()` function is called, it adds parameter `k` to the sum of all numbers smaller than `k` and returns the result. When `k` becomes 0, the function just returns 0. When running, the program follows these steps:

Since the function does not call itself when `k` is 0, the program stops there and returns the result.

Halting Condition

Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter `k` becomes 0.

It is helpful to see a variety of different examples to better understand the concept. In this example, the function adds a range of numbers between a start and an end. The halting condition for this recursive function is when **end** is not greater than **start**:

Example

Use recursion to add all of the numbers between 5 to 10.

```
publicclassMain{
```

```

publicstaticvoidmain(String[]args){
int result =sum(5,10);
System.out.println(result);
}
publicstaticintsum(int start,int
end){if(end > start){
return end +sum(start, end -1);
}else{
return
end:

```

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

Recursion Vs Iteration

Iteration

One of the most essential tools in control flow is the use of iterative statements. These iterative statements typically come in the form of a:

- **for statement**
- **for in statement**
- **for of statement**
- **while statement**
- **do...while statement**

In these iterative statements, **-label statement||**, **-continue statement||**, and **-break statement||** can be used in conjunction to give further control of the loop behavior.

```

// for statement example
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// logs 0, 1, 2, 3, 4 in separate lines

const obj = {
  key1: "foo",
  key2: "bar",
  key3: 9991
};

// for in statement example
for (key in obj) {
  console.log(key);
}
// logs key1, key2, key3 in separate lines

// for of statement example
for (value of obj) {
  console.log(value);
}
// logs foo, bar, 9991 in separate lines

// ===== //
let n = 5;

// while statement example
while (n) {
  console.log(n--);
}
// logs 5, 4, 3, 2, 1 in separate lines

// do... while statement example
do {
  console.log(++n);
} while (n < 5);
// logs 1, 2, 3, 4, 5 in separate lines

// ===== //
// label, continue, and break example
label1:
for (let i = 0; i < 10; i++) {
  if(i % 2 === 0) {
    continue label1;
  } else if(i === 7) {
    break;
  }
  console.log(i);
}
// logs 1, 3, 5 in separate lines

```

Besides the above mentioned iterative statements, there are also iterative array methods, such as:

- **forEach**
- **map**
- **filter**
- **reduce**

What separates these from the previously mentioned, is that these iterative array methods require a callback function. This is the fundamental difference in how these iterative array methods operate as compared to the traditional iterative statements above as we will see when we take a look behind the scenes.

Recursion

Now that we've learned what an iteration is, let's take a look at recursions and how they differ. Recursions describe the behavior of recursive functions, which is to invoke or call itself. A basic comparison of iteration and recursion use for calculating factorial is shown below:

```

// iterative factorial
function iterativeFactorial(n) {
  let res = 1;
  for (let i = n; i > 0; i--) {
    res = res * i;
  }
  return res;
}
console.log(iterativeFactorial(5)); // Logs 120

// ===== //

// recursive factorial
function recursiveFactorial(n) {
  if (n === 1) {
    return 1;
  }
  return n * recursiveFactorial(n - 1);
}
console.log(recursiveFactorial(5)); // Logs 120

// ===== //

// better recursive factorial for Tail Call Optimization(TCO)
function betterRecursiveFactorial(n, res) {
  if (n === 1) {
    return res;
  }
  // return function without any operation allows TCO
  return betterRecursiveFactorial(n - 1, n * res);
}
console.log(betterRecursiveFactorial(5, 1)); // Logs 120

```

Side Note: Tail Call Optimization (TCO) is an optimization carried out by the compiler or engine that allows the “loop” to continue without growing the stack. Even though ES6 came out with TCO as a part of its new standard, all the major browsers have had a bumpy ride implementing it and as of now, it’s been in limbo. That being said, it’s good to keep in mind how to convert one for TCO. Take a look [here](#) for more details regarding its implementation history for JavaScript.

When recursive Factorial is called, the following takes place:

```
5 * recursiveFactorial(5 - 1) // n !== 1 so function is called with new argument and returned
4 * recursiveFactorial(4 - 1) // n !== 1 so function is called with new argument and returned
3 * recursiveFactorial(3 - 1) // n !== 1 so function is called with new argument and returned
2 * recursiveFactorial(2 - 1) // n !== 1 so function is called with new argument and returned
1 // n === 1 so 1 is returned and the backtracking happens
2 * 1 // 2
2 * 3 // 6
4 * 6 // 24
5 * 24 // 120
```

As we can see, besides the initial call to *recursiveFactorial*, it in itself is called an additional four times, and after reaching the **base case** of $n===1$, it backtracks all the way, fulfilling each subsequent computation to reach 120.

Before looking behind the code, let's concretely define the components of a recursive function. There are two essential components that make a recursive function desirably functional: the **recursion** and the **base case**.

The **recursion** is the part where the function is called, which in our factorial example would be *recursiveFactorial(n-1)*. Take note, that the function can be called in multiple places in itself, as well as multiple times in the same expression with likely different arguments. That is technically enough to make a function recursive, but it would be undesirable as it would crash with a stack overflow error.

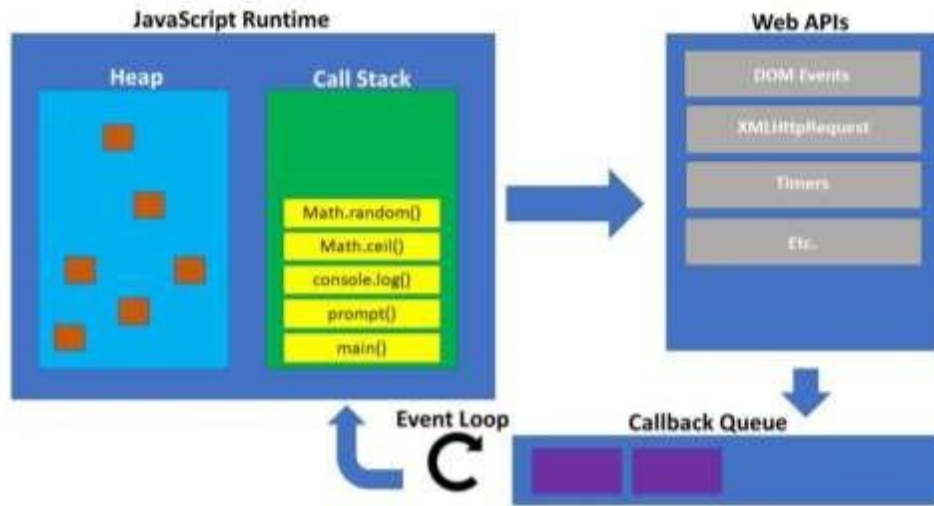
```
// calling this function will crash with the
// stack overflow error since all this function
// will do is keep on calling itself with no
// stop in sight.
function recursiveFactorial(n) {
  return n * recursiveFactorial(n - 1);
}
```

The **base case** is where we define the stopping condition. In our factorial example, the base case is **if (n===1)**. Take note that there can be as many base cases as the algorithm requires.

Going Behind-the-Scenes

First, we need to understand that JavaScript is a single-threaded concurrent programming language. This means that JavaScript does one thing at a time (JavaScript Runtime) and through a cooperative relationship with the Web APIs, callback queue, and event loop allows -multi-tasking in the form of scheduling.

Below shows the different components of JavaScript in action:



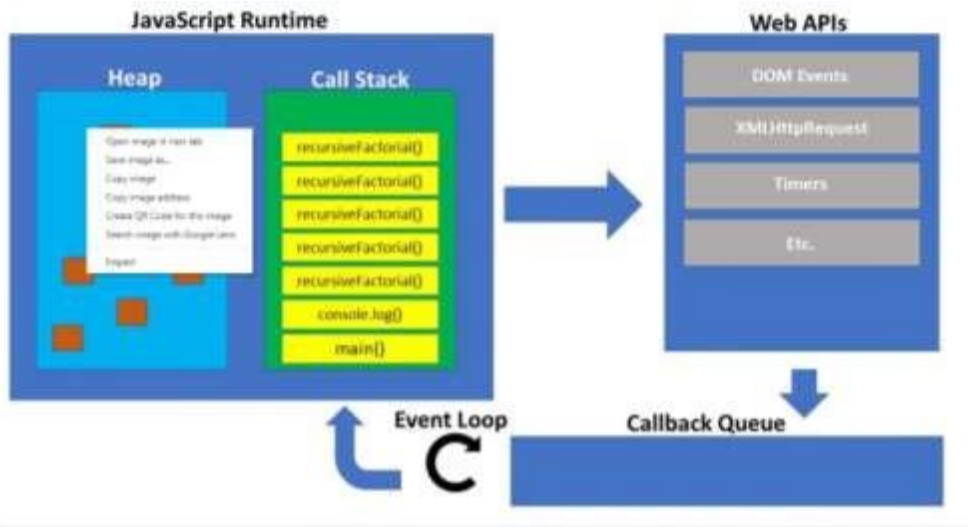
The **JavaScript Runtime** or the JavaScript engine (V8 for Chrome, SpiderMonkey for FireFox) contains the **Heap** and **Call Stack**. The **Heap** is an unstructured area of memory where memory allocation occurs for all the variables and objects. The **Call Stack** is a data structure that follows the Last-In-First-Out (LIFO) system and keeps track of the function calls in stack frames (denoted by the yellow rectangles in the figure above) which contain the function along with its arguments and local variables.

Web APIs are a part of the browser and contains the essential APIs that allows JavaScript to function in a concurrent manner. Examples include DOM events, such as the click and scroll event, AJAX requests, and the `setTimeout` function.

The **Callback Queue** is a data structure that follows the First-In-First-Out (FIFO) system and queues the functions resolved by the Web APIs.

The **Event Loop**'s purpose is to add one queue item from the **Callback Queue** to the **Call Stack** when the Call Stack is *empty*.

Can you see how the Call Stack would change with the recursiveFactorial example?:



With a more complete picture under our belt, let's circle back to iteration and recursion.

In iteration, the looping relies on itself. Little to no change occurs to the Call Stack. In recursion, however, the looping relies on repeatedly calling on itself, which consequently adds a stack frame to the Call Stack for each function call. This also means a great deal of removing and adding takes place, which in turn adds a significant burden in run time for increasing number of calls. When the data set or input is small, the difference between iteration and recursion in terms of time is insignificant, otherwise, iteration often performs better.

In the scenario of a significantly large loop or even an infinite loop in iteration, the browser tab will seem unresponsive to any action taken by the user on the page. This is because the loop taking place in the Call Stack is blocking any item coming from the Callback Queue. That being said, other tabs would work normally since only the process for that one tab is stalled.

In a similar case where a large enough recursion occurs, JavaScript actually crashes due to stack overflow. Each browser has a stack limit which if exceeded would lead to the stack overflow error.

Typically, iteration can be converted to recursion and vice versa. So aside from performance, there is also readability and maintainability to be concerned about when choosing which approach to use. Recursion, due to its algorithmic nature often tends to require a fewer number of lines of code. Also, certain algorithms are more easily understood and intuitive to program through recursion than iteration. In the end, it all depends on the scope of the project, the allocated resources, the platform, and the audience size, among other factors, when choosing the tools and techniques to use.

Global Functions

Global methods are functions that are available to any script as they are not methods of any specific object. You can invoke global methods directly just as you would do with any core JavaScript global functions such as **parseInt()** or **eval()**.

This is a complete list of all the global methods available in HPE Service Manager.

Global method	Description
base64Decode	Converts base 64 string data to its original format.
base64Encode	Converts binary data to a base 64 string format.
compile	Validates the syntax of the specified JavaScript.
doHTTPRequest	Issues an HTTP request to a specified URL.
doSOAPRequest	Issues an SOAP request to a specified URL.
execute	This method performs the specified process or program.
getLog	This method retrieves a logger named according to the value of the name parameter.
help	Displays a brief description of a Service Manager-defined JavaScript object.
makeSCWebURL	Creates a URL query to the Service Manager Web tier.
print	Displays a message in the client Messages view.
Quit	Allows JavaScript to abort processing and return a failure return code.
RCtoString	Converts a Service Manager global return code value into a localized text string.
readFile	Reads data from the local file system.
setAppMessage	Defines the message returned in the "message" attribute in the soap response.
stripHtml	Takes HTML content and strips out the HTML tags and returns the content as text without the HTML tags.
uncompressFile	Expands a .zip file into a specified location.
writeAttachmentToFile	Writes a requested attachment record to the local file system.
writeFile	Writes data to the local file system.
xmlstring	Converts a JavaScript string to an XML string.

Java Script Arrays: **Arrays**

An array is a special variable, which can hold more than one value:

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

Why Use Arrays?

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
let car1 = "Saab";  
let car2 = "Volvo";  
let car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Declaring and Allocating Arrays

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

Syntax:

It is a common practice to declare arrays with the `const` keyword.

Learn more about `const` with arrays in the chapter: [JS Array Const](#).

```
const array_name = [item1, item2, ...];
```

Example

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

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

Example

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

Using the JavaScript Keyword `new`

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

Example

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

The two examples above do exactly the same.

There is no need to use `new Array()`.

For simplicity, readability and execution speed, use the array literal method.

References and Reference Parameters

Accessing Array Elements

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

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

Note: Array indexes start with 0.

[0] is the first element. [1] is the second element.

Changing an Array Element

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

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

Example

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
```

Access the Full Array

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

Example

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

Arrays are Objects

Arrays are a special type of objects. The `typeof` operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

Arrays use **numbers** to access its "elements". In this example, `person[0]` returns John:

Array:

```
const person = ["John", "Doe", 46];
```

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

Object:

```
const person = {firstName:"John", lastName:"Doe", age:46};
```

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.

Because of this, you can have variables of different types in the same Array.

You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

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

Passing Arrays to Functions

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

```
cars.length // Returns the number of elements  
cars.sort() // Sorts the array
```

Array methods are covered in the next chapters.

The length Property

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

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let length = fruits.length;
```

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

Accessing the First Array Element

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits[0];
```

Accessing the Last Array Element

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits[fruits.length - 1];
```

One way to loop through an array, is using a `for` loop:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;
```

```
let text = "<ul>";
for (let i = 0; i < fLen; i++) {
  text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

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

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
let text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
```

```
function myFunction(value) {
  text += "<li>" + value + "</li>";
}
```

Adding Array Elements

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

Example

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

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

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[fruits.length] = "Lemon"; // Adds "Lemon" to fruits
```

WARNING !

Example

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[6] = "Lemon"; // Creates undefined "holes" in fruits
```

Sorting an Array

The `sort()` method sorts an array alphabetically:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
```

Reversing an Array

The `reverse()` method reverses the elements in an array.

You can use it to sort an array in descending order:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
fruits.reverse();
```

Numeric Sort

By default, the `sort()` function sorts values as **strings**.

This works well for strings ("Apple" comes before "Banana").

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the `sort()` method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

Use the same trick to sort an array descending:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
```

The Compare Function

The purpose of the compare function is to define an alternative sort order.

The compare function should return a negative, zero, or positive value, depending on the arguments:

```
function(a, b){return a - b}
```

When the `sort()` function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative a is sorted before b .

If the result is positive b is sorted before a .

If the result is 0 no changes are done with the sort order of the two values.

Example:

The compare function compares all the values in the array, two values at a time (a , b).

When comparing 40 and 100, the `sort()` method calls the compare function(40, 100).

The function calculates $40 - 100$ ($a - b$), and since the result is negative (-60), the sort function will sort 40 as a value lower than 100.

You can use this code snippet to experiment with numerically and alphabetically sorting:

```
<button onclick="myFunction1()">Sort Alphabetically</button>
<button onclick="myFunction2()">Sort Numerically</button>
```

```
<p id="demo"></p>
```

```
<script>
```

```
const points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;
```

```
function myFunction1() {
  points.sort();
  document.getElementById("demo").innerHTML = points;
}
```

```
function myFunction2() {
  points.sort(function(a, b){return a - b});
  document.getElementById("demo").innerHTML = points;
}
</script>
```

Sorting an Array in Random Order

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return 0.5 - Math.random()});
```

The Fisher Yates Method

The above example, `array.sort()`, is not accurate, it will favor some numbers over the others.

The most popular correct method, is called the Fisher Yates shuffle, and was introduced in data science as early as 1938!

In JavaScript the method can be translated to this:

Example

```
const points = [40, 100, 1, 5, 25, 10];

for (let i = points.length - 1; i > 0; i--) {
  let j = Math.floor(Math.random() * i)
  let k = points[i]
  points[i] = points[j]
  points[j] = k
}
```

Find the Highest (or Lowest) Array Value

There are no built-in functions for finding the max or min value in an array.

However, after you have sorted an array, you can use the index to obtain the highest and lowest values.

Sorting ascending:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
// now points[0] contains the lowest value
// and points[points.length-1] contains the highest value
```

Sorting descending:

Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
// now points[0] contains the highest value
// and points[points.length-1] contains the lowest value
```

Sorting a whole array is a very inefficient method if you only want to find the highest (or lowest) value.

Using Math.max() on an Array

You can use `Math.max.apply` to find the highest number in an array:

Example

```
function myArrayMax(arr) {
  return Math.max.apply(null, arr);
}
```

`Math.max.apply(null, [1, 2, 3])` is equivalent to `Math.max(1, 2, 3)`.

Using Math.min() on an Array

You can use `Math.min.apply` to find the lowest number in an array:

Example

```
function myArrayMin(arr) {  
  return Math.min.apply(null, arr);  
}
```

`Math.min.apply(null, [1, 2, 3])` is equivalent to `Math.min(1, 2, 3)`.

My Min / Max JavaScript Methods

The fastest solution is to use a "home made" method.

This function loops through an array comparing each value with the highest value found:

Example (Find Max)

```
function myArrayMax(arr) {  
  let len = arr.length;  
  let max = -Infinity;  
  while (len--) {  
    if (arr[len] > max) {  
      max = arr[len];  
    }  
  }  
  return max;  
}
```

This function loops through an array comparing each value with the lowest value found:

Example (Find Min)

```
function myArrayMin(arr) {  
  let len = arr.length;  
  let min = Infinity;  
  while (len--) {  
    if (arr[len] < min) {  
      min = arr[len];  
    }  
  }  
  return min;  
}
```

Sorting Object Arrays

JavaScript arrays often contain objects:

Example

```
const cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
  {type:"BMW", year:2010}
];
```

Even if objects have properties of different data types, the `sort()` method can be used to sort the array.

The solution is to write a compare function to compare the property values:

Example

```
cars.sort(function(a, b){return a.year - b.year});
```

Comparing string properties is a little more complex:

Example

```
cars.sort(function(a, b){
  let x = a.type.toLowerCase();
  let y = b.type.toLowerCase();
  if (x < y) {return -1;}
  if (x > y) {return 1;}
  return 0;
});
```

Searching Arrays

JavaScript Array `indexOf()`

Examples

Find the first index of "Apple":

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let index = fruits.indexOf("Apple");
```

Start at index 3:

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Apple"];
let index = fruits.indexOf("Apple", 3)
```

Definition and Usage

The `indexOf()` method returns the first index (position) of a specified value.

The `indexOf()` method returns -1 if the value is not found.

The `indexOf()` method starts at a specified index and searches from left to right.

By default the search starts at the first element and ends at the last.

Negative start values counts from the last element (but still searches from right to left).

See Also:

The [lastIndexOf\(\)](#) method

Syntax

```
array.indexOf(item, start)
```

Parameters

Parameter	Description
<i>item</i>	Required. The value to search for.
<i>start</i>	Optional. Where to start the search. Default value is 0. Negative values start the search from the end of the array.

Return Value

Type	Description
A number	The index (position) of the first item found. -1 if the item is not found.

Multiple-Subscripted Arrays

A multiple-subscripted array **can be initialized when it's defined, much like a single-subscripted array**. The values are grouped by row in braces. The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1. elements `b[1][0]` and `b[1][1]`, respectively.

- Arrays with 2 subscripts are often used to represent **tables** of values.
- To identify a particular element in the table, we specify:
 - its **row** (by convention, the 1st subscript), and

- its **column** (by convention, the 2nd subscript). // **Draw a picture**

- Arrays in Java can have more than 2 subscripts.

Can you think of something that might be well represented as an array with 3 subscripts?

- In Java, multiple-subscripted arrays are implemented as arrays of arrays:
 - A 2-dimensional array of `int` is implemented as an array of an array of `int`.
 - **There really are only single-subscripted arrays, but an array's elements can be anything, including arrays.**
 - This simplifies the language.
 - A good illustration of this **uniform** treatment is the inner loop of the `printArray` method: Clearly, element `a[i]` is itself an array; it has a **length** attribute.

Java Script Objects:

In JavaScript, objects are king. If you understand objects, you understand JavaScript.

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the `new` keyword)
- Numbers can be objects (if defined with the `new` keyword)
- Strings can be objects (if defined with the `new` keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

All JavaScript values, except primitives, are objects.

JavaScript Primitives

A **primitive value** is a value that has no properties or methods. A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

Value	Type	Comment
"Hello"	string	"Hello" is always "Hello"
3.14	number	3.14 is always 3.14
true	boolean	true is always true

false	boolean	false is always false
null	null (object)	null is always null
undefined	undefined	undefined is always undefined

if x = 3.14, you can change the value of x.

But you cannot change the value of 3.14.

Objects are Variables

JavaScript variables can contain single values:

Example

```
let person = "John Doe";
```

JavaScript variables can also contain many values.

Objects are variables too. But objects can contain many values.

Object values are written as **name : value** pairs (name and value separated by a colon).

Example

```
let person = { firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

A JavaScript object is a collection of **named values**

It is a common practice to declare objects with the `const` keyword.

Example

```
const person = { firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Object Properties

The named values, in JavaScript objects, are called **properties**.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Objects written as name value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

Object Methods

Methods are **actions** that can be performed on objects.

Object properties can be both primitive values, other objects, and

functions. An **object method** is an object property containing a **function**

definition.

Property	Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

JavaScript objects are containers for named values, called properties and methods.

You will learn more about methods in the next chapter

Creating a JavaScript Object

With JavaScript, you can define and create your own objects.

There are different ways to create new objects:

- Create a single object, using an object literal.
- Create a single object, with the keyword *new*.
- Define an object constructor, and then create objects of the constructed type.
- Create an object using `Object.create()`.

Using an Object Literal

This is the easiest way to create a JavaScript Object.

Using an object literal, you both define and create an object in one statement.

An object literal is a list of name:value pairs (like age:50) inside curly braces {}.

The following example creates a new JavaScript object with four properties:

Example

```
const person = { firstName:"John", lastName:"Doe", age:50, eyeColor:"blue" };
```

Spaces and line breaks are not important. An object definition can span multiple lines:

Example

```
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};
```

This example creates an empty JavaScript object, and then adds 4 properties:

Example

```
const person = {};  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue"
```

Using the JavaScript Keyword new

The following example create a new JavaScript object using `new Object()`, and then adds 4 properties:

Example

```
const person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

The examples above do exactly the same.

But there is no need to use `new Object()`.

For readability, simplicity and execution speed, use the object literal method.

JavaScript Objects are Mutable

Objects are mutable: They are addressed by reference, not by value.

If person is an object, the following statement will not create a copy of person:


```
const x = person; // Will not create a copy of person.
```

The object x is **not a copy** of person. It **is** person. Both x and person are the same object.

Any changes to x will also change person, because x and person are the same object.

Example

```
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50, eyeColor: "blue"  
}
```

```
const x = person;  
x.age = 10; // Will change both x.age and person.age
```

Math Object

The JavaScript Math object allows you to perform mathematical tasks on numbers.

Example

```
Math.PI;
```

The Math Object

Unlike other objects, the Math object has no constructor.

The Math object is static.

All methods and properties can be used without creating a Math object first.

Math Properties (Constants)

The syntax for any Math property is : *Math.property*.

JavaScript provides 8 mathematical constants that can be accessed as Math properties:

Example

```
Math.E // returns Euler's number  
Math.PI // returns PI  
Math.SQRT2 // returns the square root of 2  
Math.SQRT1_2 // returns the square root of 1/2  
Math.LN2 // returns the natural logarithm of 2  
Math.LN10 // returns the natural logarithm of 10  
Math.LOG2E // returns base 2 logarithm of E  
Math.LOG10E // returns base 10 logarithm of E
```

Math Methods

The syntax for Math any methods is : `Math.method(number)`

Number to Integer

There are 4 common methods to round a number to an integer:

<code>Math.round(x)</code>	Returns x rounded to its nearest integer
<code>Math.ceil(x)</code>	Returns x rounded up to its nearest integer
<code>Math.floor(x)</code>	Returns x rounded down to its nearest integer
<code>Math.trunc(x)</code>	Returns the integer part of x (new in ES6)

Math.round()

`Math.round(x)` returns the nearest integer:

Examples

```
Math.round(4.6);
```

```
Math.round(4.5);
```

```
Math.round(4.4);
```

Math.ceil()

`Math.ceil(x)` returns the value of x rounded **up** to its nearest integer:

Example

```
Math.ceil(4.9);
```

```
Math.ceil(4.7);
```

```
Math.ceil(4.4);
```

```
Math.ceil(4.2);
```

```
Math.ceil(-4.2);
```

Math.floor()

`Math.floor(x)` returns the value of x rounded **down** to its nearest integer:

Example

```
Math.floor(4.9);  
Math.floor(4.7);  
Math.floor(4.4);  
Math.floor(4.2);  
Math.floor(-4.2);
```

Math.trunc()

`Math.trunc(x)` returns the integer part of x:

Example

```
Math.trunc(4.9);  
Math.trunc(4.7);  
Math.trunc(4.4);  
Math.trunc(4.2);  
Math.trunc(-4.2);
```

Math.sign()

`Math.sign(x)` returns if x is negative, null or positive:

Example

```
Math.sign(-4);  
Math.sign(0);  
Math.sign(4);
```

Math.pow()

`Math.pow(x, y)` returns the value of x to the power of y:

Example

```
Math.pow(8, 2);
```

Math.sqrt()

`Math.sqrt(x)` returns the square root of x:

Example

```
Math.sqrt(64);
```

Math.abs()

`Math.abs(x)` returns the absolute (positive) value of x:

Example

```
Math.abs(-4.7);
```

Math.sin()

`Math.sin(x)` returns the sine (a value between -1 and 1) of the angle x (given in radians).

If you want to use degrees instead of radians, you have to convert degrees to radians:

Angle in radians = Angle in degrees x PI / 180.

Example

```
Math.sin(90 * Math.PI / 180); // returns 1 (the sine of 90 degrees)
```

Math.cos()

`Math.cos(x)` returns the cosine (a value between -1 and 1) of the angle x (given in radians).

If you want to use degrees instead of radians, you have to convert degrees to radians:

Angle in radians = Angle in degrees x PI / 180.

Example

```
Math.cos(0 * Math.PI / 180); // returns 1 (the cos of 0 degrees)
```

Math.min() and Math.max()

`Math.min()` and `Math.max()` can be used to find the lowest or highest value in a list of arguments:

Example

```
Math.min(0, 150, 30, 20, -8, -200);
```

Example

```
Math.max(0, 150, 30, 20, -8, -200);
```

Math.random()

`Math.random()` returns a random number between 0 (inclusive), and 1 (exclusive):

Example

```
Math.random();
```

The Math.log() Method

`Math.log(x)` returns the natural logarithm of x.

The natural logarithm returns the time needed to reach a certain level of growth:

Examples

```
Math.log(1);  
Math.log(2);  
Math.log(3);
```

Math.E and Math.log() are twins.

How many times must we multiply Math.E to get 10?

```
Math.log(10);
```

The Math.log2() Method

`Math.log2(x)` returns the base 2 logarithm of x.

How many times must we multiply 2 to get 8?

```
Math.log2(8);
```

The Math.log10() Method

`Math.log10(x)` returns the base 10 logarithm of x.

How many times must we multiply 10 to get 1000?

```
Math.log10(1000);
```

JavaScript Math Methods

Method	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
acosh(x)	Returns the hyperbolic arccosine of x
asin(x)	Returns the arcsine of x, in radians
asinh(x)	Returns the hyperbolic arcsine of x
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
atan2(y, x)	Returns the arctangent of the quotient of its arguments
atanh(x)	Returns the hyperbolic arctangent of x

cbrt(x)	Returns the cubic root of x
ceil(x)	Returns x, rounded upwards to the nearest integer
cos(x)	Returns the cosine of x (x is in radians)
cosh(x)	Returns the hyperbolic cosine of x
exp(x)	Returns the value of E ^x
floor(x)	Returns x, rounded downwards to the nearest integer
log(x)	Returns the natural logarithm (base E) of x
max(x, y, z, ..., n)	Returns the number with the highest value
min(x, y, z, ..., n)	Returns the number with the lowest value
pow(x, y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Rounds x to the nearest integer
sign(x)	Returns if x is negative, null or positive (-1, 0, 1)
sin(x)	Returns the sine of x (x is in radians)
sinh(x)	Returns the hyperbolic sine of x
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of an angle
tanh(x)	Returns the hyperbolic tangent of a number
trunc(x)	Returns the integer part of a number (x)

String Object

The **String** object lets you work with a series of characters; it wraps Javascript's string primitive data type with a number of helper methods.

As JavaScript automatically converts between string primitives and String objects, you can call any of the helper methods of the String object on a string primitive.

Syntax

Use the following syntax to create a String object –

```
var val = new String(string);
```

The **String** parameter is a series of characters that has been properly encoded.

String Properties

Here is a list of the properties of String object and their description.

Sr.No.	Property & Description
1	<u>constructor</u> Returns a reference to the String function that created the object.
2	<u>length</u> Returns the length of the string.
3	<u>prototype</u> The prototype property allows you to add properties and methods to an object.

In the following sections, we will have a few examples to demonstrate the usage of String properties.

String Methods

Here is a list of the methods available in String object along with their description.

Sr.No.	Method & Description
1	<u>charAt()</u> Returns the character at the specified index.
2	<u>charCodeAt()</u> Returns a number indicating the Unicode value of the character at the given index.
3	<u>concat()</u> Combines the text of two strings and returns a new string.
4	<u>indexOf()</u> Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
5	<u>lastIndexOf()</u> Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
6	<u>localeCompare()</u> Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.

7	<u>match()</u> Used to match a regular expression against a string.
8	<u>replace()</u> Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
9	<u>search()</u> Executes the search for a match between a regular expression and a specified string.
10	<u>slice()</u> Extracts a section of a string and returns a new string.
11	<u>split()</u> Splits a String object into an array of strings by separating the string into substrings.
12	<u>substr()</u> Returns the characters in a string beginning at the specified location through the specified number of characters.
13	<u>substring()</u> Returns the characters in a string between two indexes into the string.
14	<u>toLocaleLowerCase()</u> The characters within a string are converted to lower case while respecting the current locale.
15	<u>toLocaleUpperCase()</u> The characters within a string are converted to upper case while respecting the current locale.
16	<u>toLowerCase()</u> Returns the calling string value converted to lower case.
17	<u>toString()</u> Returns a string representing the specified object.
18	<u>toUpperCase()</u> Returns the calling string value converted to uppercase.
19	<u>valueOf()</u> Returns the primitive value of the specified object.

String HTML Wrappers

Here is a list of the methods that return a copy of the string wrapped inside an appropriate HTML tag.

Sr.No.	Method & Description
1	<u>anchor()</u> Creates an HTML anchor that is used as a hypertext target.
2	<u>big()</u> Creates a string to be displayed in a big font as if it were in a <big> tag.
3	<u>blink()</u> Creates a string to blink as if it were in a <blink> tag.
4	<u>bold()</u> Creates a string to be displayed as bold as if it were in a tag.
5	<u>fixed()</u> Causes a string to be displayed in fixed-pitch font as if it were in a <tt> tag
6	<u>fontcolor()</u> Causes a string to be displayed in the specified color as if it were in a tag.
7	<u>fontsize()</u> Causes a string to be displayed in the specified font size as if it were in a tag.
8	<u>italics()</u> Causes a string to be italic, as if it were in an <i> tag.
9	<u>link()</u> Creates an HTML hypertext link that requests another URL.
10	<u>small()</u> Causes a string to be displayed in a small font, as if it were in a <small> tag.
11	<u>strike()</u> Causes a string to be displayed as struck-out text, as if it were in a <strike> tag.
12	<u>sub()</u> Causes a string to be displayed as a subscript, as if it were in a <sub> tag
13	<u>sup()</u> Causes a string to be displayed as a superscript, as if it were in a <sup> tag

Date Object

JavaScript **Date Object** lets us work with dates:

Wed Mar 23 2022 14:33:23 GMT+0530 (India Standard Time)

Year: 2022 Month: 3 Day: 23 Hours: 14 Minutes: 33 Seconds: 23

Example

```
const d = new Date();
```

JavaScript Date Output

By default, JavaScript will use the browser's time zone and display a date as a full text string:

Wed Mar 23 2022 14:33:23 GMT+0530 (India Standard Time)

You will learn much more about how to display dates, later in this tutorial.

Creating Date Objects

Date objects are created with the `new Date()` constructor.

There are **4 ways** to create a new date object:

`new Date()`

`new Date(year, month, day, hours, minutes, seconds, milliseconds)`

`new Date(date string)`

`new Date()`

`new Date()` creates a new date object with the **current date and time**:

Example

```
const d = new Date();
```

Date objects are static. The computer time is ticking, but date objects are not.

`new Date(year, month, ...)`

`new Date(year, month, ...)` creates a new date object with a **specified date and time**.

7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

Example

```
const d = new Date(2018, 11, 24, 10, 33, 30, 0);
```

Note: JavaScript counts months from **0** to

11: January = 0.

December = 11.

Specifying a month higher than 11, will not result in an error but add the overflow to the next year:

Specifying:

```
const d = new Date(2018, 15, 24, 10, 33, 30);
```

Is the same as:

```
const d = new Date(2019, 3, 24, 10, 33, 30);
```

Specifying a day higher than max, will not result in an error but add the overflow to the next month:

Specifying:

```
const d = new Date(2018, 5, 35, 10, 33, 30);
```

Is the same as:

```
const d = new Date(2018, 6, 5, 10, 33, 30);
```

Using 6, 4, 3, or 2 Numbers

6 numbers specify year, month, day, hour, minute, second:

Example

```
const d = new Date(2018, 11, 24, 10, 33, 30);
```

5 numbers specify year, month, day, hour, and minute:

Example

```
const d = new Date(2018, 11, 24, 10, 33);
```

4 numbers specify year, month, day, and hour:

Example

```
const d = new Date(2018, 11, 24, 10);
```

3 numbers specify year, month, and day:

Example

```
const d = new Date(2018, 11, 24);
```

2 numbers specify year and month:

Example

```
const d = new Date(2018, 11);
```

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

Example

```
const d = new Date(2018)
```

Previous Century

One and two digit years will be interpreted as 19xx:

Example

```
const d = new Date(99, 11, 24);
```

Example

```
const d = new Date(9, 11, 24);
```

new Date(dateString)

`new Date(dateString)` creates a new date object from a **date string**:

Example

```
const d = new Date("October 13, 2014 11:13:00");
```

JavaScript Stores Dates as Milliseconds

JavaScript stores dates as number of milliseconds since January 01, 1970, 00:00:00 UTC (Universal Time Coordinated).

Zero time is January 01, 1970 00:00:00 UTC.

Now the time is: **1648026203871** milliseconds past January 01, 1970

new Date(milliseconds)

`new Date(milliseconds)` creates a new date object as **zero time plus milliseconds**:

Example

```
const d = new Date(0);
```

01 January 1970 **plus** 100 000 000 000 milliseconds is approximately 03 March 1973:

Example

```
const d = new Date(100000000000);
```

January 01 1970 **minus** 100 000 000 000 milliseconds is approximately October 31 1966:

Example

```
const d = new Date(-100000000000);
```

Example

```
const d = new Date(86400000);
```

One day (24 hours) is 86 400 000 milliseconds.

Date Methods

When a Date object is created, a number of **methods** allow you to operate on it.

Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of date objects, using either local time or UTC (universal, or GMT) time.

Date methods and time zones are covered in the next chapters.

Displaying Dates

JavaScript will (by default) output dates in full text string format:

Example

```
Wed Mar 23 2022 14:33:23 GMT+0530 (India Standard Time)
```

When you display a date object in HTML, it is automatically converted to a string, with the `toString()` method.

Example

```
const d = new Date();  
d.toString();
```

The `toUTCString()` method converts a date to a UTC string (a date display standard).

Example

```
const d = new Date();  
d.toUTCString();
```

The `toDateDateString()` method converts a date to a more readable format:

Example

```
const d = new Date();  
d.toDateString();
```

The `toISOString()` method converts a Date object to a string, using the ISO standard format:

Example

```
const d = new Date();  
d.toISOString();
```

Boolean and Number Object

The **Boolean** object represents two values, either "true" or "false". If *value* parameter is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false.

Syntax

Use the following syntax to create a **boolean** object.

```
var val = new Boolean(value);
```

Boolean Properties

Here is a list of the properties of Boolean object –

Sr.No.	Property & Description
1	<u>constructor</u> Returns a reference to the Boolean function that created the object.
2	<u>prototype</u> The prototype property allows you to add properties and methods to an object.

In the following sections, we will have a few examples to illustrate the properties of Boolean object.

Boolean Methods

Here is a list of the methods of Boolean object and their description.

Sr.No.	Method & Description
1	<u>toSource()</u> Returns a string containing the source of the Boolean object; you can use this string to create an equivalent object.
2	<u>toString()</u> Returns a string of either "true" or "false" depending upon the value of the object.
3	<u>valueOf()</u> Returns the primitive value of the Boolean object.

In the following sections, we will have a few examples to demonstrate the usage of the Boolean methods.

Document Object

Every web page resides inside a browser window which can be considered as an object.

A Document object represents the HTML document that is displayed in that window. The Document object has various properties that refer to other objects which allow access to and modification of document content.

The way a document content is accessed and modified is called the **Document Object Model**, or **DOM**. The Objects are organized in a hierarchy. This hierarchical structure applies to the organization of objects in a Web document.

- **Window object** – Top of the hierarchy. It is the outmost element of the object hierarchy.
- **Document object** – Each HTML document that gets loaded into a window becomes a document object. The document contains the contents of the page.
- **Form object** – Everything enclosed in the <form>...</form> tags sets the form object.

- **Form control elements** – The form object contains all the elements defined for that object such as text fields, buttons, radio buttons, and checkboxes.

Window Object.

Window Object

The window object represents an open window in a browser.

If a document contain frames (<iframe> tags), the browser creates one window object for the HTML document, and one additional window object for each frame.

Window Object Properties

Property	Description
closed	Returns a boolean true if a window is closed.
console	Returns the Console Object for the window. See also The Console Object .
defaultStatus	Deprecated.
document	Returns the Document object for the window. See also The Document Object .
frameElement	Returns the frame in which the window runs.
frames	Returns all window objects running in the window.
history	Returns the History object for the window. See also The History Object .
innerHeight	Returns the height of the window's content area (viewport) including scrollbars
innerWidth	Returns the width of a window's content area (viewport) including scrollbars
length	Returns the number of <iframe> elements in the current window
localStorage	Allows to save key/value pairs in a web browser. Stores the data with no expiration date
location	Returns the Location object for the window. See also the The Location Object .
name	Sets or returns the name of a window

<u>navigator</u>	Returns the Navigator object for the window. See also <u>The Navigator object</u> .
<u>opener</u>	Returns a reference to the window that created the window
<u>outerHeight</u>	Returns the height of the browser window, including toolbars/scrollbars
<u>outerWidth</u>	Returns the width of the browser window, including toolbars/scrollbars
<u>pageXOffset</u>	Returns the pixels the current document has been scrolled (horizontally) from the upper left corner of the window
<u>pageYOffset</u>	Returns the pixels the current document has been scrolled (vertically) from the upper left corner of the window
<u>parent</u>	Returns the parent window of the current window
<u>screen</u>	Returns the Screen object for the window See also <u>The Screen object</u>
<u>screenLeft</u>	Returns the horizontal coordinate of the window relative to the screen
<u>screenTop</u>	Returns the vertical coordinate of the window relative to the screen
<u>screenX</u>	Returns the horizontal coordinate of the window relative to the screen
<u>screenY</u>	Returns the vertical coordinate of the window relative to the screen
<u>sessionStorage</u>	Allows to save key/value pairs in a web browser. Stores the data for one session
<u>scrollX</u>	An alias of <u>pageXOffset</u>
<u>scrollY</u>	An alias of <u>pageYOffset</u>
<u>self</u>	Returns the current window
<u>status</u>	Deprecated. Avoid using it.
<u>top</u>	Returns the topmost browser window

Summary:

- Use the `function` keyword to declare a function.
- Use the `functionName()` to call a function.
- All functions implicitly return undefined if they don't explicitly return a value.
- Use the `return` statement to return a value from a function explicitly.
- The `arguments` variable is an array-like object inside a function, representing function arguments.
- The function hoisting allows you to call a function before declaring it.

JavaScript is a **multi-paradigm, dynamic language with types and operators, standard built-in objects, and methods**. Its syntax is based on the Java and C languages — many structures from those languages apply to JavaScript as well.

Questions:

1. Defined functions?
2. Explain about scope rules?
3. Explain about recursion?
4. Differentiate recursion vs iteration?
5. Explain global functions.
6. Define Arrays?
7. Write short notes on declaring and allocating arrays.
8. Differentiate between references and reference parameters?
9. Define sorting arrays?
10. What is searching arrays?
11. Explain multiple-subscripted arrays?
12. Explain in detail about Java Script Objects?

Unit V



Document Object Model (DOM):

The XML Document Object Model (DOM) class is **an in-memory representation of an XML document**. The DOM allows you to programmatically read, manipulate, and modify an XML document. The Xml Reader class also reads XML; however, it provides non-cached, forward-only, read-only access.

Modeling a document

What the Document Object Model is

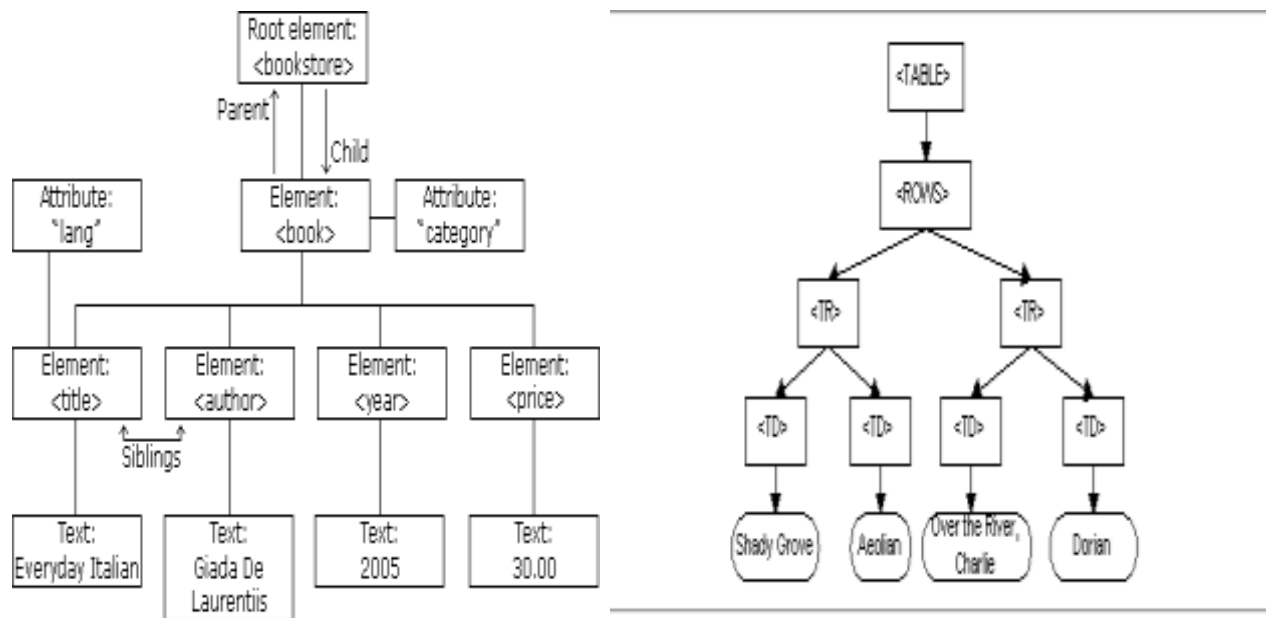
The Document Object Model is a programming API for documents. The object model itself closely resembles the structure of the documents it models. For instance, consider this table, taken from an HTML document:

```

<TABLE>
<ROWS>
<TR>
<TD>Shady Grove</TD>
<TD>Aeolian</TD>
</TR>
<TR>
<TD>Over the River, Charlie</TD>
<TD>Dorian</TD>
</TR>
</ROWS>
</TABLE>

```

The Document Object Model represents this table like this:



DOM representation of the example table

Traversing and modifying a DOM Tree

The DOM gives you access to the elements of a document, allowing you to modify the contents of a page dynamically using event-driven JavaScript. **This section introduces properties and methods of all DOM nodes that enable you to traverse the DOM tree, modify nodes and create or delete content dynamically.**

The Document Object Model (DOM) is a **standard convention for accessing and manipulating elements within HTML and XML documents.** Elements in the DOM are organized

into a tree-like data structure that can be traversed to navigate, locate, or modify elements and/or content within an XML/HTML document.

How does DOM tree work?

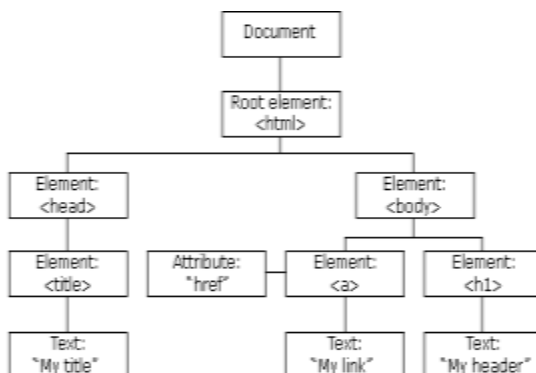
The DOM is a tree of elements, with the Document node at the root, which points to the html Element node, which in turn points to its child element nodes head and body, and so on. From each of those elements, you can navigate the DOM structure and move to different nodes.

With the HTML DOM, you can navigate the node tree using node relationships.

DOM Nodes

According to the W3C HTML DOM standard, everything in an HTML document is a node:

- The entire document is a document node
- Every HTML element is an element node
- The text inside HTML elements are text nodes
- Every HTML attribute is an attribute node (deprecated)
- All comments are comment nodes



With the HTML DOM, all nodes in the node tree can be accessed by JavaScript.

New nodes can be created, and all nodes can be modified or deleted.

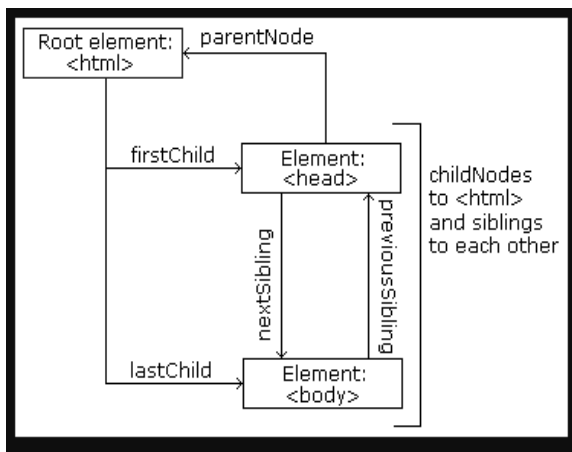
Node Relationships

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

- In a node tree, the top node is called the root (or root node)
- Every node has exactly one parent, except the root (which has no parent)
- A node can have a number of children
- Siblings (brothers or sisters) are nodes with the same parent

```
<html>
  <head>
    <title>DOM Tutorial</title>
  </head>
  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>
</html>
```



From the HTML above you can read:

- `<html>` is the root node
- `<html>` has no parents
- `<html>` is the parent of `<head>` and `<body>`
- `<head>` is the first child of `<html>`
- `<body>` is the last child of `<html>` and:
 - `<head>` has one child: `<title>`
 - `<title>` has one child (a text node): "DOM Tutorial"
 - `<body>` has two children: `<h1>` and `<p>`
 - `<h1>` has one child: "DOM Lesson one"
 - `<p>` has one child: "Hello world!"
 - `<h1>` and `<p>` are siblings

Navigating Between Nodes

You can use the following node properties to navigate between nodes with JavaScript:

- `parentNode`
- `childNodes[nodenum]`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`

Child Nodes and Node Values

A common error in DOM processing is to expect an element node to contain text.

Example:

```
<title id="demo">DOM Tutorial</title>
```

The element node `<title>` (in the example above) does **not** contain text.

It contains a **text node** with the value "DOM Tutorial".

The value of the text node can be accessed by the node's `innerHTML` property:

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

Accessing the `innerHTML` property is the same as accessing the `nodeValue` of the first child:


```
myTitle = document.getElementById("demo").firstChild.nodeValue;
```

Accessing the first child can also be done like this:

```
myTitle = document.getElementById("demo").childNodes[0].nodeValue;
```

All the (3) following examples retrieves the text of an `<h1>` element and copies it into a `<p>` element:

Example

```
<html>
<body>
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").innerHTML;
</script></body></html>
```

Example

```
<html>
<body>
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").firstChild.nodeValue;
</script></body></html>
```

Example

```
<html>
<body>
<h1 id="id01">My First Page</h1>
<p id="id02">Hello!</p>
<script>
document.getElementById("id02").innerHTML =
```

```
document.getElementById("id01").childNodes[0].nodeValue;
```

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

InnerHTML

In this tutorial we use the innerHTML property to retrieve the content of an HTML element.

However, learning the other methods above is useful for understanding the tree structure and the navigation of the DOM.

DOM Root Nodes

There are two special properties that allow access to the full document:

- `document.body` - The body of the document
- `document.documentElement` - The full document

Example

```
<html>
```

```
<body>
```

```
<h2>JavaScript HTMLDOM</h2>
```

```
<p>Displaying document.body</p>
```

```
<p id="demo"></p>
```

```
<script>
```

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

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

Example

```
<html>
```

```
<body>
```

```
<h2>JavaScript HTMLDOM</h2>
```

```
<p>Displaying document.documentElement</p>
```

```
<p id="demo"></p>
```

```
<script>
```

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

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

The nodeName Property

The `nodeName` property specifies the name of a node.

- `nodeName` is read-only
- `nodeName` of an element node is the same as the tag name
- `nodeName` of an attribute node is the attribute name
- `nodeName` of a text node is always `#text`
- `nodeName` of the document node is always `#document`

Example

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").nodeName;
</script>
```

Note: `nodeName` always contains the uppercase tag name of an HTML element.

The nodeValue Property

The `nodeValue` property specifies the value of a node.

- `nodeValue` for element nodes is `null`
- `nodeValue` for text nodes is the text itself
- `nodeValue` for attribute nodes is the attribute value

The.nodeType Property

The `nodeType` property is read only. It returns the type of a node.

Example

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
<script>
document.getElementById("id02").innerHTML = document.getElementById("id01").nodeType;
</script>
```

The most important `nodeType` properties are:

Node	Type	Example
------	------	---------

ELEMENT_NODE	1	<h1 class="heading">W3Schools</h1>
ATTRIBUTE_NODE	2	class = "heading" (deprecated)
TEXT_NODE	3	W3Schools
COMMENT_NODE	8	<!-- This is a comment -->
DOCUMENT_NODE	9	The HTML document itself (the parent of <html>)
DOCUMENT_TYPE_NODE	10	<!Doctype html>

Type 2 is deprecated in the HTML DOM (but works). It is not deprecated in the XML DOM.

DOM collections and Dynamic styles

DOM collections are **accessed as properties of DOM objects such as the document object or a DOM node**. The document object has properties containing the images collection, links collection, forms collection and anchors collection. These collections contain all the elements of the corresponding type on the page.

What is DOM dynamic?

D/s is first and foremost an **energy dynamic** that flows between two people. One person, the Dom, takes on more the role of leader, guide, enforcer, protector and/or daddy, while the other person, the sub, assumes more the role of pleaser, brat, tester, baby girl, and/or servant.

An element's style can be changed dynamically. Often such a change is made in response to user events, which we discuss in Chapter 11. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations.

Definition and Usage. The style property returns a CSSStyleDeclaration object, which represents an element's style attribute. The style property is used to get or set a specific style of an element using different CSS properties.

The HTMLCollection Object

The `getElementsByTagName()` method returns an `HTMLCollection` object.

An `HTMLCollection` object is an array-like list (collection) of HTML elements.

The following code selects all `<p>` elements in a document:

```
const myCollection = document.getElementsByTagName("p");
```

Example

The elements in the collection can be accessed by an index number.

To access the second <p> element you can write:

```
myCollection[1]
```

Note: The index starts at 0.

HTML HTMLCollection Length

The `length` property defines the number of elements in an `HTMLCollection`:

Example

```
myCollection.length
```

The `length` property is useful when you want to loop through the elements in a collection:

Example

Change the text color of all <p> elements:

```
const myCollection = document.getElementsByTagName("p");
for (let i = 0; i < myCollection.length; i++) {
  myCollection[i].style.color = "red";
}
```

An HTMLCollection is NOT an array!

An HTMLCollection may look like an array, but it is not.

You can loop through the list and refer to the elements with a number (just like an array).

However, you cannot use array methods like `valueOf()`, `pop()`, `push()`, or `join()` on an HTMLCollection.

Javascript DOM Create Dynamic Styles

CSS styles are included in HTML pages using one of two elements.

The <link> element is used to include CSS from an external file, whereas the <style> element is used to specify inline styles.

The dynamic styles don't exist on the page when it is loaded initially and they are added after the page has been loaded.

Link Element

Consider this typical <link> element:

Copy

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

This element can just as easily be created using the following DOM code:

Copy

```
let link = document.createElement("link");
link.rel = "stylesheet";

link.type = "text/css";
link.href = "styles.css";

let head = document.getElementsByTagName("head")[0];
```

code works in all major browsers without any issue.

Note that `<link>` elements should be added to the `<head>` instead of the body for this to work properly in all browsers.

The technique can be generalized into the following function:

```
function loadStyles(url){
let link = document.createElement("link");
link.rel = "stylesheet";

link.type = "text/css";
link.href = url;

let head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

Copy

The `loadStyles()` function can then be called like this:

Copy

```
loadStyles("styles.css");
```

Loading styles via an external file is asynchronous, so the styles will load out of order with the JavaScript code being executed.

Typically, it's not necessary to know when the styles have been fully loaded.

Inline

The other way to define styles is using the `<style>` element and including inline CSS, such as this:

```
<style type="text/css">
body {
  background-color: red;
}
</style>
```

Copy

The following DOM code should work:

```
let style =
document.createElement("style");style.type
= "text/css";

style.appendChild(document.createTextNode("body{background-color:red}"));

let head = document.getElementsByTagName("head")[0];
```

Copy

This code works in Firefox, Safari, Chrome, and Opera but not in Internet Explorer.

The workaround for Internet Explorer is to access the element's *styleSheet* property, which in turn has a property called *cssText* that may be set to CSS code as this code sample shows:

Copy

```
let style =
document.createElement("style");style.type
= "text/css";

try{
style.appendChild(document.createTextNode("body{background-color:red}"));
} catch (ex){
style.styleSheet.cssText = "body{background-color:red}";
```



```
head.appendChild(style);
```

This new code uses a try-catch statement to catch the error that Internet Explorer throws and then responds by using the Internet Explorer-specific way of setting styles.

The generic solution is as follows:

Copy

```
function loadStyleString(css){  
  let style =  
    document.createElement("style");  
  style.type = "text/css";  
  
  try{  
    style.appendChild(document.createTextNode(css));  
  } catch (ex){  
    style.styleSheet.cssText = css;  
  }  
  
  let head = document.getElementsByTagName("head")[0];  
  head.appendChild(style);  
}
```

The function can be called as follows:

```
loadStyleString("body{background-color:red}");
```

Copy

Styles specified in this way are added to the page instantly, so changes should be seen immediately.

Events: What are events?

Events are actions that happen when a user interacts with the page - like clicking an element, typing in a field, or loading a page.

Registering Event Handlers

The browser notifies the system that something has happened, and that it needs to be handled. It gets handled by registering a function, called an `event handler`, that listens for a particular type of event.

What does it mean to "handle an event"?

To put it in simple terms, consider this - let's assume you are interested in attending Web Development meetup events in your local community.

To do this, you sign-up for a local meetup called "Women Who Code" and subscribe to notifications. This way, anytime a new meetup is scheduled, you get alerted. That is event handling!

The "event" here is a new JS meetup. When a new meetup is posted, the website meetup.com catches this change, thereby "handling" this event. It then notifies you, thus taking an "action" on the event.

In a browser, events are handled similarly. The browser detects a change, and alerts a function (event handler) that is listening to a particular event. These functions then perform the actions as desired.

Let's look at an example of a `click` event handler:

```
<div class="buttons">
<button>Press 1</button>
<button>Press 2</button>
<button>Press 3</button>
</div>

const buttonContainer = document.querySelector('.buttons');
console.log('buttonContainer', buttonContainer);
buttonContainer.addEventListener('click', event => {
console.log(event.target.value)
})
```

What are the different types of events?

An event can be triggered any time a user interacts with the page. These events could be a user scrolling through the page, clicking on an item, or loading a page.

Here are some common events-

```
onclick dblclick mousedown mouseup mousemove keydown keyup touchmove touchstart t
ouchend onload onfocus onblur onerror onscroll
```

Different phases of events - capture, target, bubble

When an event moves through the DOM - whether bubbling up or trickling down - it is called event propagation. The event propagates through the DOM tree.

Events happen in two phases: the bubbling phase and the capturing phase.

In capture phase, also called the trickling phase, the event "trickles down" to the element that caused the event.

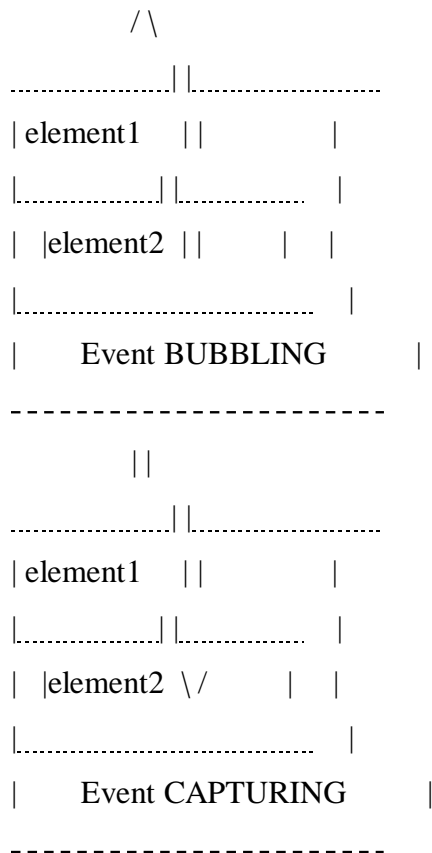
It starts from the root level element and handler, and then propagates down to the element. The capture phase is completed when the event reaches the `target`.

In the bubble phase, the event is "bubbled" up to the DOM tree. It is first captured and handled by the innermost handler (the one that is closest to the element on which the event occurred). It then bubbles up (or propagates up) to the higher levels of DOM tree, further up to its parents, and then finally to its root.

Here's a trick to help you remember this:

trickle down, bubble up

Here's an infographic from [quirksmode](#) that explains this very well:



One thing to note is that, whether you register an event handler in either phase, both phases ALWAYS happen. All events bubble by default.

You can register event handlers for either phase, bubbling or capturing, by using the function `addEventListener(type, listener, useCapture)`. If `useCapture` is set to `false`, the event handler is in the bubbling phase. Otherwise it's in the capture phase.

The order of the phases of the event depends on the browser.

To check which browser honors capture first, you can try the following code in JSfiddle:<div id="child-one">

```
<h1>
```

```
  Child One
```

```
</h1>
```

```
</div>
```

```
const childOne = document.getElementById("child-one");
```

```
const childOneHandler={()=>{
```

```
  console.log('Captured on child one')
```

```
}
```

```
const childOneHandlerCatch={()=>{
```

```
  console.log('Captured on child one in capture phase')
```

```
}
```

```
childOne.addEventListener("click", childOneHandler);
```

```
childOne.addEventListener("click", childOneHandlerCatch, true);
```

When an [event](#) occurs, you can create an event handler which is a piece of code that will execute to respond to that event. An event handler is also known as an event listener. It listens to the event and responds accordingly to the event fires.

An event listener is a [function](#) with an explicit name if it is reusable or an anonymous function in case it is used one time.

An event can be handled by one or multiple event handlers. If an event has multiple event handlers, all the event handlers will be executed when the event is fired.

There are three ways to assign event handlers.

1) HTML event handler attributes

Event handlers typically have names that begin with `on`, for example, the event handler for the `click` event is `onclick`.

To assign an event handler to an event associated with an HTML element, you can use an HTML attribute with the name of the event handler. For example, to execute some code when a button is clicked, you use the following:

```
<input type="button" value="Save" onclick="alert('Clicked!')">
```

Code language: HTML, XML(xml)

In this case, when the button is clicked, the [alert](#) box is shown.

When you assign JavaScript code as the value of the `onclick` attribute, you need to escape the HTML characters such as ampersand (&), double quotes ("), less than (<), etc., or you will get a syntax error.

An event handler defined in the HTML can call a function defined in a script. For

Example:

```
<script>
```

```
function showAlert(){
```

```
    alert('Clicked!');
```

```
}
```

```
</script>
```

```
<input type="button" value="Save" onclick="showAlert()">
```

Code language: HTML, XML(xml)

In this example, the button calls the `showAlert()` function when it is clicked.

The `showAlert()` is a function defined in a separate `<script>` element, and could be placed in an external JavaScript file.

Important notes

The following are some important points when you use the event handlers as attributes of the HTML element:

First, the code in the event handler can access the `event` object without explicitly defining it:

```
<input type="button" value="Save" onclick="alert(event.type)">
```

Code language: HTML, XML(xml)

Second, the `this` value inside the event handler is equivalent to the event's target element:

```
<input type="button" value="Save" onclick="alert(this.value)">
```

Code language: HTML, XML(xml)

Third, the event handler can access the element's properties, for example:

```
<input type="button" value="Save" onclick="alert(value)">
```

Code language: HTML, XML(xml)

Disadvantages of using HTML event handler attributes

Assigning event handlers using HTML event handler attributes are considered as bad practices and should be avoided as much as possible because of the following reasons:

First, the event handler code is mixed with the HTML code, which will make the code more difficult to maintain and extend.

Second, it is a timing issue. If the element is loaded fully before the JavaScript code, users can start interacting with the element on the webpage which will cause an error.

For example, suppose that the following `showAlert()` function is defined in an external JavaScript file:

```
<input type="button" value="Save" onclick="showAlert()">
```

Code language: HTML, XML(xml)

And when the page is loaded fully and the JavaScript has not been loaded, the `showAlert()` function is undefined. If users click the button at this moment, an error will occur.

2) DOM Level 0 event handlers

Each element has event handler properties such as `onclick`. To assign an event handler, you set the property to a function as shown in the example:

```
let btn = document.querySelector('#btn');  
btn.onclick = function() {  
  alert('Clicked!');  
};
```

Code language: JavaScript(javascript)

In this case, the anonymous function becomes the method of the `button` element. Therefore, the `this` value is equivalent to the element. And you can access the element's properties inside the event handler:

```
let btn = document.querySelector('#btn');  
btn.onclick = function() {  
  alert(this.id);  
};
```

Code language: JavaScript(javascript)

Output:

```
btn
```

By using the `this` value inside the event handler, you can access the element's properties and methods.

To remove the event handler, you set the value of the event handler property to `null`:

```
btn.onclick = null;
```

Code language: JavaScript(javascript)

The DOM Level 0 event handlers are still being used widely because of its simplicity and cross-browser support.

3) DOM Level 2 event handlers

DOM Level 2 Event Handlers provide two main methods for dealing with the registering/deregistering event listeners:

- `addEventListener()` – register an event handler
- `removeEventListener()` – remove an event handler

These methods are available in all DOM nodes.

The `addEventListener()` method

The `addEventListener()` method accepts three arguments: an event name, an event handler function, and a Boolean value that instructs the method to call the event handler during the capture phase (`true`) or during the bubble phase (`false`). For example:

```
let btn = document.querySelector('#btn');
btn.addEventListener('click',function(event) {
    alert(event.type); // click
});
```

Code language: JavaScript(javascript)

It is possible to add multiple event handlers to handle a single event, like this:

```
let btn = document.querySelector('#btn');
btn.addEventListener('click',function(event) {
    alert(event.type); // click
});
btn.addEventListener('click',function(event) {
    alert('Clicked!');
});
```

Code language: JavaScript(javascript)

The `removeEventListener()` method

The `removeEventListener()` removes an event listener that was added via the `addEventListener()`. However, you need to pass the same arguments as were passed to the `addEventListener()`. Forexample:

```
let btn = document.querySelector('#btn');  
// add the event listener  
  
let showAlert = function() {  
  alert('Clicked!');  
};  
btn.addEventListener('click', showAlert);  
// remove the event listener  
  
btn.removeEventListener('click', showAlert);
```

Code language: JavaScript(javascript)

Using an anonymous event listener as the following will not work:

```
let btn = document.querySelector('#btn');  
btn.addEventListener('click',function() {  
  alert('Clicked!');  
});  
// won't work  
  
btn.removeEventListener('click', function() {  
  alert('Clicked!');  
});
```

Code language: JavaScript(javascript)

onload

The `onload` event occurs when an object has been loaded.

`onload` is most often used within the `<body>` element to execute a script once a web page has completely loaded all content (including images, script files, CSS files, etc.).

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

The `onload` event can also be used to deal with cookies (see "More Examples" below).

Example

Execute a JavaScript immediately after a page has been loaded:


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

Syntax

In HTML:

```
<element onload="myScript">
```

In JavaScript:

```
object.onload = function(){myScript};
```

In JavaScript:

```
object.onload = function(){myScript};
```

Note: The [addEventListener\(\)](#) method is not supported in Internet Explorer 8 and earlier versions.

Technical Details

Bubbles:	No
Cancelable:	No
Event type:	UiEvent if generated from a user interface, Event otherwise.
Supported HTML tags:	<body>, <frame>, <iframe>, , <input type="image">, <link>, <script>, <style>
DOM Version:	Level 2 Events

Example

Using onload on an element. Alert "Image is loaded" immediately after an image has been loaded:

```

```

```
<script>  
function loadImage() {  
    alert("Image is loaded");  
}  
</script>
```

Example

Using the onload event to deal with cookies:

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

```
<script>  
function checkCookies() {
```

```

var text = "";

if (navigator.cookieEnabled == true) {
  text = "Cookies are enabled.";
} else {
  text = "Cookies are not enabled.";
}
document.getElementById("demo").innerHTML = text;
}
</script>

```

Onmousemove

The onmousemove event occurs when the pointer is moving while it is over an element.

Example

Execute a JavaScript when moving the mouse pointer over a <div> element:

```
<div onmousemove="myFunction()">Move the cursor over me</div>
```

Syntax

In HTML:

```
<element onmousemove="myScript">
```

In JavaScript:

```
object.onmousemove = function(){myScript};
```

In JavaScript, using the addEventListener() method:

```
object.addEventListener("mousemove",
myScript);
```

Technical Details

Bubbles:	Yes
Cancelable:	Yes
Event type:	MouseEvent
Supported HTML tags:	All HTML elements, EXCEPT: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style>, and <title>
DOM Version:	Level 2 Events

Example

This example demonstrates the difference between the onmousemove, onmouseenter and mouseover events:

```
<div onmousemove="myMoveFunction()">  
  <p id="demo">I will demonstrate onmousemove!</p>  
</div>
```

```
<div onmouseenter="myEnterFunction()">  
  <p id="demo2">I will demonstrate onmouseenter!</p>  
</div>
```

```
<div onmouseover="myOverFunction()">  
  <p id="demo3">I will demonstrate onmouseover!</p>  
</div>
```

The event Object and this

Event Object

All event objects in the DOM are based on the Event Object.

Therefore, all other event objects (like [MouseEvent](#) and [KeyboardEvent](#)) has access to the Event Object's properties and methods.

Event Properties and Methods

Property/Method	Description
bubbles	Returns whether or not a specific event is a bubbling event
cancelBubble	Sets or returns whether the event should propagate up the hierarchy or not
cancelable	Returns whether or not an event can have its default action prevented
composed	Returns whether the event is composed or not
createEvent()	Creates a new event
composedPath()	Returns the event's path
currentTarget	Returns the element whose event listeners triggered the event
defaultPrevented	Returns whether or not the preventDefault() method was called for the event
eventPhase	Returns which phase of the event flow is currently being evaluated

isTrusted	Returns whether or not an event is trusted
preventDefault()	Cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur
stopImmediatePropagation()	Prevents other listeners of the same event from being called
stopPropagation()	Prevents further propagation of an event during event flow
target	Returns the element that triggered the event
timeStamp	Returns the time (in milliseconds relative to the epoch) at which the event was created
type	Returns the name of the event

Event Types

These event types belongs to the Event Object:

Event	Description
abort	The event occurs when the loading of a media is aborted
afterprint	The event occurs when a page has started printing
beforeprint	The event occurs when a page is about to be printed
beforeunload	The event occurs before the document is about to be unloaded
canplay	The event occurs when the browser can start playing the media (when it has buffered enough to begin)
canplaythrough	The event occurs when the browser can play through the media without stopping for buffering
change	The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>)
error	The event occurs when an error occurs while loading an external file
fullscreenchange	The event occurs when an element is displayed in fullscreen mode
fullscreenerror	The event occurs when an element can not be displayed in fullscreen mode
input	The event occurs when an element gets user input

invalid	The event occurs when an element is invalid
load	The event occurs when an object has loaded
loadeddata	The event occurs when media data is loaded
loadedmetadata	The event occurs when meta data (like dimensions and duration) are loaded
message	The event occurs when a message is received through the event source
offline	The event occurs when the browser starts to work offline
online	The event occurs when the browser starts to work online
open	The event occurs when a connection with the event source is opened
pause	The event occurs when the media is paused either by the user or programmatically
play	The event occurs when the media has been started or is no longer paused
playing	The event occurs when the media is playing after having been paused or stopped for buffering
progress	The event occurs when the browser is in the process of getting the media data (downloading the media)
ratechange	The event occurs when the playing speed of the media is changed
resize	The event occurs when the document view is resized
reset	The event occurs when a form is reset
scroll	The event occurs when an element's scrollbar is being scrolled
search	The event occurs when the user writes something in a search field (for <code><input="search"></code>)
seeked	The event occurs when the user is finished moving/skipping to a new position in the media
seeking	The event occurs when the user starts moving/skipping to a new position in the media
select	The event occurs after the user selects some text (for <code><input></code> and <code><textarea></code>)

show	The event occurs when a <menu> element is shown as a context menu
stalled	The event occurs when the browser is trying to get media data, but data is not available
submit	The event occurs when a form is submitted
suspend	The event occurs when the browser is intentionally not getting media data
timeupdate	The event occurs when the playing position has changed (like when the user fast forwards to a different point in the media)
toggle	The event occurs when the user opens or closes the <details> element
unload	The event occurs once a page has unloaded (for <body>)
waiting	The event occurs when the media has paused but is expected to resume (like when the media pauses to buffer more data)

on mouseover and on mouseout

The onmouseover event occurs when the mouse pointer is moved onto an element, or onto one of its children. Tip: **This event is often used together with the onmouseout event, which occurs when a user moves the mouse pointer out of an element.** The onmouseout event occurs when the mouse pointer is moved out of an element, or out of one of its children.

Tip: This event is often used together with the [onmouseover](#) event, which occurs when the pointer is moved onto an element, or onto one of its children.

Example

Execute a JavaScript when moving the mouse pointer out of an image:

```

```

Syntax

In HTML:

```
<element onmouseout="myScript">
```

In JavaScript:

```
object.onmouseout = function(){myScript};
```

In JavaScript, using the addEventListener() method:

```
object.addEventListener("mouseout", myScript);
```

Note: The [addEventListener\(\)](#) method is not supported in Internet Explorer 8 and earlier versions.

Technical Details

Supported HTML tags: All HTML elements, EXCEPT: <base>, <bdo>,
, <head>, <html>, <iframe>, <param>, <script>, <style>, and <title>

DOM Version: Level 2 Events

Example

This example demonstrates the difference between the `onmousemove`, `onmouseleave` and `onmouseout` events:

```
<div onmousemove="myMoveFunction()">
  <p id="demo">I will demonstrate onmousemove!</p>
</div>
```

```
<div onmouseleave="myLeaveFunction()">
  <p id="demo2">I will demonstrate onmouseleave!</p>
</div>
```

```
<div onmouseout="myOutFunction()">
  <p id="demo3">I will demonstrate onmouseout!</p>
</div>
```

Let's dive into more details about events that happen when the mouse moves between elements.

Events `mouseover`/`mouseout`, `relatedTarget`

The `mouseover` event occurs when a mouse pointer comes over an element, and `mouseout` – when it leaves.

These events are special, because they have property `relatedTarget`. This property complements `target`. When a mouse leaves one element for another, one of them becomes `target`, and the other one – `relatedTarget`.

For `mouseover`:

- `event.target` – is the element where the mouse came over.
- `event.relatedTarget` – is the element from which the mouse came (`relatedTarget` → `target`).

For `mouseout` the reverse:

- `event.target` – is the element that the mouse left.
- `event.relatedTarget` – is the new under-the-pointer element, that mouse left for (`target` → `relatedTarget`).

onfocus and onblur

onblur Event

Example

Execute a JavaScript when a user leaves an input field:

```
<input type="text" onblur="myFunction()">
```

Definition and Usage

The `onblur` event occurs when an object loses focus.

The `onblur` event is most often used with form validation code (e.g. when the user leaves a form field).

Tip: The `onblur` event is the opposite of the [onfocus](#) event.

Tip: The `onblur` event is similar to the [onfocusout](#) event. The main difference is that the `onblur` event does not bubble. Therefore, if you want to find out whether an element or its child loses focus, you could use the `onfocusout` event. However, you can achieve this by using the optional `useCapture` parameter of the [addEventListener\(\)](#) method for the `onblur` event.

Syntax

In HTML:

```
<element onblur="myScript">
```

In JavaScript:

In JavaScript, using the `addEventListener()` method:

```
object.onblur = function(){myScript};
```

```
object.addEventListener("blur", myScript);
```

Note: The [addEventListener\(\)](#) method is not supported in Internet Explorer 8 and earlier versions.

Technical Details

Bubbles:	No
Cancelable:	No
Event type:	FocusEvent
Supported HTML tags:	ALL HTML elements, EXCEPT: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><ifra</code> <code><param></code> , <code><script></code> , <code><style></code> , and <code><title></code>
DOM Version:	Level 2 Events

onfocus

Example

Using `onblur` together with the `onfocus` event:

```
<input type="text" onfocus="focusFunction()" onblur="blurFunction()">
```

Definition and Usage

The `onfocus` event occurs when an element gets focus.

The `onfocus` event is most often used with `<input>`, `<select>`, and `<a>`.

Tip: The `onfocus` event is the opposite of the [onblur](#) event.

Tip: The `onfocus` event is similar to the [onfocusin](#) event. The main difference is that the `onfocus` event does not bubble. Therefore, if you want to find out whether an element or its child gets the focus, you could use the `onfocusin` event. However, you can achieve this by using the optional `useCapture` parameter of the [addEventListener\(\)](#) method for the `onfocus` event.

Example

Execute a JavaScript when an input field gets focus:

```
<input type="text" onfocus="myFunction()">
```

Syntax

In HTML:

In JavaScript:

```
<element onfocus="myScript">
```

```
object.onfocus = function(){myScript};
```

In JavaScript, using the `addEventListener()` method:

```
object.addEventListener("focus", myScript);
```


Note: The [addEventListener\(\)](#) method is not supported in Internet Explorer 8 and earlier versions.

Technical Details

Bubbles:	No
Cancelable:	No
Event type:	FocusEvent
Supported HTML tags:	ALL HTML elements, EXCEPT: <base>, <bdo>, , <head>, <html>, <iframe>, <param>, <script>, <style>, and <title>
DOM Version:	Level 2 Events

Example

Using "onfocus" together with the "onblur" event:

```
<input type="text" onfocus="focusFunction()" onblur="blurFunction()">
```

Form Processing With Onsubmit And Onreset

onsubmit:

Enter Text:

Example:

```
<script type="text/javascript">
<!--
function validForm(theForm) {
  if (theForm.theText.value == "") return false;
  return true;
};
//-->
</script>
```

```
<form name="boo1" method="GET" action="#" onsubmit="return validForm(this)">
Enter Text:<input type="text" value="" name="theText" size="30" />
<input type="submit" name="submit" value="submit" />
</form>
```

The onsubmit event can be used to validate the contents of an HTML form before it is submitted, and block that submission if the form is not filled out correctly. This can be very useful for ensuring that users have not missed some required information from the form.

The way the onsubmit event must be used is slightly different from previous events - first, it can only be used on the **form** tag, and second, it must be used in conjunction with the **return** command, to return a value of either **true** or **false**. If a value of **true** is returned, then the form will submit successfully. If a value of **false** is returned instead, then the form will not

submit. If you leave off the **return** command from the **onsubmit="..."**, then it will not be able to stop the form from being submitted.

Onreset

The onreset event occurs when a form is reset.

Example

Execute a JavaScript when a form is reset:

```
<form onreset="myFunction()">  
  Enter name: <input type="text">  
  <input type="reset">  
</form>
```

Syntax

In HTML:

```
<element onreset="myScript">
```

In JavaScript:

In JavaScript, using the addEventListener() method:

```
object.onreset = function(){myScript};  
object.addEventListener("reset", myScript);
```

Technical Details

Bubbles:	Yes
Cancelable:	Yes
Event type:	Event
Supported HTML tags:	<form>
DOM Version:	Level 2 Events

Event Bubbling And Other Events.

Event bubbling is a method of event propagation in the HTML DOM API when an event is in an element inside another element, and both elements have registered a handle to that event. It is a process that starts with the element that triggered the event and then bubbles up to the containing elements in the hierarchy.

Syntax:

```
addEventListener(type, listener, useCapture)
```

- **type:** Use to refer to the type of event.
- **listener:** Function we want to call when the event of the specified type occurs.
- **userCapture:** Boolean value. Boolean value indicates event phase. By Default useCapture is false. It means it is in the bubbling phase.

XML:

XML (Extensible Markup Language) is a **markup language similar to HTML, but without predefined tags to use**. Instead, you define your own tags designed specifically for your needs. This is a powerful way to store data in a format that can be stored, searched, and shared.

- **Xml** (eXtensible Markup Language) is a mark up language.
- XML is designed to store and transport data.
- Xml was released in late 90's. it was created to provide an easy to use and store self describing data.
- XML became a W3C Recommendation on February 10, 1998.
- XML is not a replacement for HTML.
- XML is designed to be self-descriptive.
- XML is designed to carry data, not to display data.
- XML tags are not predefined. You must define your own tags.
- XML is platform independent and language independent.

Differences between XML and HTML

XML and HTML were designed with different goals:

- XML is designed to carry data emphasizing on what type of data it is.
- HTML is designed to display data emphasizing on how data looks
- XML tags are not predefined like HTML tags.
- HTML is a markup language whereas XML provides a framework for defining markup languages.
- HTML is about displaying data,hence it is static whereas XML is about carrying information,which makes it dynamic.

Structure of an XML document

Whole structure XML and XML based languages built on [tags](#).

XML declaration

XML - declaration is not a tag. It is used for the transmission of the meta-data of a document.

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

Copy to Clipboard

Attributes:

version:

Used version XML in this document.

encoding :

Used encoding in this document.

Comments

```
<!-- Comment -->  
Copy to Clipboard
```

"Correct" XML (valid and well-formed)

Correct design rules

For an XML document to be correct, the following conditions must be fulfilled:

- Document must be well-formed.
- Document must conform to all XML syntax rules.
- Document must conform to semantic rules, which are usually set in an XML schema or a DTD (Document Type Definition).

Example

```
<?xml version="1.0" encoding="UTF-8"?>  
<message>  
<warning>  
    Hello World  
<!--missing </warning> -->  
</message>
```

Now let's look at a corrected version of that same document:

```
<?xml version="1.0" encoding="UTF-8"?>  
<message>  
<warning>  
    Hello World  
</warning>  
</message>
```

Copy to Clipboard

A document that contains an undefined tag is invalid. For example, if we never defined the `<warning>` tag, the document above wouldn't be valid.

Most browsers offer a debugger that can identify poorly-formed XML documents.

Entities

Like HTML, XML offers methods (called entities) for referring to some special reserved characters (such as a greater than sign which is used for tags). There are five of these characters that you should know:

Entity	Character	Description
<	<	Less than sign
>	>	Greater than sign
&	&	Ampersand
"	"	One double-quotation mark
'	'	One apostrophe (or single-quotation mark)

Even though there are only 5 declared entities, more can be added using the document's [Document Type Definition](#).

For example, to create a new `&warning;` entity, you can do this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE body [
<!ENTITY warning "Warning: Something bad happened... please refresh and try again.">
]>
<body>
<message>&warning;</message>
</body>
```

Copy to Clipboard

You can also use numeric character references to specify special characters; for example, `©` is the "©" symbol.

Displaying XML

XML is usually used for descriptive purposes, but there are ways to display XML data. If you don't define a specific way for the XML to be rendered, the raw XML is displayed in the browser.

One way to style XML output is to specify [CSS](#) to apply to the document using the `xml-stylesheet` processing instruction.

```
<?xml-stylesheet type="text/css" href="stylesheet.css"?>
```

Copy to Clipboard

There is also another more powerful way to display XML: the **Extensible Stylesheet Language Transformations** ([XSLT](#)) which can be used to transform XML into other languages such as HTML. This makes XML incredibly versatile.

```
<?xml-stylesheet type="text/xsl" href="transform.xsl"?>
```

Basics structuring Data

XML Tree Structure

XML documents are formed as **element trees**.

An XML tree starts at a **root element** and branches from the root to **child elements**. All elements can have sub elements (child elements):

```
<root>
  <child>
    <subchild> ....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements.

Parents have children. Children have parents. Siblings are children on the same level (brothers and sisters).

All elements can have text content (Harry Potter) and attributes (category="cooking").

Self-Describing Syntax

XML uses a much self-describing syntax.

A prolog defines the XML version and the character encoding:

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

The next line is the **root element** of the document:

```
<bookstore>
```

The next line starts a <book> element:

```
<book category="cooking">
```

The <book> elements have **4 child elements**: <title>, <author>, <year>, <price>.

```
<title lang="en">Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
```

The next line ends the book element:

```
</book>
```

XML Name Spaces

XML Namespaces

XML Namespaces provide a method to avoid element name conflicts.

Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

This XML carries HTML table information:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a table (a piece of furniture):

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

A user or an XML application will not know how to handle these differences.

Solving the Name Conflict Using a Prefix

Name conflicts in XML can easily be avoided using a name prefix.

This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

In the example above, there will be no conflict because the two <table> elements have different names.

XML Namespaces - The xmlns Attribute

When using prefixes in XML, a **namespace** for the prefix must be defined.

The namespace can be defined by an **xmlns** attribute in the start tag of an element.

The namespace declaration has the following syntax. *xmlns:prefix="URI"*.

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>
```



```
<f:table xmlns:f="https://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

In the example above:

The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace.

The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace.

When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

Namespaces can also be declared in the XML root element:

```
<root xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="https://www.w3schools.com/furniture">

  <h:table>
    <h:tr>
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
  </h:table>

  <f:table>
    <f:name>African Coffee Table</f:name>
    <f:width>80</f:width>
    <f:length>120</f:length>
  </f:table>

</root>
```

Note: The namespace URI is not used by the parser to look up information.

The purpose of using an URI is to give the namespace a unique name.

However, companies often use the namespace as a pointer to a web page containing namespace information.

Uniform Resource Identifier (URI)

A **Uniform Resource Identifier (URI)** is a string of characters which identifies an Internet Resource.

The most common URI is the **Uniform Resource Locator** (URL) which identifies an Internet domain address. Another, not so common type of URI is the **Uniform Resource Name** (URN).

Default Namespaces

Defining a default namespace for an element saves us from using prefixes in all the child elements. It has the following syntax:

```
xmlns="namespaceURI"
```

This XML carries HTML table information:

```
<table xmlns="http://www.w3.org/TR/html4/">
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a piece of furniture:

```
<table xmlns="https://www.w3schools.com/furniture">
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

Namespaces in Real Use

XSLT is a language that can be used to transform XML documents into other formats.

The XML document below, is a document used to transform XML into HTML.

The namespace "<http://www.w3.org/1999/XSL/Transform>" identifies XSLT elements inside an HTML document:

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

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

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

```
<html>
```

```
<body>
```

```
<h2>My CD Collection</h2>
```

```
<table border="1">
```

```
<tr>
```

```
<th style="text-align:left">Title</th>
```

```
<th style="text-align:left">Artist</th>
```

```
</tr>
```

```
<xsl:for-each select="catalog/cd">
```

```
<tr>
```

```
<td><xsl:value-of select="title"/></td>
```

```

        <td><xsl:value-of select="artist"/></td>
    </tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>

</xsl:stylesheet>

```

Document Type Definition – DTD

- Difficulty Level : [Easy](#)
 - Last Updated : 06 Nov, 2020
- A Document Type Definition (**DTD**) describes the tree structure of a document and something about its data. It is a set of markup affirmations that actually define a type of document for the SGML family, like GML, SGML, HTML, XML.
- A DTD can be declared inside an XML document as inline or as an external recommendation. DTD determines how many times a node should appear, and how their child nodes are ordered.

There are 2 data types, PCDATA and CDATA

- PCDATA is parsed character data.
- CDATA is character data, not usually parsed.

Why Use a DTD?

With a DTD, independent groups of people can agree on a standard DTD for interchanging data.

An application can use a DTD to verify that XML data is valid.

An Internal DTD Declaration

If the DTD is declared inside the XML file, it must be wrapped inside the <!DOCTYPE> definition:

XML document with an internal DTD

```

<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
<note>
<to>Tove</to>
<from>Jani</from>

```

```
<heading>Reminder</heading>
<body>Don't forget me this weekend</body>
</note>
```

In the XML file, select "view source" to view the DTD.

The DTD above is interpreted like this:

- **!DOCTYPE note** defines that the root element of this document is note
- **!ELEMENT note** defines that the note element must contain four elements: "to,from,heading,body"
- **!ELEMENT to** defines the to element to be of type "#PCDATA"
- **!ELEMENT from** defines the from element to be of type "#PCDATA"
- **!ELEMENT heading** defines the heading element to be of type "#PCDATA"
- **!ELEMENT body** defines the body element to be of type "#PCDATA"

An External DTD Declaration

If the DTD is declared in an external file, the <!DOCTYPE> definition must contain a reference to the DTD file:

XML document with a reference to an external DTD

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

And here is the file "note.dtd", which contains the DTD:

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

W3C XML schema documents

What is an XML Schema?

An XML Schema describes the structure of an XML document.

The XML Schema language is also referred to as XML Schema Definition (XSD).

XSD Example

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

```
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

The purpose of an XML Schema is to define the legal building blocks of an XML document:

- the elements and attributes that can appear in a document
- the number of (and order of) child elements
- data types for elements and attributes
- default and fixed values for elements and attributes

Why Learn XML Schema?

In the XML world, hundreds of standardized XML formats are in daily use. Many of these XML standards are defined by XML Schemas.

XML Schema is an XML-based (and more powerful) alternative to DTD.

XML Schemas Support Data Types

One of the greatest strength of XML Schemas is the support for data types.

- It is easier to describe allowable document content
- It is easier to validate the correctness of data
- It is easier to define data facets (restrictions on data)
- It is easier to define data patterns (data formats)
- It is easier to convert data between different data types

XML Schemas use XML Syntax

Another great strength about XML Schemas is that they are written in XML.

- You don't have to learn a new language
- You can use your XML editor to edit your Schema files
- You can use your XML parser to parse your Schema files
- You can manipulate your Schema with the XML DOM
- You can transform your Schema with XSLT

XML Schemas are extensible, because they are written in XML.

With an extensible Schema definition you can:

- Reuse your Schema in other Schemas
- Create your own data types derived from the standard types
- Reference multiple schemas in the same document

XML Schemas Secure Data Communication

When sending data from a sender to a receiver, it is essential that both parts have the same "expectations" about the content.

With XML Schemas, the sender can describe the data in a way that the receiver will understand.

A date like: "03-11-2004" will, in some countries, be interpreted as 3.November and in other countries as 11.March.

However, an XML element with a data type like this:

```
<date type="date">2004-03-11</date>
```

ensures a mutual understanding of the content, because the XML data type "date" requires the format "YYYY-MM-DD".

Well-Formed is Not Enough

A well-formed XML document is a document that conforms to the XML syntax rules, like:

- it must begin with the XML declaration
- it must have one unique root element
- start-tags must have matching end-tags
- elements are case sensitive
- all elements must be closed
- all elements must be properly nested
- all attribute values must be quoted
- entities must be used for special characters

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences.

Think of the following situation: you order 5 gross of laser printers, instead of 5 laser printers. With XML Schemas, most of these errors can be caught by your validating software.

XML Vocabularies

XML vocabularies are the elements used in particular applications or data formats - the definitions of the meanings of those formats. For example, in CDF, element names such as <SCHEDULE>, <CHANNEL>, and <ITEM> make up the vocabulary for describing collections of pages, when these pages should be downloaded, etc.

Vocabularies, along with the structural relationships between the elements, are defined in XML DTDs or XML schemas.

Book XML vocabulary - just for marking up book data:

```
<Book>  
<Title>The Wisdom of Crowds</Title>  
<Classification>non-fiction</Classification>
```

```
<Author>James Surowiecki</Author>
```

```
</Book>
```

Music XML vocabulary - just for marking up music data:

```
<Musical-Score>
```

```
<Work>Winterreise</Work>
```

```
<Genre>classical</Genre>
```

```
<Composer>Franz Schubert</Composer>
```

```
</Musical-Song>
```

Here the same data is marked up, this time using a single generic XML vocabulary:

Composition XML vocabulary - for marking up any literary data:

```
<Compositionclass="Book">
```

```
<Title>The Wisdom of Crowds</Title>
```

```
<Category>non-fiction</Category>
```

```
<Creator>James Surowiecki</Creator>
```

```
</Composition>
```

```
<Compositionclass="Musical-Score">
```

```
<Title>Winterreise</Title>
```

```
<Category>classical</Category>
```

```
<Creator>Franz Schubert</Creator>
```

```
</Composition>
```

XSLT

What is XSLT

Before XSLT, first we should learn about XSL. XSL stands for EXtensible Stylesheet Language. It is a styling language for XML just like CSS is a styling language for HTML.

XSLT stands for XSL Transformation. It is used to transform XML documents into other formats (like transforming XML into HTML).

What is XSL

In HTML documents, tags are predefined but in XML documents, tags are not predefined. World Wide Web Consortium (W3C) developed XSL to understand and style an XML document, which can act as XML based Stylesheet Language.

An XSL document specifies how a browser should render an XML document.

Main parts of XSL Document

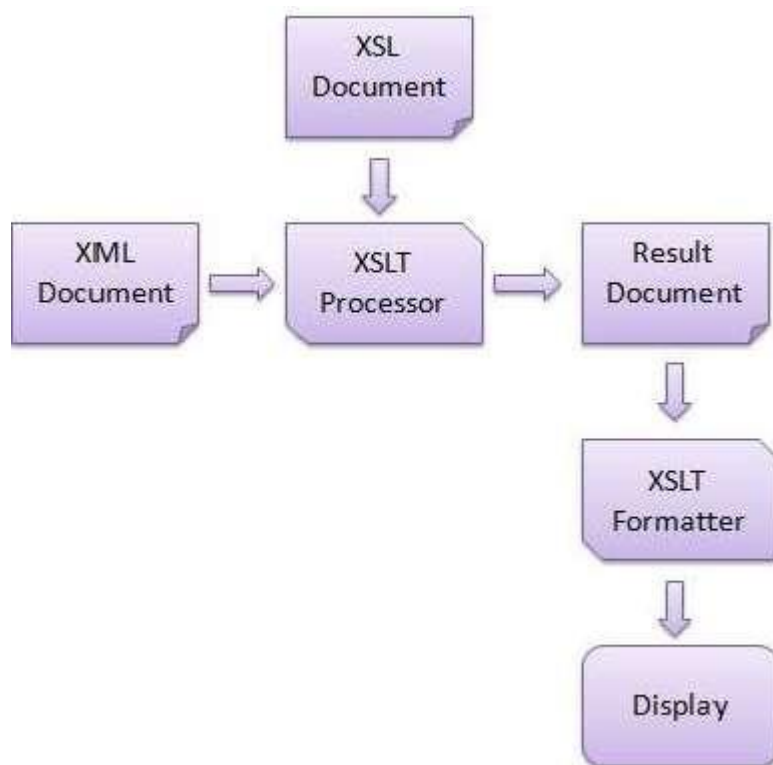
- **XSLT:** It is a language for transforming XML documents into various other types of documents.
- **XPath:** It is a language for navigating in XML documents.

- **XQuery:** It is a language for querying XML documents.
 - **XSL-FO:** It is a language for formatting XML documents.
-

How XSLT Works

The XSLT stylesheet is written in XML format. It is used to define the transformation rules to be applied on the target XML document. The XSLT processor takes the XSLT stylesheet and applies the transformation rules on the target XML document and then it generates a formatted document in the form of XML, HTML, or text format. At the end it is used by XSLT formatter to generate the actual output and displayed on the end-user.

Image representation:



Advantage of XSLT

A list of advantages of using XSLT:

- XSLT provides an easy way to merge XML data into presentation because it applies user defined transformations to an XML document and the output can be HTML, XML, or any other structured document.

- XSLT provides XPath to locate elements/attribute within an XML document. So it is more convenient way to traverse an XML document rather than a traditional way, by using scripting language.
- XSLT is template based. So it is more resilient to changes in documents than low level DOM and SAX.
- By using XML and XSLT, the application UI script will look clean and will be easier to maintain.
- XSLT templates are based on XPath pattern which is very powerful in terms of performance to process the XML document.
- XSLT can be used as a validation language as it uses tree-pattern-matching approach.
- You can change the output simply modifying the transformations in XSL files.

Summary:

- There are three ways to assign an event handler: HTML event handler attribute, element's event handler property, and `addEventListener()`.
- Assign an event handler via the HTML event handler attributes should be avoided.
- **In** this tutorial, you will learn the various ways to perform event handling in JavaScript.

The Document Object Model API. **The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents.** A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch.

Questions:

1. Describe Document Object Model (DOM)?
2. Explain about Events?
3. Define XML?
4. Explain in detail about XML schema documents and XML Name Spaces?
5. Explain XSLT?

