

Unit 1: Review of Web Technologies

1.1 Introduction

Web technology is related to the interface between web servers and their clients using the **internet**. This information includes markup languages, programming interfaces and languages, and standards for document identification and display.

Origin of Internet

Its origin dates back to 1969 when it was called **ARPAnet** (Advanced Research Project Agency network) and was exclusively used for military purposes. It soon emerged with non-governmental and parallel academic networks which grew and eventually came to be called **Internet** in the year 1979.

Today, **Internet** is simply a network of world-wide network of computer networks connected to each other by devices called **Internetworking devices**. The computers on the Internet contains information on history, politics and medicine, science and technology, sports, current events and many more topics and thus it is also called '**information super highway**'. The Internet is growing exponentially every day and has made the planet a '**global village**' where everybody can be connected to each other.

Applications of Internet

- Exchange emails.
- Send/receive documents, sound, animation and graphics or picture files all over the world.
- Browse information on matters related to academic and professional topics.
- Join specific topic-oriented discussion groups - **Blogs**.
- Sell products and services – **E-commerce**.
- Entertainment (games, music, etc).
- Download software and software updates.
- Online education and exams.

Internet vs. Intranet vs. Extranet

Internet: A worldwide network of thousands of smaller computer networks and millions of commercial, educational, government and personal computers. The **Internet** is like an electronic city with virtual libraries, stores, art galleries and so on.

Intranet: A network within an organization that uses Internet technologies (such as the HTTP or FTP protocol). Intranet can be called internal Internet. Only the employees of the company can use this network.

Extranet: An **Extranet** is an **Intranet** that supports controlled public access. Extranets offer controlled access to an Intranet for remote access and e-commerce purposes.

Internet in Nepal

The Internet was first introduced into Nepal in 1993 in a venture of the then Royal Nepal Academy of Science and Technology (RONAST) and a private company, Mercantile Office Systems (MOS).

ISP (Internet Service Provider)

A business that provides access to the Internet for such things as electronic mail, chat rooms, or use of the World Wide Web. They supply Internet access in wide range from dial-up modem to DSL and cable

modem broadband service to dedicated T1/T3 lines (reserved circuits that operate over either copper or fiber optic cables). Some ISP's are multinational, offering access in many locations while others are limited to a specific region. **Example:** *Worldlink Communications, Mercantile Communications (First ISP in Nepal), Nepal Telecommunications Corporation, Infocom Pvt. Ltd., Websurfer Nepal etc.*

URI vs. URL vs. URN

- **URIs identify** and **URLs locate**; however, **locators are also identifiers**, so every URL is also a URI, but there are URIs which are not URLs.
- Bishnu Rawal: This is my name, which is an identifier. It is like a URI, but cannot be a URL, as it tells you nothing about my location or how to contact me. In this case it also happens to identify at least 5 other people in the Nepal alone.
- 4914 West Bay Street, Nassau, Bahamas: This is a locator, which is an identifier for that physical location. It is like both a URL and URI (since all URLs are URIs), and also identifies me indirectly as "resident of..". In this case it uniquely identifies me, but that would change if I get a roommate.
- URNs (Uniform Resource Name) are **URLs**, except those are much more regulated and intended to be globally unique.
- Examples:
 - URL: ftp://ftp.is.co.za/rfc/rfc1808.txt
 - URL: http://www.ietf.org/rfc/rfc2396.txt
 - URL: ldap://[2001:db8::7]/c=GB?objectClass?one
 - URL: mailto:John.Doe@example.com
 - URL: news:comp.infosystems.www.servers.unix
 - URL: telnet://192.0.2.16:80/
 - URN (not URL): rn:oasis:names:specification:docbook:dtd:xml:4.1.2
 - URN (not URL): tel:+1-816-555-1212 (?)
 - URN (not URL): isbn:0-486-27557-4 (Shakespeare's play Romeo and Juliet)
 -

Web URL Syntax

URLs have three basic parts: (Example- "**http://www.smccd.edu/accounts/csmlibrary/index.htm**")

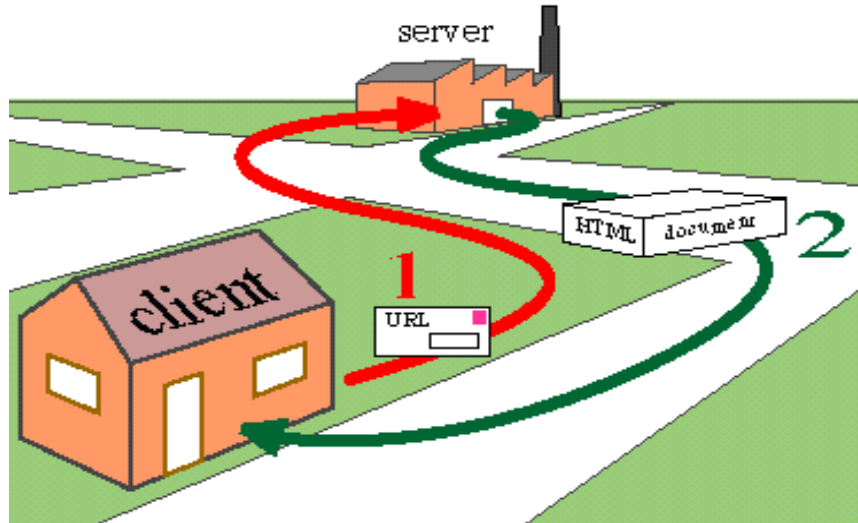
1. **Protocol:** identifies the method (set of rules) by which the resource is transmitted. "**http://**"
2. **server name:** Also called **domain name**, identifies the computer on which the resource is found. The domain (.org, .edu, .gov etc) is important because it usually identifies the type of organization that created or sponsored the resource. "**www.smccd.edu**"
3. **resource ID:** name of the file for the page and any directories or subdirectories under which it is stored on the specified computer. "**/accounts/csmlibrary/index.htm**"

WWW (World Wide Web)

WWW is a system of interlinked hypertext documents that are accessed via the Internet. With a web browser, one can view web pages that may contain text, images, videos, and other multimedia and navigate between them via hyperlinks.

- **Tim Berners Lee**, a British computer scientist invented WWW in 1989.
- Consists of all the public Web sites connected to the Internet worldwide, including the client devices (such as computers and cell phones) that access Web content.
- The WWW is just one of many applications of the Internet and computer networks and is based on: HTML, HTTP, Web servers and Web browsers.
- **How does it work?**

- Web information is stored in documents called **Web pages (Html files)**. Web pages are files stored on computers called **Web servers**. Computers reading the Web pages are called **Web clients**. Web clients view the pages with a program called a **Web browser**.
- A **web browser** is client software that allows you to display and interact with a hypertext document hosted on the web server. Popular browsers are Chrome, Mozilla Firefox, Opera, Internet Explorer and Netscape Navigator. A browser fetches a Web page from a server by a request. A request is a standard HTTP request containing a page location address. Something like: **`http://www.someone.com/page.htm`**
- All Web pages contain display instructions like **HTML tags** which the browser reads to display page information.



Who is making web standards?

- The rule-making body of the Web is the **W3C (World Wide Web Consortium)** founded in 1994 and currently led by Tim Berners-Lee (Inventor of WWW), the consortium is made up of member organizations which maintain full-time staff for the purpose of working together with **IETF** in the development of standards for the World Wide Web. As of 24 May 2014, the W3C has 385 members.
- **W3C** puts together specifications for Web standards.
- Some common web standards are HTML, CSS, XML, SOAP, XQUERY, XPATH, XSLT, WSDL etc.

World Wide Web Coordinating Groups (Besides W3C)

There are many types of technologies which are used to support the World Wide Web and more are being developed all the time. There are several groups involved in the development and coordination of these technologies.

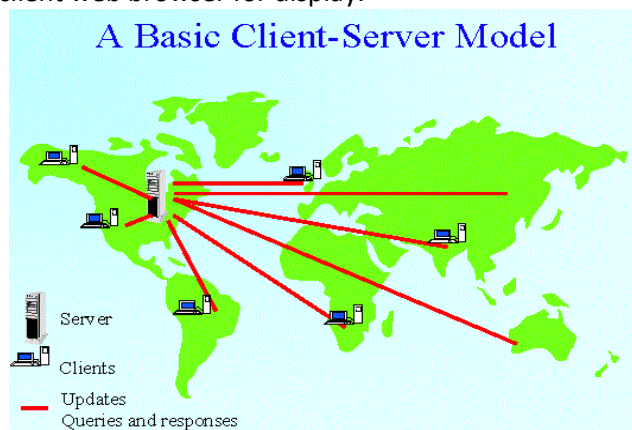
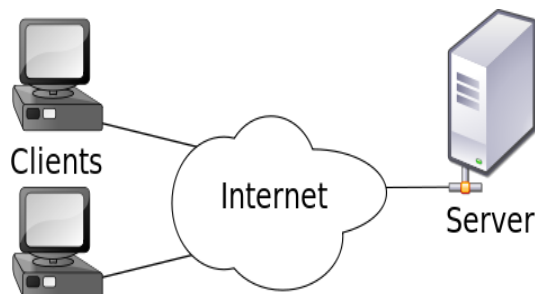
- **IAB** - Internet Architecture Board. Web site: [IAB](#). The IAB websites states that "The IAB does not aim to produce polished technical proposals on such topics. Rather, the intention is to stimulate action by the IESG or within the IETF community that will lead to proposals that meet general consensus."
- **IANA** - Internet Assigned Numbers Authority. Web site: [IANA](#). They control the assignment of internet addresses and domain names.
- **IESG** - Internet Engineering Steering Group. Web site: [IESG](#). According to RFC 2418, the IESG "has responsibility for developing and reviewing specifications intended as Internet Standards."

- **IETF - Internet Engineering Task Force.** Web site: [IETF](#). Their web site says "The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet."
- **InterNIC - Internet Network Information Center**, the authority for allocating internet addresses. Web site: [InterNIC](#).
- **IRTF - Internet Research Task Force.** Web site: [IRTF](#). Their web site states their mission is "To promote research of importance to the evolution of the future Internet by creating focused, long-term and small Research Groups working on topics related to Internet protocols, applications, architecture and technology."
- **ISOC - Internet Society, promotes internet policies.** Web site: [ISOC](#).
- **ISTF - Internet Societal Task Force.** Web site: [ISTF](#). Their mission is "To assure the open development, evolution and use of the Internet for the benefit of all people throughout the world".
- **W3C - World Wide Web Consortium** sets standards for the web working with the IETF. [W3C](#)
- **OASIS - Organization for the Advancement of Structured Information Standards.** [OASIS](#)
- **Internet2** [Internet2](#) An Organization that supports internet related technologies including XML, DHTML, JAVA and more.
- **IRT - Internet Related Technologies.** Website: [Internet Related Technologies](#) Their website states "Internet2, led by over 180 U.S. universities working in partnership with industry and government, is developing and deploying advanced network applications and technologies, accelerating the creation of tomorrow's Internet."
- **[Graphic Communications Association](#)** A trade association that provides standards for the printing and publishing industries.

Client-Server Model

The **client-server model** of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called **servers**, and service requesters, called **clients**. Often clients and servers communicate over a **computer network** on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients.

Example: When a bank customer accesses online banking services with a web browser (the client), the client initiates a request to the bank's web server. The customer's login credentials may be stored in a database, and the web server accesses the database server as a client. An application server interprets the returned data by applying the bank's business logic, and provides the output to the web server. Finally, the web server returns the result to the client web browser for display.



- Clients initiate communication sessions with servers which await incoming requests.
- Examples of computer applications that use the client–server model are Email, network printing, and the World Wide Web.

Server types

- **Proxy Servers:** Proxy servers sit between a client program (typically a web browser) and an external server (typically another server on the web) to filter requests, improve performance, and share connections.
- **Mail Servers:** They involve getting and posting emails using protocols such as POP and IMAP for receiving emails and SMTP for transferring emails.
- **Web Servers:** They hold the text and graphics for a specific web site which can be viewed by web browsers. The web servers respond to the request for information made by the client. They communicate with the clients via HTTP.
- **File Servers:** They are used to upload or download files and employ the FTP protocol.
- **Telnet Servers:** They enable users to log on to a remote computer and perform tasks as if they are working on the remote computer itself.
- **Database Servers:** Database server is the term used to refer to the **back-end** system of a database application using client/server architecture. The back-end, sometimes called a database server, performs tasks such as data analysis, storage, data manipulation, archiving, and other non-user specific tasks.
- **Application Servers:** Sometimes referred to as a type of **middleware**, application servers occupy a large chunk of computing territory between database servers and the end user, and they often connect the two.
- **Communications server:** Carrier-grade computing platform for communications networks.
- **Name server** or DNS
- **Game server:** a server that video game clients connect to in order to play online together.

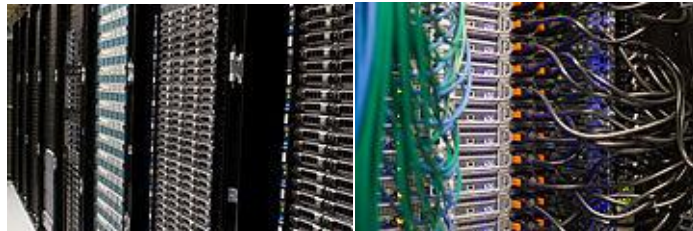


Fig: Wikimedia Foundation servers as seen from the front (left) and rear (right)

Email protocols: SMTP vs POP3 vs IMAP

SMTP, POP3 and IMAP are TCP/IP protocols used for mail delivery. Each protocol is just a specific set of communication rules between computers.

- **SMTP:** SMTP stands for **Simple Mail Transfer Protocol**. SMTP is used when email is delivered from an email client, such as Outlook Express, to an email server or when email is delivered from one email server to another. SMTP uses port 25.
- **POP3:** POP3 stands for **Post Office Protocol**. POP3 allows an email client to download an email from an email server. The POP3 protocol is simple and does not offer many features except for download. Its design assumes that the email client downloads all available email from the server, deletes them from the server and then disconnects. POP3 normally uses port 110.
- **IMAP:** IMAP stands for **Internet Message Access Protocol**. IMAP shares many similar features with POP3. It, too, is a protocol that an email client can use to download email from an email

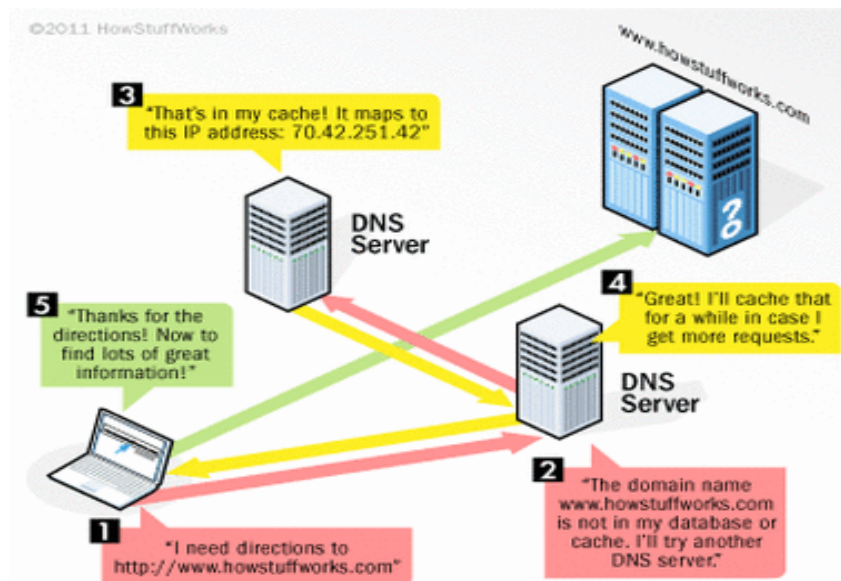
server. However, IMAP includes many more features than POP3. The IMAP protocol is designed to let users keep their email on the server. IMAP requires more disk space on the server and more CPU resources than POP3, as all emails are stored on the server. IMAP normally uses port 143.

Mailing Example:

1. Suppose you use **SendMail** as your **email server** to send an email to `bill@microsoft.com`. You click *Send* in your email client, say, Outlook.
2. Outlook delivers the email to SendMail using the SMTP protocol.
3. SendMail delivers the email to Microsoft's mail server, `mail.microsoft.com`, using SMTP.
4. Bill's Mozilla Mail client downloads the email from `mail.microsoft.com` to his laptop using the POP3 protocol (or IMAP).

DNS (Domain Name System)

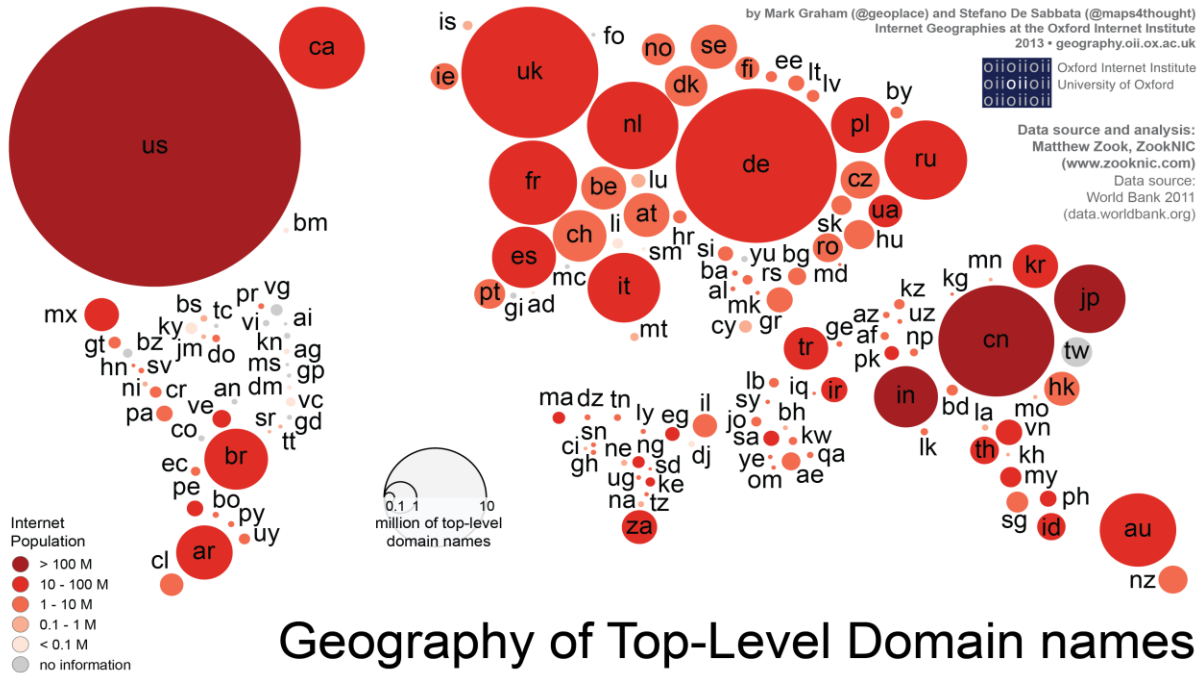
- The **Domain Name System (DNS)** is a hierarchical distributed naming system for computers, services, or any resource connected to the Internet or a private network.
- It translates easily memorized domain names to the numerical IP addresses needed for the purpose of locating computer services and devices worldwide.
- Every computer on the Internet can have both a domain name and an IP address, and when you use a domain name, the computers known as DNS servers translate that name to the corresponding IP address.
- The names of the domains describe organizational and geographic relations. They indicate what country the network connection is in, what kind of organization owns it, and sometimes further details.



Domain Naming:

- **Non-geographic domains:** There are many top-level domain types that are non-geographical. Some of them are:
 - *com* – for commercial organizations (i.e. Business)
 - *net* – for network resources e.g. ISPs

- *gov* – for government organizations (Non-military)
- *edu* – for educational organizations (Universities, Secondary schools, etc)
- *mil* – for military (Army, Navy, Air-force)
- *org* – Other organizations
- **Geographic domains:** The geographically based top-level domain types use two-letter country destinations. E.g. *au* – Australia, *jp* – Japan, *us* – United States



This graphic maps a combination of generic top-level domains ([gTLDs](#)) and country code top-level domains ([ccTLDs](#)) in order to provide an indication of the total number of domain registrations in every country worldwide.

1.2 Review of HTML

What exactly web page is?

A *web page* is a document composed basically of text and special codes called **tags of some markup languages** which make the display of the WWW possible.

- Besides textual information, a web document may also contain images, sound, animation, video and also links to other pages anywhere on the web.
- A **web site** is a collection of web pages maintained by a company, university, government or any individual.
- A **home page** is a web page which opens first while opening any web site.

To create a **web page**, we need only a text editor and a browser.

What is a Markup Language then?

Markup language is a modern system for annotating a document in a way that is syntactically distinguishable from the text. The idea came from the "marking up" of paper manuscripts, i.e., the revision instructions by editors, traditionally written with a blue pencil on authors' manuscripts. In digital media this "blue pencil instruction text" was replaced by tags, that is, instructions are expressed directly by tags or "instruction text encapsulated by tags". A markup language is not a

programming language since it does not contain constructs (loops etc.) which are must for the general purpose programming language (C, C# etc).

Examples: Typesetting instructions such as those found in **troff**, **TeX** and **LaTeX**, or structural markers such as **XML** tags. Markup instructs the software displaying the text to carry out appropriate actions. Some markup languages, such as the widely used **HTML**, have pre-defined presentation semantics, meaning that their specification prescribes how the structured data are to be presented; others, such as XML, do not.

What is HTML?

HyperText Markup Language (**HTML**), one of widely used document formats of the WWW, is an instance of SGML: **Standard Generalized Markup Language** (though, strictly, it does not comply with all the rules of SGML), and follows many of the markup conventions used in the publishing industry.

In 1989, physicist [Sir Tim Berners-Lee](#) wrote a memo proposing an Internet based hypertext system, then specified HTML and wrote the browser and server software in the last part of 1990. The first publicly available description of HTML was a document called "HTML Tags", first mentioned on the Internet by Berners-Lee in late 1991. It describes 18 elements comprising the initial, relatively simple design of HTML. Eleven of these elements still exist in HTML 4.

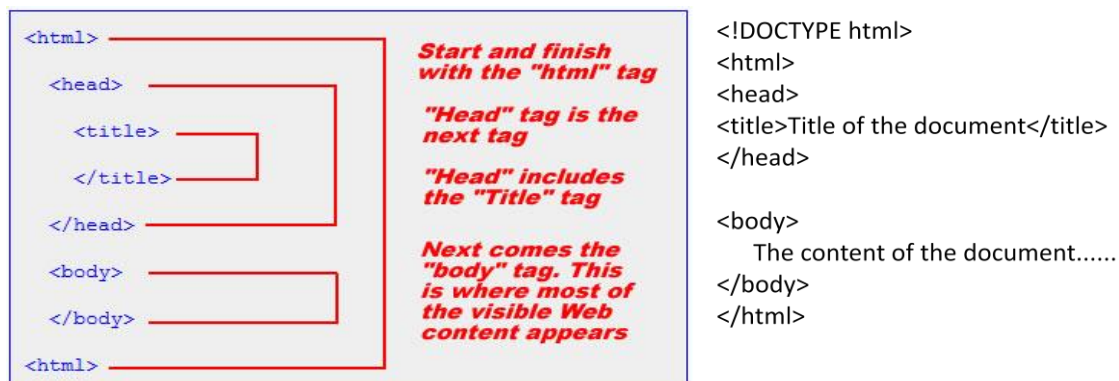
- HTML is the standard markup language used to create web pages.
- A markup language is a set of markup tags
- The tags describe document content
- HTML documents contain HTML tags and plain text. HTML documents are also called web pages

HTML Tags

- HTML markup tags are usually called HTML tags.
- HTML tags are keywords (tag names) surrounded by **angle brackets** like `<html>`
- HTML tags normally **come in pairs** like `<p>` and `</p>`
- The first tag in a pair is the **start tag/opening tag**, the second tag is the **end tag/closing tag**
- The end tag is written like the start tag, with a **slash** before the tag name : `<tagname>content </tagname>`
- HTML tags are not case sensitive: `<P>` means the same as `<p>`

Html document structure

An HTML document is structured with two main elements: **HEAD** and **BODY**



HTML Versions

Version	Year
HTML	1991
HTML+	1993
HTML 2.0	1995
HTML 3.2	1997
HTML 4.01	1999
XHTML	2000
HTML5	2012

Guys, let's explore different html element tags and their usage:

What is this `<!DOCTYPE>` thing at the top?

- The `<!DOCTYPE>` declaration helps the browser to display a web page correctly.
- There are many different documents on the web, and a browser can only display an HTML page 100% correctly if it knows the HTML version and type used.
- Common Declarations
 - HTML5: `<!DOCTYPE html>`
 - HTML 4.01: `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">` XHTML 1.0: `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`

HTML Elements

- An HTML element is everything from the start tag to the end tag.
- **Example:** `<p>This is a paragraph</p>`, `<h1>Biggy Heading</h1>`

Hey!

- Some HTML elements have **empty content**: `<input>`
- Empty elements are **closed in the start tag**: `
`
- Most HTML elements can have **attributes**: `<input type="text">`, **type** is attribute here

Nested Html Elements

- Most HTML elements can be nested (can contain other HTML elements).
- HTML documents consist of nested HTML elements.
- Example:

```
<div id="demo">
  <div id="center">
    <p id="paragraph">
    </p>
  </div>
</div>
```

Html Attributes

- Attributes provide **additional information** about an element and are always specified in the **start tag**
- Attributes come in name/value pairs like: **name="value"**
- Example
 - HTML links are defined with the <a> tag. The link address is specified in the **href attribute**:
Go to google
- Other attributes can be: class, id, title, style etc.

Html <head> element

- The <head> element is a container for all the head elements. Elements inside <head> can include scripts, instruct the browser where to find style sheets, provide meta information, and more.
- The following tags can be added to the head section: <title>, <style>, <meta>, <link>, <script>, <noscript> and <base>.
- Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Title of the document</title>
  <base href="http://www.example.com/" target="_blank" >
  <meta charset="UTF-8">
  <meta name="description" content="Free          Web          tutorials">
  <meta name="keywords" content="HTML,CSS,XML,JavaScript">
  <meta name="author" content="Hege Refsnes">
  <meta http-equiv="refresh" content="30">
  <script type="text/javascript" src="../scripts/custom.js">
  <style rel="stylesheet" type="text/css" href="../content/cus.css">
</head>
<body>
  The content of the document.....
</body>
</html>
```

- **<meta> tag**
 - The <meta> tag provides metadata (data about data) about the HTML document. Metadata will not be displayed on the page, but will be machine parsable.
 - Meta elements are typically used to specify page description, keywords, author of the document, last modified, and other metadata.
 - The metadata can be used by browsers (how to display content or reload page), search engines (keywords), or other web services.

Html Heading Tags

- The <h1> to <h6> tags are used to define HTML headings.
- <h1> defines the most important heading. <h6> defines the least important heading.

```
<body>
  <h1>This is heading 1</h1>
  <h2>This is heading 2</h2>
  <h3>This is heading 3</h3>
```

```
<h4>This is heading 4</h4>
<h5>This is heading 5</h5>
<h6>This is heading 6</h6>
</body>
```

Html Formatting

- HTML uses tags like `` and `<i>` for formatting output, like **bold** or *italic* text.
- Examples: `<p>This text is bold</p>`

Html text formatting tags

- [](#) Defines bold text
- [](#) Defines emphasized text
- [<i>](#) Italic
- [<small>](#) Defines smaller text
- [](#) Defines important text
- [<sub>](#) Defines subscripted text
- [<sup>](#) Defines superscripted text
- [<ins>](#) Defines inserted text
- [](#) Defines deleted text
- [<mark>](#) Defines marked/highlighted text

HTML "Computer Output" Tags

- [<code>](#) Defines computer code text
- [<kbd>](#) Defines keyboard text
- [<samp>](#) Defines sample computer code
- [<var>](#) Defines a variable
- [<pre>](#) Defines preformatted text

HTML Citations, Quotations, and Definition Tags

- [<abbr>](#) Defines an abbreviation or acronym
- [<address>](#) Defines contact information
- [<bdo>](#) Defines the text direction
- [<blockquote>](#) Defines a section that is quoted from another source
- [<q>](#) Defines an inline (short) quotation
- [<cite>](#) Defines the title of a work
- [<dfn>](#) Defines a definition term

Html comments

- syntax: `<!-- Write your comments here -->`
- **Hey!** There is an exclamation point (!) in the opening tag, but not in the closing tag.
- Comments are not displayed by the browser
- With comments you can place notifications and reminders in your HTML.
- **Conditional Comments:** Nothing more than simple HTML comments that IE (up to version 9) happens to take a peep at, are used to throw a chunk of HTML at these browsers and only these browsers.

```
<link href="everything.css" rel="stylesheet">
```

```
<!--[if IE]><link href="stupidie.css" rel="stylesheet"><![endif]-->
```

Hey! Everything between `<!--[if IE]>` and `<![endif]-->` will be picked up by Internet Explorer.

We can also target specific versions of Internet Explorer: `<!--[if IE 6]>...`, `<!--[if IE 7]>...`, `<!--[if IE gte 8]>...`, `<!--[if IE 9]>...`

HTML Hyperlinks

- The HTML `<a>` tag defines a hyperlink.
- A hyperlink (or link) is a word, group of words, or image that you can click on to jump to another document.
- The most important attribute of the `<a>` element is the **href** attribute, which indicates the link's destination.
- **Link Targets**
 - We use it to control link-open behavior. This example will open yahoo in a new window:
``

- Predefined targets are:
 - **_blank** loads the page into a new browser window.
 - **_self** loads the page into the current window.
 - **_parent** loads the page into the frame that is superior to the frame the hyperlink is in.
 - **_top** cancels all frames, and loads in full browser window.
- **Text Links**
 - The HTML code for a link is simple. It looks like this: `Link text`
 - **Example:** `Login to FB`
 - **Use of base path:**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hyperlink Example</title>
    <base href= "http://www.tutorialspoint.com/">
  </head>
  <body>
    <p>Click following link</p>
    <a href="/html/index.htm" target="_blank">HTML Tutorial</a>
  </body>
</html>
```
 - Linking to page section using **name or id** attribute
 - First create a link to the place where you want to reach with-in a webpage and name it using `<a...>` tag as follows: `<h1>HTML Text Links </h1>`
 - Second step is to create a hyperlink to link the document and place where you want to reach: `Go to Top`
 - **Setting link colors (In practice this is done by CSS styles which we will cover later)**
 - We can set colors of your links, active links and visited links using **link**, **alink** and **vlink** attributes of `<body>` tag.
 - **Example:**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hyperlink Example</title>
    <base href="http://www.tutorialspoint.com/">
  </head>
  <body alink="#54A250" link="#040404" vlink="#F40633">
    <p>Click following link</p>
    <a href="/html/index.htm " target="_blank" >HTML Tutorial</a>
  </body>
</html>
```
- **Image links**

It's simple to use an image as hyperlink. We just need to use an image inside hyperlink at the place of text as shown below:

```
<a href="http://www.youtube.com" target="_self">
  
```


- **HTML Email Links**

While using <a> tag as an email tag, you will use **mailto:email address** along with *href* attribute. Following is the syntax of using **mailto** instead of using http.

```
<a href= "mailto:abc@example.com">Send Email</a>
```

Now if a user clicks this link, it launches one Email Client (e.g. Outlook) installed in your computer.

Html Blocks

- All the HTML elements can be categorized into two categories **(a)** Block Level Elements **(b)** Inline Elements
- **Block Elements**
Block elements appear on the screen as if they have a line break before and after them. For example the <p>, <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, , , <dl>, <pre>, <hr />, <blockquote>, and <address> elements are all block level elements. They all start on their own new line, and anything that follows them appears on its own new line.
- **Inline Elements**
Inline elements, on the other hand, can appear within sentences and do not have to appear on a new line of their own. The , <i>, <u>, , , <sup>, <sub>, <big>, <small>, , <ins>, , <code>, <cite>, <dfn>, <kbd>, and <var> elements are all inline elements.

Grouping HTML Elements

- There are two important tags which we use very frequently to group various other HTML tags (i) <div> tag and (ii) tag
- **The <div> tag**
Important block level tag which plays a big role in grouping various other HTML tags and applying CSS on them. Even, <div> tag can be used to create webpage layout where we define different parts (Left, Right, Top etc) of the page. This tag does not provide any visual change on the block but this has more meaning when it is used with CSS.

Example:

```
<body>
  <!-- First group of tags -->
  <div style="color: red">
    <h4>This is first group</h4>
    <p>Following is a list of vegetables</p>
    <ul>
      <li>Beetroot</li>
      <li>Ginger</li>
      <li>Potato</li>
      <li>Radish</li>
    </ul>
  </div>
  <!-- Second group of tags -->
  <div style="color:green">
    <h4>This is second group</h4>
    <p>Following is a list of fruits</p>
    <ul>
```



```
        <li>Apple</li>
        <li>Banana</li>
        <li>Mango</li>
        <li>Strawberry</li>
    </ul>
</div>
</body>
```

- **The tag**

The HTML is an inline element and it can be used to group inline-elements in an HTML document. This tag also does not provide any visual change on the block but has more meaning when it is used with CSS.

- The difference between the tag and the <div> tag is that the tag is used with inline elements where as the <div> tag is used with block-level elements.
- Example:

```
<p>This is <span style="color:red">red</span> and this is <span style="color:green">green </span></p>
```

Html Lists

The most common HTML lists are ordered and unordered lists:

HTML Unordered and ordered lists

- Unordered list starts with tag.
- An ordered list starts with the tag.
- Both cases: Each list item starts with the tag.
- Example:

```
<ul>
<li>Coffee</li>
<li>Milk</li>
</ul>
<ol>
<li>Coffee</li>
<li>Milk</li>
</ol>
```

HTML Description Lists

- A description list is a list of terms/names, with a description of each term/name.
- The <dl> tag defines a description list.
- The <dl> tag is used in conjunction with <dt> (defines terms/names) and <dd> (describes each term/name)
- Ex:

```
<dl>
<dt>Coffee</dt>
<dd>- black hot drink</dd>
<dt>Milk</dt>
<dd>- white cold drink</dd>
</dl>
```

Html Images

- In HTML, images are defined with the tag.
- The tag is empty, which means that it contains attributes only, and has no closing tag.

- To display an image on a page, you need to use the src (Source) attribute: URL of the image you want to display and alt (Alternate text): alternate text for an image, if the image cannot be loaded.
- Height and width (in pixels) can also be set.
- Example:

```

```

HTML Character Entities

- Some characters have a special meaning in HTML, like the less than sign (<) that defines the start of an HTML tag and therefore should be avoided in the plain text. If we want the browser to actually display these characters we must insert character entities in the HTML source.
- A character entity has three parts:
 - An ampersand (&),
 - An entity name or # and
 - An entity number, and finally a semicolon (;).
- To display a less than sign in an HTML document we must write: **<** or **<**;
- **Hey!** Entities are case sensitive.
- ASCII Entities with Entity Names:

<u>Result</u>	<u>Description</u>	<u>Entity Name</u>	<u>Entity Number</u>
"	quotation mark	"	"
&	ampersand	&	&
<	less-than	<	<
>	greater-than	>	>
©	copyright	©	©
®	registered trademark	®	®

Html background

- HTML provides you following two ways to decorate webpage background.
- Html Background with Colors

```
<!-- Format 1 - Use color name -->  
<table bgcolor="lime" >  
<!-- Format 2 - Use hex value -->  
<table bgcolor="#f1f1f1" >  
<!-- Format 3 - Use color value in RGB terms -->  
<table bgcolor="rgb(0,0,120)" >
```

- Html Background with Images

```
<!-- Set table background -->  
<table background="/images/html.gif" width="100%" height="100">
```

Html Tables

The HTML tables allow web authors to arrange data like text, images, links, other tables, etc. into rows and columns of cells. The HTML tables are created using the **<table>** tag in which the **<tr>** tag is used to create table rows and **<td>** tag is used to create data cells.

Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Tables</title>
  </head>
  <body>
    <table border="1">
      <tr>
        <td>Row 1, Column 1</td>
        <td>Row 1, Column 2</td>
      </tr>
      <tr>
        <td>Row 2, Column 1</td>
        <td>Row 2, Column 2</td>
      </tr>
    </table>
  </body>
</html>
```

Table Heading

- Defined using **<th>** tag. This tag could be used to replace **<td>** tag. Normally used for top row representing column names.

```
<table border="1">
  <tr>
    <th>Name</th>
    <th>Salary</th>
  </tr>
  <tr>
    <td>Ramesh Raman</td>
    <td>5000</td>
  </tr>
  ...
</table>
```

Cellpadding and Cellspacing Attributes

- Used to adjust the white space in table cells.
- **cellspacing** attribute: specifies the space between the cells
- **Cellpadding**: represents the distance between cell borders and its content.
- Example: `<table border="1" cellpadding="5" cellspacing="5">... </table>`

Colspan and Rowspan Attributes

- **colspan**: used to merge two or more columns into a single column.
- **rowspan**: used to merge two or more rows.
- Example:

```
...
<tr>
  <td rowspan="2">Row 1 Cell 1</td>
  <td>Row 1 Cell 2</td>
  <td>Row 1 Cell 3</td>
```

```
</tr>
<tr>
    <td>Row 2 Cell 2</td>
    <td>Row 2 Cell 3</td>
</tr>
<tr>
    <td colspan="3">Row 3 Cell 1</td>
</tr>
...
```

Tables Backgrounds: Two ways

- **bgcolor** attribute - background color for whole table or just for one cell.
- **background** attribute - background image for whole table or just for one cell.
- Also bordercolor attribute can be used to set border colors.
- Example:

```
<table border="1" bordercolor="green" bgcolor="yellow">
<table border="1" bordercolor="green" background="/images/test.png">
```

Table Height, Width, Caption

- Table width or height can be specified in terms of pixels or in terms of percentage of available screen area.
- Example:

```
<table border="1" width="400" height="150">
<table border="1" width="100%">
    <caption>This is the caption</caption>
...
</table>
```

Table Header, Body, and Footer

- Tables can be divided into three portions: a header, a body, and a foot
- **<thead>** - to create a separate table header.
- **<tbody>** - to indicate the main body of the table.
- **<tfoot>** - to create a separate table footer.
- A table may contain several **<tbody>** elements to indicate different *pages* or groups of data. But it is notable that **<thead>** and **<tfoot>** tags should appear before **<tbody>**
- **Example:**

```
<table border="1" width="100%">
    <thead>
        <tr> <td colspan="4">This is the head of the table</td> </tr>
    </thead>
    <tfoot>
        <tr>
            <td colspan="4">This is the foot of the table</td>
        </tr>
    </tfoot>
    <tbody>
        <tr>
            <td>Cell 1</td>
            <td>Cell 2</td>
            <td>Cell 3</td>
```

```
                <td>Cell 4</td>
            </tr>
        </tbody>
    </table>
```

Using font attributes

```
<font face = "name" size = "n" color = "color_name">
    ... ..
</font>
```

Colors: Can be given as: azure, blue, black, cyan, fuchsia, grey, green, lime, magenta, maroon, navy, olive, red, silver, white, yellow, etc.

Colors: Can also be given as:

```
#FFFFFF – represents white
#000000 – represents black
#FFFF00 – represents yellow
```

and so on for more combination of colors.

Size: Can be from 1 to 7.

Example: ``

```
<body>
    <font face="Times New Roman" size="5">Times New Roman</font>
    <br />
    <font face="Verdana" size="5" color="red">Verdana</font>
    <br />
    <font face="Comic sans MS" size="5">Comic Sans MS</font>
    <br />
    <font face="WildWest" size="5">WildWest</font>
    <br /> <font face="Bedrock" size="5">Bedrock</font>
    <br />
</body>
```

Alternate font faces:

```
<font face="arial,helvetica">
<font face="Lucida Calligraphy,Comic Sans MS,Lucida Console">
```

Horizontal line: <hr> Tag

- The `<hr>` tag adds a horizontal line.
- Color can be given to the horizontal line as `<hr color = "red">` sIt is empty tag.

Html Frames

- HTML frames are used to divide browser window into multiple sections where each section can load a separate HTML document. A collection of frames in the browser window is known as a **frameset**. The window is divided into frames in a similar way the tables are organized: into rows and columns.
- It's never recommended to use frames since:
 - Some smaller devices cannot cope with frames often because their screen is not big enough to be divided up.
 - Sometimes your page will be displayed differently on different computers due to different screen resolution.
 - The browser's *back button* might not work as the user hopes.

- There are still few browsers that do not support frame technology.

Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Frames</title>
  </head>
  <frameset rows="10%,80%,10%">
    <frame name="top" src="/html/top_frame.htm" />
    <frame name="main" src="/html/main_frame.htm" />
    <frame name="bottom" src="/html/bottom_frame.htm" />
  </frameset>
  <body> Your browser does not support frames. </body>
</html>
```

- Define **<frameset>** instead of **<body>**.
- **<frameset>** tag defines how to divide the window into frames.
 - **rows:** attribute defines horizontal frames
 - **cols:** defines vertical frames.
 - Each frame is indicated by **<frame>** tag and it defines which HTML document shall open into the frame using **src** attribute.

Inline Iframes (Inline Frames)

- We can define an inline frame with HTML tag **<iframe>**. It can appear anywhere in our document. The **<iframe>** tag defines a rectangular region within the document in which the browser can display a separate document, including scrollbars and borders.
- The **src** attribute is used to specify the URL of the document that occupies the inline frame.

Example:

```
<body>
  <p>Document content goes here...</p>
  <iframe src="/html/menu.htm" width="555" height="200">
    Sorry your browser does not support inline frames.
  </iframe>
  <p>Document content also go here...</p>
</body>
```

Html Forms

- HTML Forms are required when you want to collect some data from the site visitor. For example during user registration you would like to collect information such as name, email address, credit card, etc.
- A form will take input from the site visitor and then will post it to a back-end application such as CGI, ASP Script or PHP script etc. The back-end application will perform required processing on the passed data based on defined business logic inside the application.
- There are various form elements available like text fields, textarea fields, drop-down menus, radio buttons, checkboxes, etc.
- The HTML **<form>** tag is used to create an HTML form and it has following syntax:

```
<form action="Script URL" method="GET|POST">
```

form elements like input, textarea etc.

</form>

Form attributes

- **action:** Backend script which is ready to process passed data.
- **method:** Method to be used to upload data. The most frequently used are GET and POST methods.
- **target:** Specify the target window or frame where the result of the script will be displayed. It takes values like `_blank`, `_self`, `_parent` etc.
- **enctype:** You can use the enctype attribute to specify how the browser encodes the data before it sends it to the server. Possible values are:
 - **application/x-www-form-urlencoded** - This is the standard method most forms use in simple scenarios.
 - **multipart/form-data** - This is used when you want to upload binary data in the form of files like image, word file etc.

Form controls

- There are different types of form controls that you can use to collect data using HTML form:
 - Text Input Controls
 - Single-line text input controls - `<input type="text">`
 - List of attributes for **<input>** tag; type: Type of input control, name: Name to the control which is sent to the server . Value: initial value, size: width in characters. maxlength: maximum number of characters a user can enter into the text box.
 - Password input controls - `<input type="password">`
 - Multi-line text input controls - `<textarea>`.
 - Checkboxes Controls
 - Checkboxes are used when more than one option is required to be selected. They are also created using HTML `<input>` tag but type attribute is set to **checkbox**.
 - Attributes: **value:** The value that will be used if the checkbox is selected, **checked:** Set to *checked* if you want to select it by default.
 - Radio Box Controls
 - Radio buttons are used when out of many options, just one option is required to be selected. They are also created using HTML `<input>` tag but type attribute is set to **radio**.
 - Select Box (Dropdown Box) Controls
 - A select box, also called drop down box which provides option to list down various options in the form of drop down list, from where a user can select one or more options.
 - **<select>**: Attributes: Name to the control which is sent to the server to be recognized and get the value, size: This can be used to present a scrolling list box, multiple: If set to "multiple" then allows a user to select multiple items from the menu.
 - **<option>**: **value:** The value that will be used if an option in the select box box is selected, **selected:** Specifies that this option should be the initially selected value when the page loads, **label:** An alternative way of labeling options
 - File Select (Upload) boxes

- If you want to allow a user to upload a file to your web site, you will need to use a file upload box, also known as a file select box. This is also created using the `<input>` element but type attribute is set to **file**.
- Attribute: **accept**-Specifies the types of files that the server accepts.
- Clickable Buttons
 - We can also create a clickable button using `<input>` tag by setting its type attribute to **one of** following values:
 - `submit`- creates a button that automatically submits a form.
 - `reset`- creates a button that automatically resets form controls to their initial values.
 - `button`- creates a button that is used to trigger a client-side script when the user clicks.
 - `image`- creates a clickable button but we can use an image as background of the button.
- Hidden Form Controls
 - Hidden form controls are used to hide data inside the page which later on can be pushed to the server. For This control hides inside the code and does not appear on the actual page.

Example:

```
<form action="" method="POST">
  <div>First name: <input type="text" name="first_name" /> </div>
  <div>Last name: <input type="text" name="last_name" /> </div>
  <div>Last name: <input type="password" name="pwd" /> </div>
  <div><textarea rows="5" cols="50" name="description"></div>
  <input type="checkbox" name="maths" value="on"> Maths
  <input type="checkbox" name="physics" value="on"> Physics
  <input type="radio" name="subject" value="maths"> Maths
  <input type="radio" name="subject" value="physics"> Physics
  <select name="dropdown">
    <option value="Maths" selected>Maths</option>
    <option value="Physics">Physics</option>
  </select>
  <input type="file" name="fileupload" accept="audio/*|video/*|image/*|MIME_type" />
  <input type="submit" name="submit" value="Submit" />
  <input type="reset" name="reset" value="Reset" />
  <input type="button" name="ok" value="OK" />
  <input type="image" name="imagebutton" src="/html/images/logo.png" />
  <input type="hidden" name="pageName" value="10" />
</form>
```

HTML Marquee

An HTML marquee is a scrolling piece of text displayed either horizontally across or vertically down your webpage depending on the settings. This is created by using HTML `<marquees>` tag.

- **Note:** The HTML `<marquee>` tag may not be supported by various browsers so its not recommended to rely on this tag, instead you can use JavaScript and CSS to create such effects.
- Example:

```
<body>
  <marquee>This is basic example of marquee</marquee>
  <marquee width="50%">This example will take only 50% width</marquee>
  <marquee direction="right">This text will scroll from left to right</marquee>
  <marquee direction="up">This text will scroll from bottom to up</marquee>
</body>
```

Working with dynamic image

Adding dynamic image (such as AVI files) and sounds (such as MID files and WAV files) to the web page makes it more attractive.

- **Adding dynamic image:** For adding moving image, we have to use the tag following the *dynsrc* (dynamic source) of the file as:

```

```

Here, loop="infinite" plays the file continuously whereas if you set loop="2", then it plays the file 2 times only.

- **Adding background sound:** For adding background sound to your web page, do as follows:

```
<bgsound src = "path" loop = "infinite">
```

Sound files can be searched within the computer by typing *.mid, *.wav, etc. The hardware requirements to get sound facility are sound card (it may be in-built) and sound box.

Extra Examples:

How to draw a border with a caption around a form. (GroupBox)

```
<!DOCTYPE html>
<html>
  <head>
    <title>GroupBox Demosntration</title>
  </head>
<body>
  <fieldset>
    <legend>
      Health information
    </legend>
    <form>
      Height <input type="text" size="3">
      Weight <input type="text" size="3">
    </form>
  </fieldset>
  <p>If there is no border around the input form, your browser is too old.</p>
</body>
</html>
```

Create an image map, with clickable regions. Each of the regions is a hyperlink.

```
<!--image_regions_hyperlink.htm-->
```

```
<html>
  <body>
    <p>
      Click on head to link to rose, left leg to link to ace and right leg to link to test.htm
```

```
</p>

<map name="link">
  <area shape="rect" coords="75,0,125,40" alt="Link Rose" href="rose.gif">
  <area shape="circle" coords="25,125,25" alt="Link Ace" href="ace.gif">
  <area shape="circle" coords="175,125,25" alt="Link test.htm" href="test.htm">
</map>
</p>
</body>
</html>
```

Redirect a user if your site address has changed.

```
<!--meta_redirect.htm-->
<html>
  <head>
    <meta http-equiv="Refresh" content="5;url=index.htm" >
  </head>
  <body>
    <p>Sorry! We have moved! The new URL is: <a href="index.htm">http://www.xyz.com.np</a>
    </p>
    <p>You will be redirected to the new address in five seconds.</p>
    <p>If you see this message for more than 5 seconds, please click on the link above!</p>
  </body>
</html>
```

Adding image as marquee

```
<!--marquee_single_picture.htm-->
<html>
  <body>
    <marquee direction="right"></marquee>
    <marquee behavior="scrolling" direction="down"></marquee>
  </body>
</html>
```

Marquee of pictures

```
<html>
<head>
  <title>Moving Pictures</title>
</head>
<body bgcolor="silver" text="maroon">
  <center><h1>Creating marquee with pictures</h1></center>
<br><br>
<marquee>
  
  
  
  
```



```




</marquee>
</body>
</html>
```

Putting background sound, calling a dynamic image and marquee

```
<html>
  <head>
    <title>Moving Pictures</title>
  </head>
  <body bgcolor="sky blue">
    <bgsound src="grden_01.mid" loop="infinite">
    <center>
      
      <br>
      <font size="7" face="Lucida Handwriting">
        <u>Sphere</u>
      <br>
      </font>
    </center>
    <br>
    <font size="6" color="red" face="Arial">
      <marquee bgcolor="yellow">The ball is moving.</marquee>
    </font>
  </body>
</html>
```

Cascading Style Sheets (CSS)

Applying CSS

There are three ways to apply CSS to HTML: In-line, internal (Embedded), and external.

In-line

In-line styles are injected straight into the HTML tags using the `style` attribute.

They look something like this:

```
<p style="color: red">text</p>
```

This will make that specific paragraph red.

But, if you remember, the best-practice approach is that the HTML should be a standalone, presentation free document, and so in-line styles should be avoided wherever possible.

Internal

Embedded, or internal, styles are used for the whole page. Inside the `head` element, the `style` tags surround all of the styles for the page.

```
<!DOCTYPE html>
<html>
<head>
<title>CSS Example</title>
<style>

    p {
        color: red;
    }

    a {
        color: blue;
    }

</style>
...
```

This will make all of the paragraphs in the page red and all of the links blue.

Although preferable to soiling our HTML with inline presentation, it is similarly usually preferable to keep the HTML and the CSS files separate, and so we are left with our savior, a third category.

External

External styles are used for the whole, multiple-page website. There is a **separate CSS file with extension .css**, which will simply look something like:

```
/*style.css */
p {
    color: red;
}

a {
    color: blue;
}
```

If this file is saved as “style.css” in the same directory as your HTML page then it can be linked to in the HTML like this:

```
<!DOCTYPE html>
<html>
<head>
  <title>CSS Example</title>
  <link rel="stylesheet" href="style.css">
...

```

Apply!

It would be a good idea to try out the code as we go along, so start a fresh new file with your text-editor and save the blank document as “style.css” in the same directory as your HTML file.

Selectors, Properties, and Values

Whereas HTML has **tags**, CSS has **selectors**. Selectors are the names given to styles in internal and external style sheets. For now, we will be concentrating on **HTML selectors**, which are simply the names of HTML tags and are used to change the style of a specific type of element.

For each selector there are “properties” inside curly brackets, which simply take the form of words such as **color**, **font-weight** or **background-color**. A **value** is given to the property following a **colon** (NOT an “equals” sign) and **semi-colons** separate the properties.

```
body {
  font-size: 14px;
  color: navy;
}
```

This will apply the given values to the **font-size** and **color** properties to the body selector. So basically, when this is applied to an HTML document, text between the body tags (which is the content of the whole window) will be 14 pixels in size and navy in color.

Lengths and Percentages

There are many property-specific units for values used in CSS, but there are some general units that are used by a number of properties and it is worth familiarizing yourself with these before continuing.

px (such as **font-size: 12px**) is the unit for pixels.

em (such as **font-size: 2em**) is the unit for the calculated size of a font. So “2em”, for example, is two times the current font size.

pt (such as **font-size: 12pt**) is the unit for points, for measurements typically in printed media.

% (such as **width: 80%**) is the unit for... wait for it... percentages.

Other units include **pc** (picas), **cm** (centimeters), **mm** (millimeters) and **in** (inches).

Hey! When a value is **zero**, you do not need to state a unit. For example, if you wanted to specify no border, it would be **border: 0**.

Hey! “**px**” in this case, doesn’t actually necessarily mean pixels - the little squares that make up a computer’s display - all of the time. Modern browsers allow users to zoom in and out of a page so that, even if you specify **font-size: 12px**, or **height: 200px**, for example, although these will be the genuine

specified size on a non-zoomed browser, they will still increase and decrease in size with the user's preference.

Colors

CSS brings **16,777,216** colors to your disposal. They can take the form of a **name**, an **RGB** (red/green/blue) value or a **hex** code.

The following values, to specify full-on as red-as-red-can-be, all produce the same result:

- `red`
- `rgb(255,0,0)`
- `rgb(100%,0%,0%)`
- `#ff0000`
- `#f00`

Predefined color names include `aqua`, `black`, `blue`, `fuchsia`, `gray`, `green`, `lime`, `maroon`, `navy`, `olive`, `orange`, `purple`, `red`, `silver`, `teal`, `white`, and `yellow`. `transparent` is also a valid value.

Hey! With the possible exception of `black` and `white`, color names have limited use in modern, well designed web sites because they are so specific and limiting.

The three values in the RGB value are from 0 to 255, 0 being the lowest level (no red, for example), 255 being the highest level (full red, for example). These values can also be a percentage.

Hexadecimal (previously and more accurately known as “**sexadecimal**”) is a **base-16** number system can also be used as color parameter. The **hex** number is prefixed with a hash character (**#**) and can be three or six digits in length. Basically, the three-digit version is a compressed version of the six-digit (`#ff0000` becomes `#f00`, `#cc9966` becomes `#c96`, etc.). The three-digit version is easier to decipher (the first digit, like the first value in RGB, is red, the second green and the third blue) but the six-digit version gives you more control over the exact color.

Hey! CSS3, the latest version of CSS, also allows you to define HSL colors - hue, saturation and lightness.

color and background-color

Colors can be applied by using `color` and `background-color` (note that this must be the American English “color” and not “colour”).

A blue background and yellow text could look like this:

```
h1 {
  color: yellow;
  background-color: blue;
}
```

These colors might be a little too harsh, so you could change the code of your CSS file for slightly different shades:

```
body {
  font-size: 14px;
  color: navy;
}
```

```
h1 {  
  color: #ffc;  
  background-color: #009;  
}
```

Save the CSS file and refresh your browser. You will see the colors of the first heading (the `h1` element) have changed to yellow and blue. You can apply the `color` and `background-color` properties to most HTML elements, including `body`, which will change the colors of the page and everything in it.

Text

You can alter the size and shape of the text on a web page with a range of properties.

font-family

This is the font itself, such as Times New Roman, Arial, or Verdana.

The user's browser has to be able to find the font you specify, which, in most cases, means it needs to be on **their computer** so there is little point in using obscure fonts that are only sitting on **your computer**. There are a select few "safe" fonts (the most commonly used are Arial, Verdana and Times New Roman), but you can specify more than one font, **separated by commas**. The purpose of this is that if the user does not have the first font you specify, the browser will go through the list until it finds one it does have. This is useful because different computers sometimes have different fonts installed. So `font-family: arial, helvetica, serif`, will look for the Arial font first and, if the browser can't find it, it will search for Helvetica, and then a common serif font.

Hey! If the name of a font is more than one word, it should be put in quotation marks, such as `font-family: "Times New Roman"`.

font-size

The size of the font. Be careful with this - text such as headings should not just be an HTML paragraph (`p`) in a large font - you should still use headings (`h1`, `h2` etc.) even though, in practice, you could make the font-size of a paragraph larger than that of a heading (not recommended for sensible people).

font-weight

This state whether the text is bold or not. Most commonly this is used as `font-weight: bold` or `font-weight: normal` but other values are `bolder`, `lighter`, `100`, `200`, `300`, `400` (same as `normal`), `500`, `600`, `700` (same as `bold`), `800` or `900`.

font-style

This state whether the text is italic or not. It can be `font-style: italic` or `font-style: normal`.

text-decoration

This state whether the text has got a line running under, over, or through it.

- `text-decoration: underline`, does what you would expect.
- `text-decoration: overline` places a line above the text.
- `text-decoration: line-through` puts a line through the text ("strike-through").

This property is usually used to decorate links and you can specify no underline with `text-decoration: none`.

Hey! Underlines should only really be used for links. They are a commonly understood web convention that has lead users to generally expect underlined text to be a link.

text-transform

This will change the case of the text.

- **text-transform: capitalize** turns the first letter of every word into uppercase.
- **text-transform: uppercase** turns everything into uppercase.
- **text-transform: lowercase** turns everything into lowercase.
- **text-transform: none** I'll leave for you to work out.

So, a few of these things used together might look like this:

```
body {
  font-family: arial, helvetica, sans-serif;
  font-size: 14px;
}

h1 {
  font-size: 2em;
}

h2 {
  font-size: 1.5em;
}

a {
  text-decoration: none;
}

strong {
  font-style: italic;
  text-transform: uppercase;
}
```

Text spacing

To space out the text on a page: **letter-spacing** and **word-spacing** properties are for spacing between letters or words. The value can be a length or **normal**.

The **line-height** property sets the height of the lines in an element, such as a paragraph, without adjusting the size of the font. It can be a number (which specifies a multiple of the font size, so "2" will be two times the font size, for example), a length, a percentage, or normal.

The **text-align** property will align the text inside an element to left, right, center, or justify.

The **text-indent** property will indent the first line of a paragraph, for example, to a given length or percentage. This is a style traditionally used in print, but rarely in digital media such as the web.

```
p {
  letter-spacing: 0.5em;
  word-spacing: 2em;
  line-height: 1.5;
  text-align: center;
}
```

Margins and Padding

margin and **padding** are the two most commonly used properties for spacing-out elements. A margin is the space **outside** something, whereas padding is the space **inside** something.

Change the CSS code for **h2** to the following:

```
h2 {  
  font-size: 1.5em;  
  background-color: #ccc;  
  margin: 20px;  
  padding: 40px;  
}
```

This leaves a 20-pixel width space around the secondary header and the header itself is fat from the 40-pixel width padding.

The four sides of an element can also be set individually. **margin-top**, **margin-right**, **margin-bottom**, **margin-left**, **padding-top**, **padding-right**, **padding-bottom** and **padding-left** are the self-explanatory properties you can use.

The Box Model

Margins, padding and borders are all part of what's known as **the Box Model**. The Box Model works like this: in the middle you have the content area (let's say an image), surrounding that you have the padding, surrounding that you have the border and surrounding that you have the margin. It can be visually represented like this:



You don't have to use all of these, but it can be helpful to remember that the box model can be applied to every element on the page, and that's a powerful thing!

Borders

Borders can be applied to most HTML elements within the body. To make a border around an element, all you need is **border-style**. The values can be **solid**, **dotted**, **dashed**, **double**, **groove**, **ridge**, **inset** and **outset**.

`border-width` sets the width of the border, most commonly using pixels as a value. There are also properties for `border-top-width`, `border-right-width`, `border-bottom-width` and `border-left-width`.

Finally, `border-color` sets the color.

Add the following code to the CSS file:

```
h2 {  
  border-style: dashed;  
  border-width: 3px;  
  border-left-width: 10px;  
  border-right-width: 10px;  
  border-color: red;  
}
```

This will make a red dashed border around all HTML secondary headers (the `h2` element) that is 3 pixels wide on the top and bottom and 10 pixels wide on the left and right (these having over-ridden the 3 pixel wide width of the entire border).

Class and ID Selectors

Till now, we looked solely at **HTML selectors** - those that represent an HTML **tag**. You can also define your own selectors in the form of **class** and **ID** selectors. The benefit of this is that you can have the same HTML element, but present it differently depending on its class or ID.

In the CSS, a class selector is a name preceded by a **full stop** (".") and an ID selector is a name preceded by a **hash character** ("#").

So the CSS might look something like:

```
#top {  
  background-color: #ccc;  
  padding: 20px  
}  
  
.intro {  
  color: red;  
  font-weight: bold;  
}
```

The HTML refers to the CSS by using the attributes `id` and `class`. It could look something like this:

```
<div id="top">  
  
<h1>Chocolate curry</h1>  
  
<p class="intro">This is my recipe for making curry purely with chocolate</p>  
  
<p class="intro">Mmm mm mmmmm</p>  
  
</div>
```

The difference between an ID and a class is that an **ID** can be used to **identify one element**, whereas a **class** can be used to **identify more than one**.

You can also apply a selector to a specific HTML element by simply stating the HTML selector first, so `p.jam { /* whatever */ }` will only be applied to **paragraph** elements that have the class "jam".

Grouping and Nesting

Two ways that you can simplify your code - both HTML and CSS - and make it easier to manage.

Grouping

You can give the same properties to a number of selectors without having to repeat them. For example, if you have something like:

```
h2 {
  color: red;
}

.thisOtherClass {
  color: red;
}

.yetAnotherClass {
  color: red;
}
```

You can simply separate selectors with **commas** in one line and apply the same properties to them all so, making the above:

```
h2, .thisOtherClass, .yetAnotherClass {
  color: red;
}
```

Nesting

If the CSS is structured well, there shouldn't be a need to use many class or ID selectors. This is because you can specify properties to selectors **within** other selectors.

For example:

```
#top {
  background-color: #ccc;
  padding: 1em;
}

#top h1 {
  color: #ff0;
}

#top p {
```

```
color: red;
font-weight: bold;
}
```

This removes the need for classes or IDs on the `p` and `h1` tags if it is applied to HTML that looks something like this:

```
<div id="top">
  <h1>Chocolate curry</h1>
  <p>This is my recipe for making curry purely with chocolate</p>
  <p>Mmm mm mmmmm</p>
</div>
```

This is because, by separating selectors with spaces, we are saying “`h1` inside ID `top` is colour `#ff0`” and “`p` inside ID `top` is `red` and `bold`”.

Shorthand Properties

Some CSS properties allow a string of values, replacing the need for a number of properties. These are represented by values separated by spaces.

Margins and Padding

```
p {
  margin-top: 1px;
  margin-right: 5px;
  margin-bottom: 10px;
  margin-left: 20px;
}
```

Can be summed up as:

```
p {
  margin: 1px 5px 10px 20px;
}
```

By stating just two values (such as `padding: 1em 10em;`), the **first value will be the top and bottom** and the **second value will be the right and left**.

Borders

`border-width` can be used in exactly the same way as `margin` and `padding`. You can also combine `border-width`, `border-color`, and `border-style` with the `border` property. So:

```
p {
  border-width: 1px;
  border-color: red;
  border-style: solid;
}
```

Can be simplified as:

```
p {
  border: 1px red solid;
}
```

The width/color/style combination can also be applied to `border-top`, `border-right` etc.

Fonts

Font-related properties can also be gathered together with the `font` property:

```
p { font: italic bold 12px/2 courier; }
```

This combines `font-style`, `font-weight`, `font-size`, `line-height`, and `font-family`.

So, to put it all together, try this code:

```
p {
  font: 14px/1.5 "Times New Roman", times, serif;
  padding: 30px 10px;
  border: 1px black solid;
  border-width: 1px 5px 5px 1px;
  border-color: red green blue yellow;
  margin: 10px 50px;
}
```

Lovely, isn't it? ☺

Background Images

Used in a very different way to the `img` HTML element, CSS background images are a powerful way to add detailed presentation to a page.

To jump in at the deep end, the shorthand property `background` can deal with the entire basic background image manipulation aspects.

```
body {
  background: white url(http://www.htmldog.com/images/bg.gif) no-repeat top right;
}
```

This amalgamates these properties:

- `background-color`, which we have come across before.
- `background-image`, which is the location of the image itself.
- `background-repeat`, which is how the image repeats itself. Its value can be:
 - `repeat`, the equivalent of a "tile" effect across the whole background,
 - `repeat-y`, repeating on the y-axis, above and below,
 - `repeat-x` (repeating on the x-axis, side-by-side), or
 - `no-repeat` (which shows just one instance of the image).
- `background-position`, which can be `top`, `center`, `bottom`, `left`, `right`, a length, or a percentage, or any sensible combination, such as `top right`.

Hey!

Background-images can be used in most HTML elements - not just for the whole page (body) and can be used for simple but effective results. As an example, background images are used on this web site as the bullets in lists, as the magnifying glass in the search box, and as the icons in the top left corner of some notes, such as this one.

Display

A key trick to the manipulation of HTML elements is understanding that there's nothing at all special about how most of them work. Most pages could be made up from just a few tags that can be styled any way you choose. The browser's default visual representation of most HTML elements consists of varying font styles, margins, padding and essentially, **display types**.

The most fundamental types of display are **inline**, **block** and **none** and they can be manipulated with the `display` property and the shockingly surprising values `inline`, `block` and `none`.

Inline

inline does just what it says - boxes that are displayed inline follow the flow of a line. Anchor (links) and emphasis are examples of elements that are displayed inline by default.

The following code, for example, will cause all list items in a list to appear next to each other in one continuous line rather than each one having its own line:

```
li { display: inline }
```

Block

block makes a box standalone, fitting the entire width of its containing box, with an effective line break before and after it. Unlike inline boxes, block boxes allow greater manipulation of height, margins, and padding. Heading and paragraph elements are examples of elements that are displayed this way by default in browsers.

The next example will make all links in “nav” large clickable blocks:

```
#navigation a {  
  display: block;  
  padding: 20px 10px;  
}
```

Hey! **display: inline-block** will keep a box inline but lend the greater formatting flexibility of block boxes, allowing margin to the right and left of the box, for example.

None

none, well, doesn't display a box at all, which may sound pretty useless but can be used to good effect with dynamic effects, such as switching extended information on and off at the click of a link, or in alternative stylesheets.

The following code, for example, could be used in a print stylesheet to basically “turn off” the display of elements such as navigation that would be useless in that situation:

```
#navigation, #related_links { display: none }
```

Hey! **display: none** and **visibility: hidden** vary in that, **display: none** takes the element's box completely out of play, whereas **visibility: hidden** keeps the box and its flow in place without visually representing its contents. For example, if the second paragraph of 3 were set to **display: none**, the first paragraph would run straight into the third whereas if it were set to **visibility: hidden**, there would be a gap where the paragraph should be.

Page Layout (Positioning)

In the olden days, pre-hominid apes used HTML **tables** to layout web pages. Hilarious, right? But CSS, that 2001 soon came along and changed all of that.

Layout with CSS is easy. You just take a chunk of your page and shove it wherever you choose. You can place these chunks **absolutely** or **relative** to another chunk.

Positioning

The `position` property is used to define whether a box is absolute, relative, static or fixed:

- `static` is the default value and renders a box in the normal order of things, as they appear in the HTML.
- `relative` is much like `static` but the box can be offset from its original position (normal flow) with the properties `top`, `right`, `bottom` and `left`.
- `absolute` pulls a box out of the normal flow of the HTML and delivers it to a world all of its own. In this crazy little world, the absolute box can be placed anywhere on the page using `top`, `right`, `bottom` and `left` **relative to its parent element**.
- `fixed` behaves like `absolute`, but it will absolutely position a box in reference to the browser window as opposed to the web page, so fixed boxes should stay exactly where they are on the screen even when the page is **scrolled**.

Layout using absolute positioning

You can create a traditional two-column layout with absolute positioning if you have something like the following HTML:

```
<div id="navigation">
  <ul>
    <li><a href="this.html">This</a></li>
    <li><a href="that.html">That</a></li>
    <li><a href="theOther.html">The Other</a></li>
  </ul>
</div>

<div id="content">
  <h1>Ra ra banjo banjo</h1>
  <p>Welcome to the city of gestures</p>
  <p>(Another greeting)</p>
</div>
```

And if you apply the following CSS:

```
#navigation {
  position: absolute;
  top: 0;
  left: 0;
  width: 200px;
}

#content {
  margin-left: 200px;
}
```

You will see that this will set the navigation bar to the left and set the width to 200 pixels. Because the navigation is absolutely positioned, it has nothing to do with the flow of the rest of the page so all that is needed is to set the left margin of the content area to be equal to the width of the navigation bar.

How stupidly easy! And you aren't limited to this two-column approach. With clever positioning, you can arrange as many blocks as you like. If you wanted to add a third column, for example, you could add a "navigation2" chunk to the HTML and change the CSS to:

```
#navigation {
  position: absolute;
  top: 0;
  left: 0;
  width: 200px;
}

#navigation2 {
  position: absolute;
  top: 0;
  right: 0;
  width: 200px;
}

#content {
  margin: 0 200px; /* setting top and bottom margin to 0 and right and left margin to 200px */
}
```

Hey!

The only downside to absolutely positioned boxes is that because they live in a world of their own, there is no way of accurately determining where they end. If you were to use the examples above and all of your pages had small navigation bars and large content areas, you would be okay, but, especially when using relative values for widths and sizes, you often have to abandon any hope of placing anything, such as footer, below these boxes. If you wanted to do such a thing, one way would be to **float** your chunks, rather than absolutely positioning them.

Floating

Floating a box will shift it to the right or left of a line, with surrounding content flowing around it. Floating is normally used to shift around smaller chunks within a page, such as pushing a navigation link to the right of a container, but it can also be used with bigger chunks, such as navigation columns.

Using **float**, you can **float: left** or **float: right**.

Working with the same HTML as above, you could apply the following CSS:

```
#navigation {
  float: left;
  width: 200px;
}

#navigation2 {
  float: right;
  width: 200px;
}

#content {
```

```
margin: 0 200px;  
}
```

Then, if you do not want the next box to wrap around the floating objects, you can apply the **clear** property:

clear: left will clear left floated boxes

clear: right will clear right floated boxes

clear: both will clear both left and right floated boxes.

So if, for example, you wanted a footer in your page, you could add a chunk of HTML...

```
<div id="footer">  
  <p>Footer! Hoorah!</p>  
</div>
```

...and then add the following CSS:

```
#footer {  
  clear: both;  
}
```

And there you have it. A footer that will appear underneath all columns, regardless of the length of any of them.

Hey!

This has been a general introduction to positioning and floating, with emphasis on the larger “chunks” of a page, but remember, these methods can be applied to any box within those boxes, too. With a combination of positioning, floating, margins, padding and borders, you should be able to represent any web design.

Pseudo Classes

Pseudo classes are bolted on to selectors to specify a state or relation to the selector. They take the form of **selector:pseudo_class { property: value; }**, simply with a colon in between the selector and the pseudo class.

Links

link, targeting **unvisited links**, and **visited**, targeting, you guessed it, **visited links**, are the most basic pseudo classes.

The following will apply colors to all links in a page depending on whether the user has visited that page before or not:

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

Dynamic Pseudo Classes

Also commonly used for links, the dynamic pseudo classes can be used to apply styles when something happens to something.

- **active** is for when something activated by the user, such as when a link is clicked on.
- **hover** is for a when something is passed over by an input from the user, such as when a cursor moves over a link.
- **focus** is for when something gains focus, that is when it is selected by, or is ready for, keyboard input.

Hey! **focus** is most often used on **form elements** but can be used for **links**. Although most users will navigate around and between pages using a pointing device such as a mouse using a tab key and they will gain focus one at a time.

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

a:hover {
  text-decoration: none;
  color: blue;
  background-color: yellow;
}

input:focus, textarea:focus {
  background: #eee;
}
```

focus isn't supported by some of the older browsers so be careful not to use it for anything vital.

First Children

Finally, there is the **first-child** pseudo class. This will target something only if it is the very first descendant of an element. So, in the following HTML...

```
<body>
  <p>I'm the body's first paragraph child. I rule. Obey me.</p>
  <p>I resent you.</p>
...
```

...if you only want to style the **first** paragraph, you could use the following CSS:

```
p:first-child {
  font-weight: bold;
  font-size: 40px;
}
```

CSS3 has also delivered a whole new set of pseudo classes: **last-child**, **target**, **first-of-type**, and more.

Pseudo Elements

Pseudo elements suck on to selectors to apply styles much like **pseudo classes**, taking the form of `selector::pseudoelement { property: value; }`.

There are four of the suckers. (OOPS!)

First Letters and First Lines

The **first-letter** pseudo element applies to the first letter inside a box and **first-line** to the top-most displayed line in a box.

As an example, you could create drop caps and a bold first-line for paragraphs with something like this:

```
p {
  font-size: 12px;
}

p::first-letter {
  font-size: 24px;
  float: left;
}

p::first-line {
  font-weight: bold;
}
```

Hey!

This is CSS 3 specs that suggest pseudo elements to use two colons as `p::first-line` to differentiate them with pseudo classes. But some older browsers only understands `p:first-line` syntax for pseudo-elements, just not listen it, who cares about oldies anyway, oh I mean browsers grown old.

Before and After Content

The **before** and **after** pseudo elements are used in conjunction with the **content** property to place content either side of a box without touching the HTML.

What? Content in my **CSS**? But I thought **HTML** was for content.

Well, it is. So use carefully. Look at it like this: You are borrowing content to use solely as presentation, such as using “!” because it looks pretty. Not because you actually want to exclaim anything.

The value of the **content** property can be **open-quote**, **close-quote**, any **string** enclosed in quotation marks, or any **image**, using `url(imagename)`.

```
blockquote:before {
  content: open-quote;
}
```

```
blockquote:after {
  content: close-quote;
}
```

```
li:before {
```

```
    content: "POW! ";  
}  
  
p:before {  
    content: url(images/jam.jpg);  
}
```

The `content` property effectively creates another box to play with so you can also add styles to the “presentational content”:

```
li:before {  
    content: "POW! ";  
    background: red;  
    color: #fc0;  
}
```

Javascript: Client Side Scripting

JavaScript is a scripting language produced by Netscape for use within HTML Web pages. It is a lightweight, interpreted programming language supported by all modern browsers.

JavaScript Reserved Words

In JavaScript you cannot use these reserved words as variables, labels, or function names:

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

Words marked with* are new in ECMAScript5

JavaScript DataTypes

JavaScript allows you to work with three primitive data types:

- Numbers eg. 123, 120.50 etc.
- Strings of text e.g. "This text string" etc.
- Boolean e.g. true or false.

JavaScript also defines two trivial data types, **null** and **undefined**, each of which defines only a single value.

JavaScript Variables

JavaScript variables are "containers" for storing information as other languages:

```
var a = 5; // Integer
var b = 6;
var c = a + b;
```

```
var x = "college"; // string
var y = 'college'; // string - also valid
var PI = 3.14; // floating point number
var lastName = "Doe", age = 30, job = "carpenter";
var z = y + 2; // valid statement, results "college2"
```

JavaScript Comments

- JavaScript comments can be used to explain the code, and make the code more readable.
- JavaScript comments can also be used to prevent execution, when testing alternative code.

Single Line Comments

Single line comments start with `//`.

```
<script>
  // Change heading:
  document.getElementById("myH").innerHTML = "My First Page";
  // Change paragraph:
  document.getElementById("myP").innerHTML = "My first paragraph.";
</script>
```

Multi-line Comments

Multi-line comments start with `/*` and end with `*/`.

```
<script>
  /*
  The code below will change
  the heading with id = "myH"
  and the paragraph with id = "myp"
  in my web page:
  */
  document.getElementById("myH").innerHTML = "My First Page";
  document.getElementById("myP").innerHTML = "My first paragraph.";
</script>
```

Operators

Simple answer can be given using expression *4 + 5 is equal to 9*. Here 4 and 5 are called operands and + is called operator. JavaScript language supports following type of operators.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

Let's have a look on all operators one by one.

Let A = 10 and B = 20 then:

The Arithmetic Operators:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by denominator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

The Comparison Operators:

Operator	Description	Example
===	Checks if the values (without type-casting) of two operands are equal or not, if yes then condition becomes true.	'0'===0 is false
!==	Checks if the values (with type checking) of two operands are equal or not, if yes then condition becomes true.	'0'!==0 is true
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	'0'==0 is true
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Logical Operators:

Operator	Description	Example
----------	-------------	---------

&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

The Bitwise Operators:

Assuming, A = 2 and B = 3 then:

Operator	Description	Example
&	Called Bitwise AND operator. It performs a Boolean AND operation on each bit of its integer arguments.	(A & B) is 2 .
	Called Bitwise OR Operator. It performs a Boolean OR operation on each bit of its integer arguments.	(A B) is 3.
^	Called Bitwise XOR Operator. It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both.	(A ^ B) is 1.
~	Called Bitwise NOT Operator. It is a unary operator and operates by reversing all bits in the operand.	(~B) is -4.
<<	Called Bitwise Shift Left Operator. It moves all bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, etc.	(A << 1) is 4.
>>	Called Bitwise Shift Right with Sign Operator. It moves all bits in its first operand to the right by the number of places specified in the second operand. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits. Shifting a value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on.	(A >> 1) is 1.
>>>	Called Bitwise Shift Right with Zero Operator. This operator is just like the >> operator, except that the bits shifted in on the left are always zero,	(A >>> 1) is 1.

The Assignment Operators:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	$C \% = A$ is equivalent to $C = C \% A$

Hey! Same logic applies to Bitwise operators so they will become like $\ll=$, $\gg=$, $\>>=$, $\&=$, $|=$ and $\^=$.

Miscellaneous Operator

The Conditional Operator (? :)

There is an operator called conditional operator. This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. The conditional operator has this syntax:

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

The typeof Operator

The *typeof* is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"

Undefined	"undefined"
Null	"object"

Functions

Functions are reusable blocks of code that carry out a specific task. To execute the code in a function you call it. A function can be passed arguments to use, and a function may return a value to whatever called it.

Syntax

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 [functionName](parameter1, parameter2, parameter3) {  
    code to be executed  
}
```

Hey! [] denotes optional construct.

Function **parameters** are the names listed in the function definition.

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

Inside the function, the arguments are used as **local variables**.

Function can be defined in following ways. Here's a function that adds two numbers:

```
function add(a, b) {  
    return a + b;  
};  
  
OR  
  
var add = function (a, b) {  
    return a + b;  
};
```

a and **b** are the function's parameters, and the value it returns is signified by the return keyword. The **return** keyword also stops execution of the code in the function; nothing after it will be run.

```
var result = add(1, 2); // result is now 3
```

This calls add with the arguments 1 and 2, which, inside add, will be saved in the variables **a** and **b**.

Arrays

Arrays are lists of any kind of data, including other arrays. Each item in the array has an **index**-a number-which can be used to retrieve an **element** from the array.

The indices start at 0; that is, the first element in the array has the index 0, and subsequent elements have incrementally increasing indices, so the last element in the array has an index one less than the length of the array.

In JavaScript, you create an array using the array-literal syntax:

```
var emptyArray = [];  
var shoppingList = ['Milk', 'Bread', 'Beans'];  
var numbers = [1, 2, 3, 4];
```

You retrieve a specific element from an array using square bracket syntax:

```
shoppingList[0];  
Milk
```

Setting specific value:

```
shoppingList[1] = 'Cookies';  
// shoppingList is now ['Milk', 'Cookies', 'Beans']
```

You can find the number of elements in the array using its length property:

```
shoppingList.length;  
3
```

Array Methods

- **valueOf()/toString():** Returns array as a string

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.valueOf();  
document.getElementById("demo").innerHTML = fruits.toString();
```

- **join()**

```
<p id="demo"></p>  
<script>  
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.join(" * ");  
</script>
```

- **Popping and Pushing**

```
shoppingList.push('A new car');  
// shoppingList is now ['Milk', 'Bread', 'Beans', 'A new car']  
shoppingList.pop();  
// shoppingList is back to ['Milk', 'Bread', 'Beans']
```

The **pop()** method returns the string that was "popped out".

The **push()** method returns the new array length.

```
var helloFrom = function (personName) {  
    return "Hello from " + personName;  
}
```

```
var people = ['Tom', 'Yoda', 'Ron'];
```

```
people.push('Bob');  
people.push('Dr Evil');
```

```
people.pop();
```

```
for (var i=0; i < people.length; i++) {
```

```
var greeting = helloFrom(people[i]);
alert(greeting);
}
```

▪ **shift() and unshift():**

- The **shift()** method removes the first element of an array, and "shifts" all other elements one place down
- The **unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift(); // Removes the first element "Banana" from fruits
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon"); // Adds a new element "Lemon" to fruits
```

▪ **splice():**The **splice()** method can be used to add new items to an array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon", "Kiwi") define the new elements to be **added**.

Deleting with splice:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0,1); // Removes the first element of fruits
```

▪ **sort():**

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort(); // Sorts the elements of fruits
fruits.reverse(); // Reverses the order of the elements
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a-b}); //Numeric sort: ascending
points.sort(function(a, b){return b-a}); //descending
```

▪ **concat():**The **concat()** method creates a new array by concatenating two arrays.

```
var myGirls = ["Cecilie", "Lone"];
var myBoys = ["Emil", "Tobias", "Linus"];
var myChildren = myGirls.concat(myBoys); // Concatenates (joins) myGirls and myBoys
```

▪ **slice():**slices out a piece of an array

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(1,3); // citrus = 'orange,Lemon'
```

Objects

JavaScript objects are like a real life objects; they have properties and abilities. A JavaScript object is, in that sense, a collection of named properties and methods - a function. An object can be stored in a variable, and the properties and methods accessed using the dot syntax.

Variables can hold objects, and creating an object is done using a special syntax signified by braces:

```
var jedi = {
  name: "Yoda",
  age: 899,
  talk: function () { alert("another... Sky... walk..."); }
};
```

You can get data back out of an object using the dot syntax:

```
jedi.name; or jedi["name"]
```

```
Yoda
jedi.age; or jedi["age"]
899
jedi.talk();
//produces an alert box
```

You can also reassign properties of an object:

```
jedi.name = "Mace Windu";
```

And add new ones on the fly:

```
jedi.height = "1.75m";
```

Properties can be any kind of data including objects and arrays. Adding an object as a property of another object creates a nested object:

```
var person = {
  age: 122
};

person.name = {
  first: "Jeanne",
  last: "Calment"
};
```

Creating an empty object and adding properties and methods to it is possible too:

```
var dog = {};
dog.bark = function () { alert("Woof!"); };
```

Built-in JavaScript Objects

1. Date Object

The Date object lets us work with dates. A date consists of a year, a month, a week, a day, a minute, a second, and a millisecond. Date objects are created with the new Date() constructor.

There are 4 ways of initiating a date:

```
new Date()
new Date(milliseconds)
new Date(dateString)
new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

Using new Date(), without parameters, creates a new date object with the current date and time:

```
<script>
var d = new Date(); //Or simply Date() function can be used
var d = new Date("October 13, 2014 11:13:00");
var d = new Date(0); //new date with number of millisecond since 1970/01/01
var d = new Date(99,5,24,11,33,30,0); // The 7 numbers specify the year,
month, day, hour, minute, second, and millisecond
var d = new Date(99,5,24); //Omitting last 4 params
document.getElementById("demo").innerHTML = d;
</script>
```

Date Mehtods

```
<script>
    d = new Date();
    document.getElementById("demo").innerHTML = d.toString();
    document.getElementById("demo").innerHTML = d.toUTCString();
    document.getElementById("demo").innerHTML = d.toDateString();
</script>
```

Date get methods

Get methods are used for getting a part of a date. Here are the most common (alphabetically):

Method	Description
getDate()	Get the day as a number (1-31)
getDay()	Get the weekday a number (0-6)
getFullYear()	Get the four digit year (yyyy)
getHours()	Get the hour (0-23)
getMilliseconds()	Get the milliseconds (0-999)
getMinutes()	Get the minutes (0-59)
getMonth()	Get the month (0-11)
getSeconds()	Get the seconds (0-59)
getTime()	Get the time (milliseconds since January 1, 1970)

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.getTime();
document.getElementById("demo").innerHTML = d.getDay();
...Try others
</script>
```

Date set methods

Set methods are used for setting a part of a date. Here are the most common (alphabetically):

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day yyyy.mm.dd)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)

setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

```
<script>
  var d = new Date();
  d.setFullYear(2020, 0, 14);
  d.setDate(20);
  d.setDate(d.getDate() + 50);
  document.getElementById("demo").innerHTML = d;
</script>
```

Parsing Dates

If you have an input value (or any string), you can use the **Date.parse()** method to convert it to milliseconds. **Date.parse()** returns the number of milliseconds between the date and January 1, 1970:

```
<script>
  var msec = Date.parse("March 21, 2012");
  document.getElementById("demo").innerHTML = msec;
</script>
```

JavaScript Strings

A JavaScript string simply stores a series of characters like "Mount Everest". A string can be any text inside quotes (single or double quotes).

```
var answer = "It's alright";
var answer = "He is called 'Johnny'";
var answer = 'He is called "Johnny"';
```

String length

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
var sln = txt.length;
```

Strings Can be Objects

Normally, JavaScript strings are primitive values, created from literals: **var firstName = "John"**
But strings can also be defined as objects with the keyword new: **var firstName = new String("John")**

```
var x = "John";
var y = new String("John");

// type of x will return String
// type of y will return Object
```

Hey! Don't create String objects. They slow down execution speed, and produce nasty side effects.

String Properties and Methods

With JavaScript, methods and properties are also available to primitive values (unlike other languages, where only objects does have), because JavaScript treats primitive values as objects when executing methods and properties.

String Properties

Property	Description
Constructor	Returns the function that created the String object's prototype
Length	Returns the length of a string
Prototype	Allows you to add properties and methods to an object

String Methods

Method	Description
charAt()	Returns the character at the specified index (position)
charCodeAt()	Returns the Unicode of the character at the specified index
concat()	Joins two or more strings, and returns a copy of the joined strings
fromCharCode()	Converts Unicode values to characters
indexOf()	Returns the position of the first found occurrence of a specified value in a string
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string
localeCompare()	Compares two strings in the current locale
match()	Searches a string for a match against a regular expression, and returns the matches
replace()	Searches a string for a value and returns a new string with the value replaced
search()	Searches a string for a value and returns the position of the match
slice()	Extracts a part of a string and returns a new string
split()	Splits a string into an array of substrings
substr()	Extracts a part of a string from a start position through a number of characters
substring()	Extracts a part of a string between two specified positions
toLocaleLowerCase()	Converts a string to lowercase letters, according to the host's locale
toLocaleUpperCase()	Converts a string to uppercase letters, according to the host's locale
toLowerCase()	Converts a string to lowercase letters
toString()	Returns the value of a String object

toUpperCase()	Converts a string to uppercase letters
trim()	Removes whitespace from both ends of a string
valueOf()	Returns the primitive value of a String object

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate"); //pos = 7
var pos = str.lastIndexOf("locate"); //pos = 21
var pos = str.search("locate"); //pos = 7

var str = "Apple, Banana, Kiwi";
var res = str.slice(7,13); //res = Banana
var res = str.substring(7,13); //res = Banana
var res = str.substr(7,6); //res = Banana, second argument: length of extracted part
var res = str.slice(-12,-6); //res = Banana, counts from end of string
var res = str.slice(7);
```

Hey! Negative positions does not work in Internet Explorer 8 and earlier.

Hey! substring() cannot accept negative indexes, that is how it differs from slice()

```
str = "Please visit Microsoft!";
var n = str.replace("Microsoft","W3Schools");

var text1 = "Hello World!"; // String
var text2 = text1.toUpperCase(); // text2 is text1 converted to upper
var text2 = text1.toLowerCase();

var text1 = "Hello";
var text2 = "World";
var text3 = text1.concat(" ",text2);
var text = "Hello" + " " + "World!";
var text = "Hello".concat(" ","World!");

var str = "HELLO WORLD";
//Safe methods
str.charAt(0); // returns H
str.charCodeAt(0); // returns 72
```

//Unsafe way

```
str[0];
```

This is **unsafe** and **unpredictable**:

- It does not work in all browsers (not in IE5, IE6, IE7)
- It makes strings look like arrays (but they are not)
- str[0] = "H" does not give an error (but does not work)

Splitting strings (Coverting to array)

```
var txt = "a,b,c,d,e"; // String
txt.split(","); // Split on commas
```

```
txt.split(" ");           // Split on spaces
txt.split("|");          // Split on pipe

var txt = "Hello";       // String
txt.split("");           // Split in characters
for (var i = 0; i < arr.length; i++) {
    text += arr[i] + "<br>"
}
document.getElementById("demo").innerHTML = text;
```

JavaScript Numbers

Numbers can be written with, or without, decimals.

```
var x = 34.00; // A number with decimals
var y = 34;    // A number without decimals
```

Extra large or extra small numbers can be written with scientific (exponent) notation:

Example

```
var x = 123e5; // 12300000
var y = 123e-5; // 0.00123
```

JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc. JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard. This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:

Value (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

Precision

Integers (numbers without a period or exponent notation) are considered accurate up to 15 digits.

```
var x = 999999999999999; // x will be 999999999999999
var y = 999999999999999; // y will be 10000000000000000
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

Example

```
var x = 0.2 + 0.1; // x will be 0.30000000000000004
```

Hexadecimal

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

Example

```
var x = 0xFF; // x will be 255
```

By default, JavaScript displays numbers as base 10 decimals. But you can use the `toString()` method to output numbers as base 16 (hex), base 8 (octal), or base 2 (binary).

```
var myNumber = 128;
myNumber.toString(16); // returns 80
myNumber.toString(8); // returns 200
myNumber.toString(2); // returns 10000000
```

Infinity

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

```
var myNumber = 2;
while (myNumber != Infinity) { // Execute until Infinity
  myNumber = myNumber * myNumber;
}
```

Division by 0 (zero) also generates Infinity:

```
var x = 2 / 0; // x will be Infinity
var y = -2 / 0; // y will be -Infinity
```

NaN - Not a Number

NaN is a JavaScript reserved word indicating that a value is not a number.

Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

```
var x = 100 / "Apple"; // x will be NaN (Not a Number)
var x = 100 / "10"; // x will be 10, ya its true
```

You can use the global JavaScript function isNaN() to find out if a value is a number.

```
var x = 100 / "Apple";
isNaN(x); // returns true because x is Not a Number
```

Number Properties and Methods

JavaScript treats primitive values as objects when executing methods and properties.

Number Properties

Property	Description
MAX_VALUE	Returns the largest number possible in JavaScript
MIN_VALUE	Returns the smallest number possible in JavaScript
NEGATIVE_INFINITY	Represents negative infinity (returned on overflow)
NaN	Represents a "Not-a-Number" value
POSITIVE_INFINITY	Represents infinity (returned on overflow)

```
var x = Number.MAX_VALUE;
```

Number properties belongs to the JavaScript's number object wrapper called **Number**.

These properties can only be accessed as `Number.MAX_VALUE`.

Using `myNumber.MAX_VALUE`, where `myNumber` is a variable, expression, or value, will return undefined:

Number Methods

Methods specific to numbers are:

Method	Description
<code>toString()</code>	Returns a number as a string
<code>toExponential()</code>	Returns a string, with a number rounded and written using exponential notation.
<code>toFixed()</code>	Returns a string, with a number rounded and written with a specified number of decimals.
<code>toPrecision()</code>	Returns a string, with a number written with a specified length
<code>valueOf()</code>	Returns a number as a number

```
var x = 123;
x.toString();           // returns 123 from variable x
(123).toString();      // returns 123 from literal 123
(100 + 23).toString(); // returns 123 from expression 100 + 23
```

```
var x = 9.656;
x.toExponential(2);    // returns 9.66e+0
x.toExponential(4);    // returns 9.6560e+0
x.toExponential(6);    // returns 9.656000e+0
```

```
var x = 9.656;
x.toFixed(0);           // returns 10
x.toFixed(2);           // returns 9.66
x.toFixed(4);           // returns 9.6560
x.toFixed(6);           // returns 9.656000
```

```
var x = 9.656;
x.toPrecision();        // returns 9.656
x.toPrecision(2);       // returns 9.7
x.toPrecision(4);       // returns 9.656
x.toPrecision(6);       // returns 9.65600
```

Converting Variables to Numbers

There are 3 JavaScript functions that can be used to convert variables to numbers:

- The `Number()` method
- The `parseInt()` method
- The `parseFloat()` method

These methods are not **number** methods, but **global** JavaScript methods.

The `Number()` Method

```
x = true;
Number(x);      // returns 1
x = false;
Number(x);      // returns 0
x = new Date();
Number(x);      // returns 1404568027739
x = "10"
Number(x);      // returns 10
x = "10 20"
Number(x);      // returns NaN
```

The parseInt() Method

```
parseInt("10");      // returns 10
parseInt("10.33");   // returns 10
parseInt("10 20 30"); // returns 10
parseInt("10 years"); // returns 10
parseInt("years 10"); // returns NaN
```

The parseFloat() Method

```
parseFloat("10");      // returns 10
parseFloat("10.33");   // returns 10.33
parseFloat("10 20 30"); // returns 10
parseFloat("10 years"); // returns 10
parseFloat("years 10"); // returns NaN
```

HTML, CSS and JavaScript together

Mostly, JavaScript runs in your web browser alongside HTML and CSS, and can be added to any web page using a **script** tag. The **script** element can either contain JavaScript directly (**internal**) or link to an external resource via a **src** attribute (**external**).

A browser then runs JavaScript line-by-line, starting at the top of the file or **script** element and finishing at the bottom (unless you tell it to go elsewhere).

Internal

You can just put the JavaScript inside a **script** element:

```
<script>
  alert("Hello, world.");
</script>
```

External

An external JavaScript resource is a **text file** with a **.js** extension, just like an external CSS resource with a **.css** extension.

To add a JavaScript file to your page, you just need to use a script tag with a **src** attribute pointing to the file. So, if your file was called **script.js** and sat in the same directory as your HTML file, our **script** element would look like this:

```
<script src="script.js"></script>
```

Hey!

You might also come across another way on your view-source travels: inline. This involves event attributes inside HTML tags that look something like

```
<a href="somewhere.html" onclick="alert('Noooooo!');">Click me</a>.
```

Just move along and pretend you haven't witnessed this aberration. We really, really, really want to separate our technologies so it's preferable to avoid this approach.

Console

In a modern browser you'll find some developer tools - often you can right click on a page, then click "inspect element" to bring them up. Find the console and you'll be able to type JavaScript, hit enter and have it run immediately.

Conditionals

```
if (43 < 2) {  
    // Run the code in here  
} else {  
    // Run a different bit of code  
}
```

Looping

while

```
var i = 1;  
while (i < 10) {  
    alert(i);  
    i = i + 1;  
}  
// i is now 10
```

for

```
for (var i = 1; i < 10; i++) {  
    alert(i);  
}
```

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

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


for-in

```
var person = {fname:"John", lname:"Doe", age:25};

var text = "";
var x;
for (x in person) {
    text += person[x];
}
```

Switch Statement

Use the switch statement to select one of many blocks of code to be executed.

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    case 2:
        day = "Tuesday";
        break;
    case 3:
        day = "Wednesday";
        break;
    case 4:
        day = "Thursday";
        break;
    case 5:
        day = "Friday";
        break;
    case 6:
        day = "Saturday";
        break;
}
```

Common Code and Fall-Through a default

```
switch (new Date().getDay()) {
    case 1:
    case 2:
    case 3:
    default:
        text = "Looking forward to the Weekend";
        break;
    case 4:
    case 5:
        text = "Soon it is Weekend";
        break;
    case 0:
    case 6:
        text = "It is Weekend";
}
```

JavaScript Errors - Throw and Try to Catch

The **try** statement lets you test a block of code for errors.

The **catch** statement lets you handle the error.

The **throw** statement lets you create custom errors.

The **finally** statement lets you execute code, after try and catch, regardless of the result.

```
<!DOCTYPE html>
<html>
<body>
  <p id="demo"></p>
  <script>
    try {
      ahgkjhklert("Welcome guest!");
    }
    catch(err) {
      document.getElementById("demo").innerHTML = err.message;
    }
  </script>
</body>
</html>
```

The throw Statement

The **throw** statement allows you to create a custom error.

The technical term for this is: **throw an exception**.

The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big"; // throw a text
throw 500;      // throw a number
```

If you use **throw** together with **try** and **catch**, you can control program flow and generate custom error messages.

Input Validation Example

This example examines input. If the value is wrong, an exception (err) is thrown.

The exception (err) is caught by the catch statement and a custom error message is displayed:

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

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="message"></p>

<script>
function myFunction() {
```

```
var message, x
message = document.getElementById("message");
message.innerHTML = "";
x = document.getElementById("demo").value;
try {
    if(x == "") throw "is Empty";
    if(isNaN(x)) throw "not a number";
    if(x > 10) throw "too high";
    if(x < 5) throw "too low";
}
catch(err) {
    message.innerHTML = "Input " + err;
}
}
</script>

</body>
</html>
```

The finally Statement

The **finally** statement lets you execute code, after try and catch, regardless of the result:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}
```

```
function myFunction()
    var message, x;
    message = document.getElementById("message");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "") throw "Empty";
        if(isNaN(x)) throw "Not a number";
        if(x > 10) throw "Too high";
        if(x < 5) throw "Too low";
    }
    catch(err) {
        message.innerHTML = "Error: " + err + ".";
    }
    finally {
        document.getElementById("demo").value = "";
    }
}
```

```
}  
}
```

Basic Form validation

JavaScript provides a way to validate form's data on the client's computer before sending it to the web server for further manipulation. Form validation generally performs two functions.

- **Basic Validation** - First of all, the form must be checked to make sure data was entered into each form field that required it. This would need just loop through each field in the form and check for data.
- **Data Format Validation** - Secondly, the data that is entered must be checked for correct form and value. This would need to put more logic to test correctness of data.

We will take an example to understand the process of validation. Here is the simple form to proceed:

```
<html>  
<head>  
<title>Form Validation</title>  
<script type="text/javascript">  
    // Form validation code will come here.  
</script>  
</head>  
<body>  
    <form action="/cgi-bin/test.cgi" name="myForm"  
        onsubmit="return(validate());">  
        <table cellspacing="2" cellpadding="2" border="1">  
            <tr>  
                <td align="right">Name</td>  
                <td><input type="text" name="Name" /></td>  
            </tr>  
            <tr>  
                <td align="right">EMail</td>  
                <td><input type="text" name="EMail" /></td>  
            </tr>  
            <tr>  
                <td align="right">Zip Code</td>  
                <td><input type="text" name="Zip" /></td>  
            </tr>  
            <tr>  
                <td align="right">Country</td>  
                <td>  
                <select name="Country">  
                    <option value="-1" selected>[choose yours]</option>  
                    <option value="1">USA</option>  
                    <option value="2">UK</option>  
                    <option value="3">INDIA</option>  
                </select>  
                </td>  
            </tr>  
            <tr>  
                <td align="right"></td>  
                <td><input type="submit" value="Submit" /></td>  
            </tr>  
        </table>  
    </form>  
</body>  
</html>
```

Basic Form Validation:

First we will show how to do a basic form validation. In the above form we are calling validate()function to validate data when onsubmit event is occurring. Following is the implementation of this validate() function:

```
<script type="text/javascript">
// Form validation code will come here.
function validate()
{
    if( document.myForm.Name.value == "" )
    {
        alert( "Please provide your name!" );
        document.myForm.Name.focus() ;
        return false;
    }
    if( document.myForm.EMail.value == "" )
    {
        alert( "Please provide your Email!" );
        document.myForm.EMail.focus() ;
        return false;
    }
    if( document.myForm.Zip.value == "" ||
        isNaN( document.myForm.Zip.value ) ||
        document.myForm.Zip.value.length != 5 )
    {
        alert( "Please provide a zip in the format #####." );
        document.myForm.Zip.focus() ;
        return false;
    }
    if( document.myForm.Country.value == "-1" )
    {
        alert( "Please provide your country!" );
        return false;
    }
    return( true );
}
</script>
```

Data Format Validation

Now we will see how we can validate our entered form data before submitting it to the web server.

This example shows how to validate an entered email address which means email address must contain at least an @ sign and a dot (.). Also, the @ must not be the first character of the email address, and the last dot must at least be one character after the @ sign:

```
<script type="text/javascript">
function validateEmail()
{
    var emailID = document.myForm.EMail.value;
    atpos = emailID.indexOf("@");
    dotpos = emailID.lastIndexOf(".");
    if (atpos < 1 || ( dotpos - atpos < 2 ))
    {
        alert("Please enter correct email ID")
        document.myForm.EMail.focus() ;
        return false;
    }
    return( true );
}
}
```

```
</script>
```

JavaScript Popup Boxes

JavaScript has three kinds of popup boxes: **Alert** box, **Confirm** box, and **Prompt** box.

Alert Box

An alert box is often used if you want to make sure information comes through to the user. When an alert box pops up, the user will have to click "OK" to proceed.

Syntax

```
window.alert("sometext");
```

The **window.alert** method can be written without the window prefix.

Example

```
alert("I am an alert box!");
```

Confirm Box

A confirm box is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

Syntax

```
window.confirm("sometext");
```

The **window.confirm()** method can be written without the window prefix.

Example

```
var r = confirm("Are you sure?");
if (r == true) {
    x = "You pressed OK!";
} else {
    x = "You pressed Cancel!";
}
```

Prompt Box

A prompt box is often used if you want the user to input a value before entering a page. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Syntax

```
window.prompt("sometext", "defaultText");
```

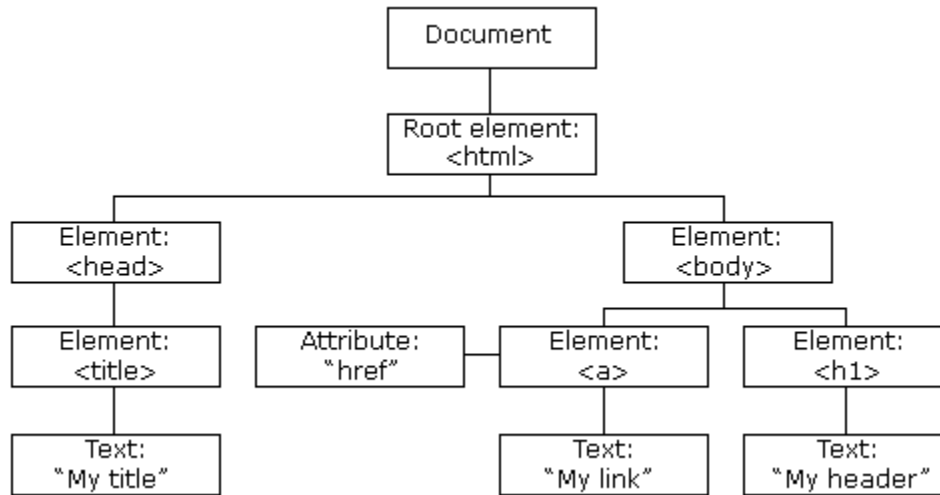
The **window.prompt()** method can be written without the window prefix.

Example

```
var person = prompt("Please enter your name", "Harry Potter");
if (person != null) {
    document.getElementById("demo").innerHTML =
        "Hello " + person + "! How are you today?";
}
```

The Document Object Model (DOM)

When a web page is loaded, the browser creates a Document Object Model of the page. The HTML DOM model is constructed as a tree of Objects:



DOM represents the internals of the page as the browser sees it, and allows the developer to alter it with JavaScript. This is a way to manipulate the structure and style of an HTML page.

If you'd like to have a look at the DOM for a page, open up the developer tools in your browser and look for the "elements" pane. It's a great insight into how the browser thinks, and in most browsers you can remove and modify elements directly. Give it a try!

Trees and Branches

HTML is an XML-like structure in that the elements form a structure of parents' nodes with children, like the branches of a tree. There is one root element (**html**) with branches like **head** and **body**, each with their own branches. For this reason, the DOM is also called the **DOM tree**.

Modifying the DOM, by picking an element and changing something about it, is something done often in JavaScript. To access the DOM from JavaScript, the **document** object is used. It's provided by the browser and allows code on the page to interact with the content.

Getting an Element

The first thing to know is how to get an element. There are a number of ways of doing it, and browsers support different ones. Starting with the best supported we'll move through to the latest, and most useful, versions.

By ID

`document.getElementById` is a method for getting hold of an element - unsurprisingly - by its ID.

```
var pageHeader = document.getElementById('page-header');
```

The `pageHeader` element can then be manipulated - its size and color can be changed, and other code can be declared to handle the element being clicked on or hovered over. It's supported in pretty much all the browsers you need to worry about. Notice that `getElementById` is a method of the **document** object. Many of the methods used to access the page are found on the document object.

By Tag Name

`document.getElementsByTagName` works in much the same way as `getElementById`, except that it takes a tag name (`a`, `ul`, `li`, etc) instead of an ID and returns a **NodeList**, which is essentially an array of the DOM Elements.

By Class Name

`document.getElementsByClassName` returns the same kind of `NodeList` as `getElementsByTagName`, except that you pass a class name to be matched, not a tag name.

By CSS Selector

A couple of new methods are available in modern browsers that make selecting elements easier by allowing the use of CSS selectors. They are `document.querySelector` and `document.querySelectorAll`.

```
var pageHeader = document.querySelector('#header');
var buttons = document.querySelectorAll('.btn');
```

`querySelector`, like `getElementById`, returns only one element whereas `querySelectorAll` returns a `NodeList`. If multiple elements match the selector you pass to `querySelector`, only the first will be returned.

HTML DOM Events

What can JavaScript Do?

Event handlers can be used to handle, and verify, user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user input data
- And more ...

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign your own event handler functions to HTML elements
- You can prevent events from being sent or being handled
- And more ...

An HTML event can be something the browser does, or something a user does. Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

```
<some-HTML-element some-event='some JavaScript'>
```

With double quotes:

```
<some-HTML-element some-event="some JavaScript">
```


In the following example, an onclick attribute (with code), is added to a button element:

```
<button onclick='getElementById("demo").innerHTML=Date()>The time is?</button>
```

In the example above, the JavaScript code changes the content of the element with id="demo". In the next example, the code changes the content of it's own element (using **this.innerHTML**):

```
<button onclick="this.innerHTML=Date()">The time is?</button>
```

Hey! JavaScript code is often several lines long. It is more common to see event attributes calling functions.

```
<button onclick="displayDate()">The time is?</button>
```

Assign Events Using the HTML DOM

```
<script>
    document.getElementById("myBtn").onclick=function(){displayDate()};
    function displayDate() {
        document.getElementById("demo").innerHTML = Date();
    }
</script>
```

Common HTML Events

Here is a list of some common HTML events:

Event	Description
Onchange	An HTML element has been changed
OnClick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
Onkeydown	The user pushes a keyboard key
Onload	The browser has finished loading the page

DOM Events in Detail

Input Events

- **onblur** - When a user leaves an input field
- **onchange** - When a user changes the content of an input field
- **onchange** - When a user selects a dropdown value
- **onfocus** - When an input field gets focus
- **onselect** - When input text is selected
- **onsubmit** - When a user clicks the submit button
- **onreset** - When a user clicks the reset button
- **onkeydown** - When a user is pressing/holding down a key

- **onkeypress** - When a user is pressing/holding down a key
- **onkeyup** - When the user releases a key
- **onkeydown** - When the user releases a key
- **onkeydown** vs **onkeyup** - Both

Mouse Events

- **onmouseover/onmouseout** - When the mouse passes over an element
- **onmousedown/onmouseup** - When pressing/releasing a mouse button
- **onmousedown** - When mouse is clicked: Alert which element
- **onmouseup** - When mouse is clicked: Alert which button
- **onmousemove/onmouseout** - When moving the mouse pointer over/out of an image
- **onmouseover/onmouseout** - When moving the mouse over/out of an image
- **onmouseover** an image map

Click Events

- **onclick** - When button is clicked
- **ondblclick** - When a text is double-clicked

Load Events

- **onload** - When the page has been loaded
- **onload** - When an image has been loaded
- **onerror** - When an error occurs when loading an image
- **onunload** - When the browser closes the document

- **onresize** - When the browser window is resized

```
function myFunction() {
    var w = window.outerWidth;
    var h = window.outerHeight;
    var txt = "Window size: width=" + w + ", height=" + h;
    document.getElementById("demo").innerHTML = txt;
}
```

Others

- What is the keycode of the key pressed?

```
<head>
<script>
    function whichButton(event) {
        document.getElementById("demo").innerHTML = event.keyCode;
    }
</script>
</head>
<body onkeyup="whichButton(event)">
    <p id="demo"></p>
</body>
```

- What are the coordinates of the cursor?

```
<head>
<script>
    function show_coords(event) {
        document.getElementById("demo").innerHTML =
```

```
        "X= " + event.clientX + "<br>Y= " + event.clientY;
    }
</script>
</head>
<body>
    <p onmousedown="show_coords(event)">
        Click this paragraph to display the x and y coordinates of the
        mouse pointer.</p>
    <p id="demo"></p>
</body>
```

- What are the coordinates of the cursor, relative to the screen?

```
<head>
<script>
    function coordinates(event) {
        document.getElementById("demo").innerHTML =
            "X = " + event.screenX + "<br>Y = " + event.screenY;
    }
</script>
</head>
<body>
    <p onmousedown="coordinates(event)">
        Click this paragraph, to display the x and y coordinates of
        the cursor, relative to the screen.
    </p>
    <p id="demo"></p>
</body>
```

- Was the shift key pressed?

```
<head>
<script>
    function isKeyPressed(event) {
        var text = "The shift key was NOT pressed!";
        if (event.shiftKey == 1) {
            text = "The shift key was pressed!";
        }
        document.getElementById("demo").innerHTML = text;
    }
</script>
</head>
<body onmousedown="isKeyPressed(event)">
    <p>Click on this paragraph. An alert box will tell you if you
    pressed the shift key or not.</p><p id="demo">
</p>
</body>
```

- Which event type occurred?
`event.type`

Changing HTML Style

The HTML DOM allows JavaScript to change the style of HTML elements. To change the style of an HTML element, use this syntax:

```
document.getElementById(id).style.property = new style
```

The following example changes the style of a <p> element:

```
<body>
<p id="p2">Hello World!</p>
<script>
    document.getElementById("p2").style.color = "blue";
</script>
<p>The paragraph above was changed by a script.</p>
</body>
```

Changing style upon button click:

```
<!DOCTYPE html>
<html>
<body>
    <h1 id="id1">My Heading 1</h1>
    <button type="button"
        onclick="document.getElementById('id1').style.color = 'red'">Click Me!</button>
</body>
</html>
```

JS HTML DOM Document

In the HTML DOM object model, the document object represents your web page and is the owner of all other objects in your web page. If you want to access objects in an HTML page, you always start with accessing the document object. Below are some examples of how you can use the document object to access and manipulate HTML.

Finding HTML Elements

Method	Description
document.getElementById("element_id")	Find an element by element id
document.getElementsByTagName("tag_name")	Find elements by tag name
document.getElementsByClassName("class_name")	Find elements by class name

Changing HTML Elements

Method	Description
element.innerHTML="bla bla"	Change the inner HTML of an element
element.attribute=	Change the attribute of an HTML element
element.setAttribute(attribute,value)	Change the attribute of an HTML element

`element.style.property=`

Change the style of an HTML element

Adding and Deleting Elements

Method	Description
<pre>document.createElement("tr/input etc") <div id="div1"> <p id="p1">This is a paragraph.</p> <p id="p2">This is another paragraph.</p> </div> <script> var para = document.createElement("p"); var node = document.createTextNode("This is new."); para.appendChild(node); var element = document.getElementById("div1"); element.appendChild(para); </script></pre>	Create an HTML element
<pre>document.removeChild(node) <div id="div1"> <p id="p1">This is a paragraph.</p> <p id="p2">This is another paragraph.</p> </div> <script> var para = document.createElement("p"); var node = document.createTextNode("This is new."); para.appendChild(node); var element = document.getElementById("div1"); var child = document.getElementById("p1"); element.insertBefore(para,child); </script></pre>	Remove an HTML element
<pre>document.appendChild(node)</pre>	Add an HTML element
<pre>document.replaceChild(newNode, curr_Node) <div id="div1"> <p id="p1">This is a paragraph.</p> <p id="p2">This is another paragraph.</p> </div> <script> var para = document.createElement("p"); var node = document.createTextNode("This is new."); para.appendChild(node); var parent = document.getElementById("div1"); var child = document.getElementById("p1"); parent.replaceChild(para,child); </script></pre>	Replace an HTML element
<pre>document.write(text)</pre>	Write into the HTML output stream

Adding Events Handlers

Method	Description
<code>document.getElementById(id).onclick=function(){code}</code>	Adding event handler code to an onclick event

Finding HTML Objects

The first HTML DOM Level 1 (1998) defined 11 HTML objects, object collections, and properties. These are still valid in HTML5.

Later, in HTML DOM Level 3, more objects, collections, and properties were added.

Method	Description
<code>document.anchors</code>	Returns all <a> with a value in the name attribute
<code>document.applets</code>	Returns all <applet> elements (Deprecated in HTML5)
<code>document.baseURI</code>	Returns the absolute base URI of the document
<code>document.body</code>	Returns the <body> element
<code>document.cookie</code>	Returns the document's cookie
<code>document.doctype</code>	Returns the document's doctype
<code>document.documentElement</code>	Returns the <html> element
<code>document.documentMode</code>	Returns the mode used by the browser
<code>document.documentURI</code>	Returns the URI of the document
<code>document.domain</code>	Returns the domain name of the document server
<code>document.domConfig</code>	Returns the DOM configuration
<code>document.embeds</code>	Returns all <embed> elements
<code>document.forms</code>	Returns all <form> elements
<code>document.head</code>	Returns the <head> element
<code>document.images</code>	Returns all <image> elements
<code>document.implementation</code>	Returns the DOM implementation
<code>document.inputEncoding</code>	Returns the document's encoding (character set)
<code>document.lastModified</code>	Returns the date and time the document was updated
<code>document.links</code>	Returns all <area> and <a> elements value in href
<code>document.readyState</code>	Returns the (loading) status of the document

document.referrer	Returns the URI of the referrer (the linking document)
document.scripts	Returns all <script> elements
document.strictErrorChecking	Returns if error checking is enforced
document.title	Returns the <title> element
document.URL	Returns the complete URL of the document

JavaScript Cookies

Cookies let you store user information in web pages.

What are Cookies?

Cookies are data, stored in small text files, on your computer. When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user. Cookies were invented to solve the problem "how to remember information about the user": When a user visits a web page, his name can be stored in a cookie. Next time the user visits the page, the cookie "remembers" his name.

Cookies are saved in **name-value** pairs like: username=John Doe

When a browser request a web page from a server, cookies belonging to the page is added to the request. This way the server gets the necessary data to "remember" information about users.

Create a Cookie with JavaScript

JavaScript can create, read, and delete cookies with the **document.cookie** property.

With JavaScript, a cookie can be created like this:

```
document.cookie="username=John Doe";
```

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

```
document.cookie="username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
```

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

```
document.cookie="username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path="/";
```

Read a Cookie with JavaScript

With JavaScript, cookies can be read like this:

```
var x = document.cookie;
```

Hey! document.cookie will return all cookies in one string much like: cookie1=value; cookie2=value; cookie3=value;

Change a Cookie with JavaScript

With JavaScript, you can change a cookie the same way as you create it:

```
document.cookie="username=John Smith; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";  
The old cookie is overwritten.
```

Delete a Cookie with JavaScript

Deleting a cookie is very simple. Just set the expires parameter to a passed date:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Note that you don't have to specify a cookie value when you delete a cookie.

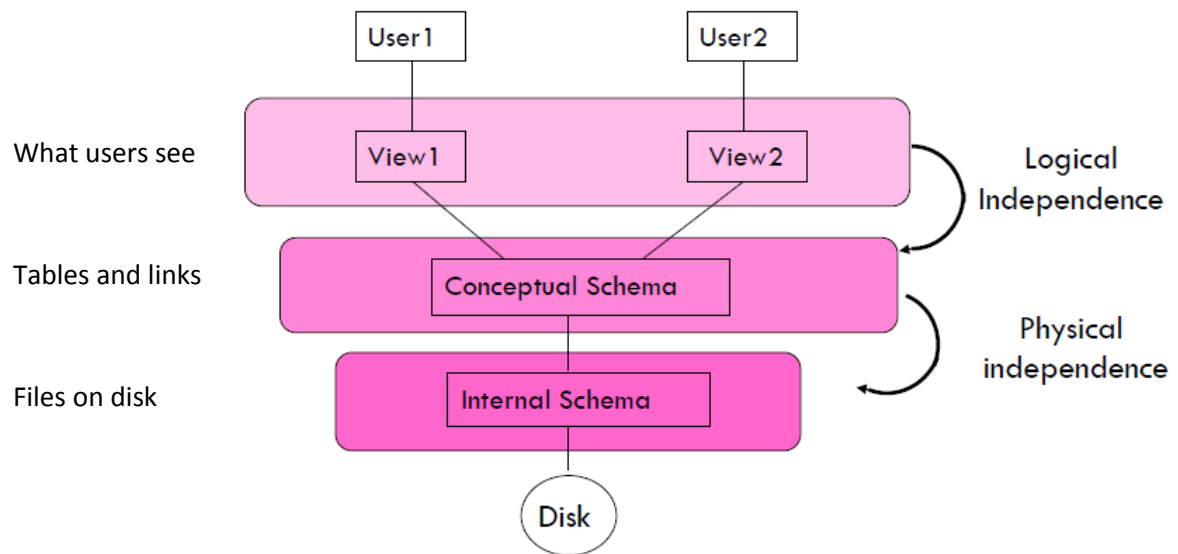
Example Altogether

```
function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime() + (exdays*24*60*60*1000));  
    var expires = "expires="+d.toUTCString();  
    document.cookie = cname + "=" + cvalue + ";" + expires;  
}  
  
function getCookie(cname) {  
    var name = cname + "=";  
    var ca = document.cookie.split(';');  
    for(var i=0; i<ca.length; i++) {  
        var c = ca[i];  
        while (c.charAt(0)==' ') c = c.substring(1);  
        if (c.indexOf(name) != -1) return c.substring(name.length, c.length);  
    }  
    return "";  
}  
  
function checkCookie() {  
    var user = getCookie("username");  
    if (user != "") {  
        alert("Welcome again " + user);  
    } else {  
        user = prompt("Please enter your name:", "");  
        if (user != "" && user != null) {  
            setCookie("username", user, 365);  
        }  
    }  
}
```


Unit 2: Issues of Web Technology

Web Application Architecture

Database Architecture and Data Independence



Each level is independent of the levels below it

Logical Independence: The ability to change the logical schema without changing the external schema or application programs

- Can add new fields, new tables without changing views
- Can change structure of tables without changing view

Physical Independence: The ability to change the physical schema without changing the logical schema

- Storage space can change
- Type of some data can change for reasons of optimization

Moral: Keep the VIEW (what the user sees) independent of the MODEL (domain knowledge)

N-tier Architectures

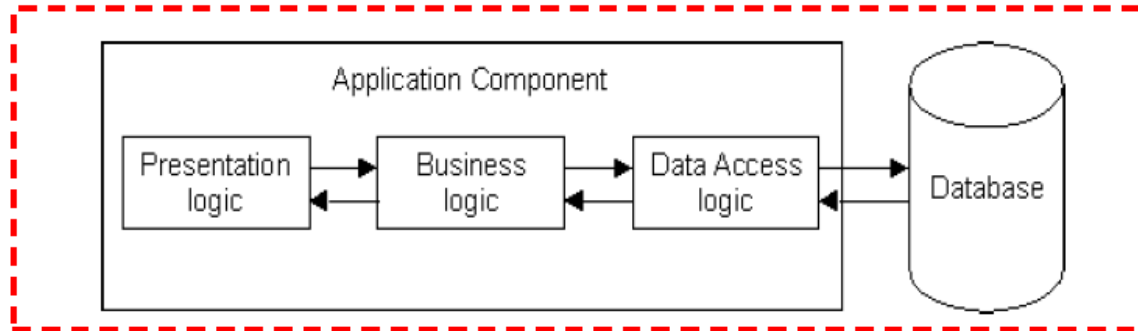
Significance of "Tiers"

- N-tier architectures have the same components
 - Presentation
 - Business/Logic
 - Data

N-tier architectures try to separate the components into different tiers/layers

- Tier: physical separation
- Layer: logical separation

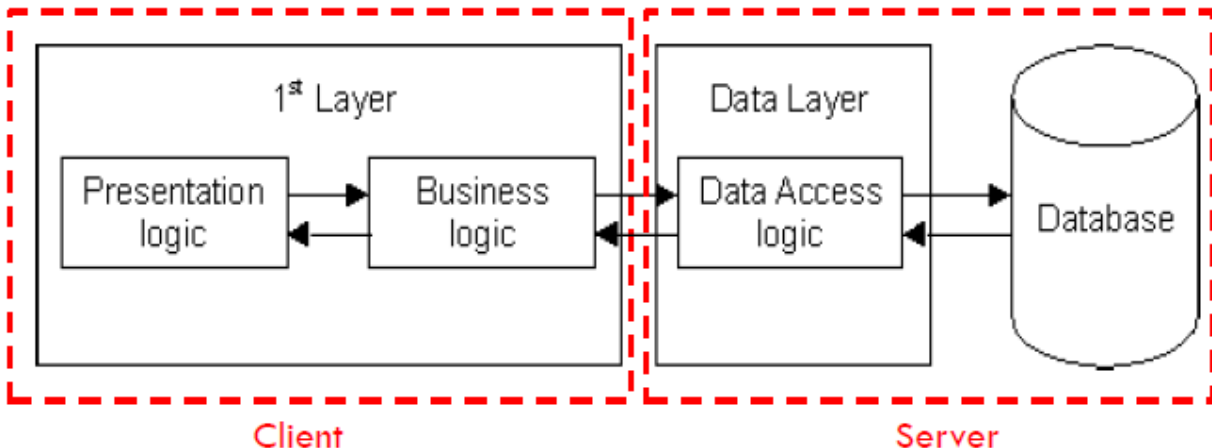
1-Tier Architecture



All 3 layers are on the same machine:

- All code and processing kept on a single machine
- Presentation, Logic, Data layers are tightly connected
- Scalability: Single processor means hard to increase volume of processing
- Portability: Moving to a new machine may mean rewriting everything
- Maintenance: Changing one layer requires changing other layers

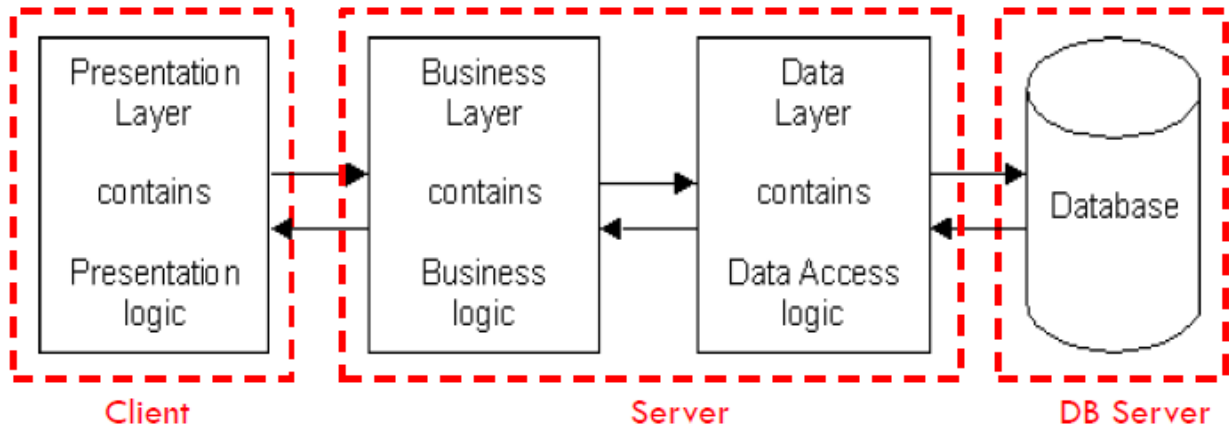
2-Tier Architecture



Database runs on Server

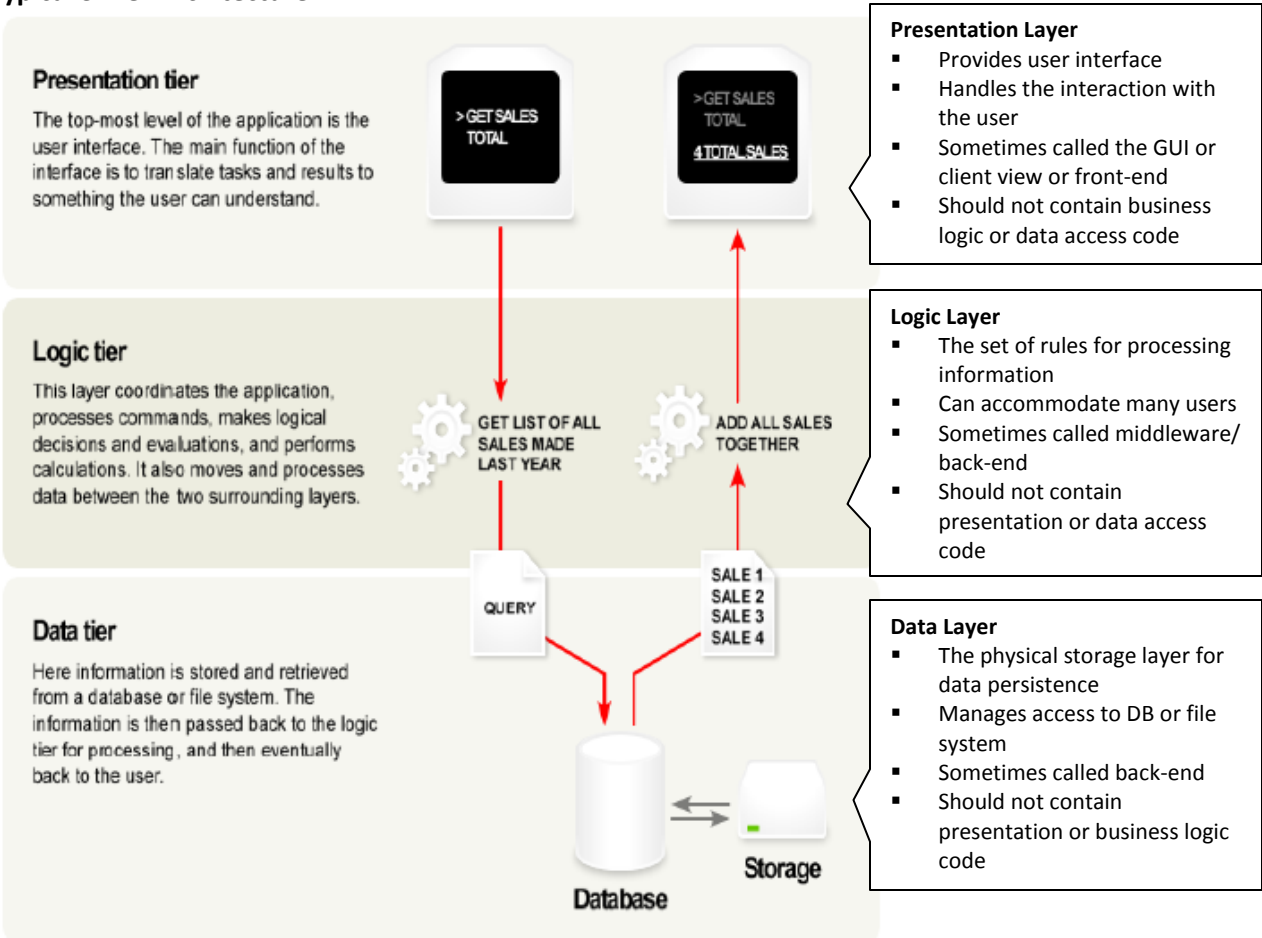
- Separated from client
- Easy to switch to a different database
- Presentation and logic layers still tightly connected
- Heavy load on server
- Potential congestion on network
- Presentation still tied to business logic

3-Tier Architecture



- Each layer can potentially run on a different machine
- Presentation, logic, data layers disconnected

Typical 3-Tier Architecture



3-Tier Architecture Principles

- Client-server architecture
- Each tier (Presentation, Logic, Data) should be independent and should not expose dependencies related to the implementation

- Unconnected tiers should not communicate
- Change in platform affects only the layer running on that particular platform

The 3-Tier Architecture for Web Apps

- Presentation Layer
 - Static or dynamically generated content rendered by the browser (front-end)
- Logic Layer
 - A dynamic content processing and generation level application server, e.g., Java EE, ASP.NET, PHP, ColdFusion platform (middleware)
- Data Layer
 - A database, comprising both data sets and the database management system or RDBMS software that manages and provides access to the data (back-end)

3-Tier Architecture - Advantages

- Independence of Layers
- Easier to maintain
- Components are reusable
- Faster development (division of work)
- Web designer does presentation
- Software engineer does logic
- DB admin does data model

Design Patterns

Design Problems & Decisions

- Construction and testing
 - How do we build a web application?
 - What technology should we choose?
- Re-use
 - Can we use standard components?
- Scalability
 - How will our web application cope with large numbers of requests?
- Security
 - How do we protect against attack, viruses, malicious data access, denial of service?
- Different data views
 - user types, individual accounts, data protection

Moral: *Need for general and reusable solution: **Design Patterns***

What is Design Pattern?

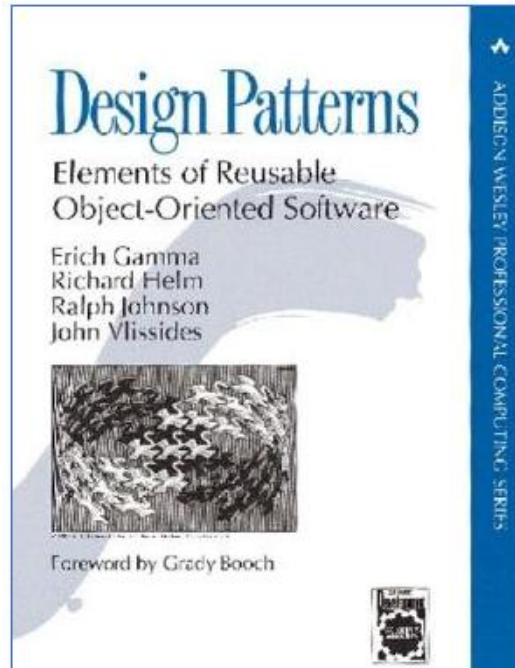
A general and reusable solution to a commonly occurring problem in the design of software

- A template for how to solve a problem that has been used in many different situations
- NOT a finished design
 - the pattern must be adapted to the application
 - cannot simply translate into code

Origin of Design Patterns

- Architectural concept by Christopher Alexander (1977/79)
- Adapted to OO Programming by Beck and Cunningham (1987)

- Popularity in CS after the book: “Design Patterns: Elements of Re-useable Object-oriented software”, 1994. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.



- Now widely-used in software engineering

MVC Design Pattern

Design Problem

- Need to change the **look-and-feel** without changing the **core/logic**.
- Need to **present data** under **different contexts** (e.g., powerful desktop, web, mobile device).
- Need to **interact** with/access data under **different contexts** (e.g., touch screen on a mobile device, keyboard on a computer)
- Need to maintain **multiple views** of the **same data** (list, thumbnails, detailed, etc.)

Design Solution

- Separate core functionality from the presentation and control logic that uses this functionality
- Allow multiple views to share the same data model
- Make supporting multiple clients easier to implement, test, and maintain

The Model-View-Controller Pattern

- Design pattern for graphical systems that promotes separation between model and view
- With this pattern the logic required for data **maintenance** (database, text file) is separated from how the data is **viewed** (graph, numerical) and how the data can be **interacted with** (GUI, command line)

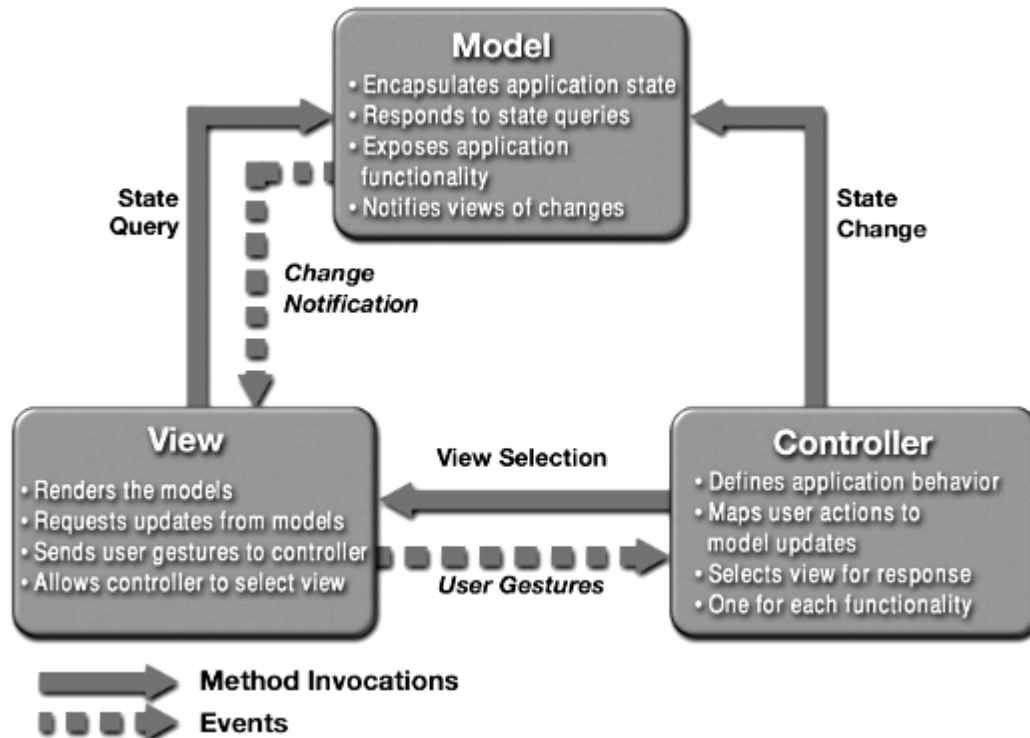


Fig: MVC Architecture

The MVC Pattern

Model

- manages the behavior and data of the application domain
- responds to requests for information about its state (usually from the view)
- follows instructions to change state (usually from the controller)

View

- renders the model into a form suitable for interaction, typically a user interface (multiple views can exist for a single model for different purposes)

Controller

- receives user input and initiates a response by making calls on model objects
- accepts input from the user and instructs the model and viewport to perform actions based on that input

The MVC Pattern (in practice)

Model

- Contains domain-specific knowledge
- Records the state of the application
- E.g., what items are in shopping cart
- Often linked to a database
- Independent of view
- One model can link to different views

View

- Presents data to the user

- Allows user interaction
- Does no processing

Controller

- defines how user interface reacts to user input (events)
- receives messages from view (where events come from)
- sends messages to model (tells what data to display)

The MVC for Web Applications

Model

- database tables (persistent data)
- session information (current system state data)
- rules governing transactions

View

- (X)HTML
- CSS style sheets (CSS2, CSS3)
- server-side templates (e.g. ASP.NET Web Forms)

Controller

- client-side scripting
- http request processing
- business logic/preprocessing

MVC Advantages

- Clarity of Design
 - model methods give an API for data and state
 - eases the design of view and controller
- Efficient Modularity
 - any of the components can be easily replaced
- Multiple Views
 - many views can be developed as appropriate
 - each uses the same API for the model
- Easier to Construct and Maintain
 - simple (text-based) views while constructing
 - more views and controllers can be added
 - stable interfaces ease development
- Distributable
 - natural fit with a distributed environment

3-tier Architecture vs. MVC Architecture

Communication

- **3-tier:** The presentation layer never communicates directly with the data layer-only through the logic layer (linear topology)
- **MVC:** All layers communicate directly (triangle topology)

Usage

- **3-tier:** Mainly used in web applications where the client, middleware and data tiers ran on physically separate platforms

- **MVC:** Historically used on applications that run on a single graphical workstation (applied to separate platforms as *Model 2*) [**Model 2** is a complex design pattern used in the design of Java Web applications which separates the display of content from the logic used to obtain and manipulate the content]

HyperText Transfer Protocol (HTTP)

Definition:

The HTTP is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. **Hypertext** is structured text that uses logical links (hyperlinks) between nodes containing text. **HTTP** is the protocol to exchange or transfer hypertext.

HTTP is a stateless, application-layer protocol for communicating between distributed systems, and is the foundation of the modern web.

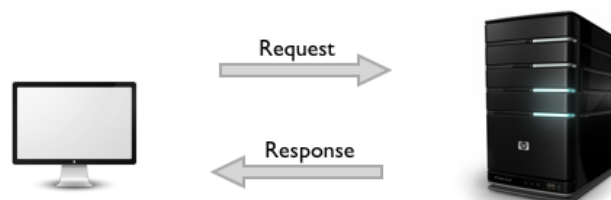
Origin:

Tim Berners-Lee and his team are credited with inventing the original HTTP along with HTML and the associated technology for a web server and a text-based web browser. Berners-Lee first proposed the "WorldWideWeb" project in 1989-now known as the World Wide Web. The first version of the protocol had only one method, namely GET, which would request a page from a server. The response from the server was always an HTML page.

How it works?

HTTP allows for communication between a variety of hosts and clients, and supports a mixture of network configurations. *To make this possible, it assumes very little about a particular system, and does not keep state between different message exchanges.* This makes HTTP a **stateless** protocol. The communication usually takes place over TCP/IP, but any reliable transport can be used. The default port for TCP/IP is **80**, but other ports can also be used.

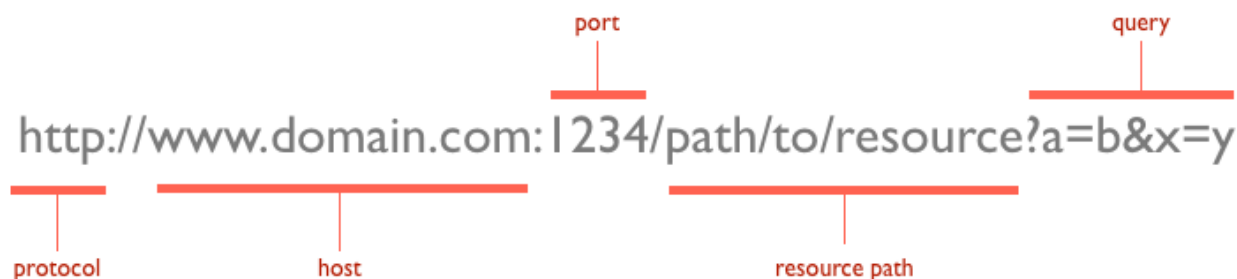
Communication between a host and a client occurs, via a **request/response pair**. The client (Web browser) initiates an HTTP request message, which is serviced through a HTTP response message in return. A web browser is an example of a *user agent* (UA). Other types of user agent include the indexing software used by search providers (web crawlers), voice browsers, mobile apps, and other software that accesses, consumes, or displays web content.



URLs

At the heart of web communications is the request message, which are sent via Uniform Resource Locators (URLs). You guys are already familiar with URLs, but for completeness sake, I'll include it here. URLs have a simple structure that consists of the following components:

For more notes visit <https://collegenote.pythonanywhere.com>



The protocol is typically **http**, but it can also be **https** for secure communications. The default **port** is 80, but one can be set explicitly, as illustrated in the above image. The resource path is the **local path** to the resource on the server.

Verbs

URLs reveal the identity of the particular host with which we want to communicate, but the action that should be performed on the host is specified via HTTP verbs. Of course, there are several actions that a client would like the host to perform. HTTP has formalized on a few that captures the essentials that are universally applicable for all kinds of applications.

These request verbs are:

- **GET**: *fetch* an existing resource. The URL contains all the necessary information the server needs to locate and return the resource.
- **POST**: *create* a new resource. POST requests usually carry a payload that specifies the data for the new resource.
- **PUT**: *update* an existing resource. The payload may contain the updated data for the resource.
- **DELETE**: *delete* an existing resource.

The above four verbs are the most popular, and most tools and frameworks explicitly expose these request verbs. **PUT** and **DELETE** are sometimes considered specialized versions of the **POST** verb, and they may be packaged as POST requests with the payload containing the exact action: *create*, *update* or *delete*.

There are some lesser used verbs that HTTP also supports:

- **HEAD**: this is similar to GET, but without the message body. It's used to retrieve the server headers for a particular resource, generally to check if the resource has changed, via timestamps.
- **TRACE**: used to retrieve the hops that a request takes to round trip from the server. Each intermediate proxy or gateway would inject its IP or DNS name into the via header field. This can be used for diagnostic purposes.
- **OPTIONS**: used to retrieve the server capabilities. On the client-side, it can be used to modify the request based on what the server can support.

Status Codes

With URLs and verbs, the client can initiate requests to the server. In return, the server responds with status codes and message payloads. The status code is important and tells the client how to interpret the server response. The HTTP spec defines certain number ranges for specific types of responses:

1xx: Informational Messages

All HTTP/1.1 clients are required to accept the Transfer-Encoding header.

This class of codes was introduced in HTTP/1.1 and is purely provisional. The server can send a **Expect: 100-continue** message, telling the client to continue sending the remainder of the request, or ignore if it has already sent it. HTTP/1.0 clients are supposed to ignore this header.

2xx: Successful

This tells the client that the request was successfully processed. The most common code is **200 OK**. For a GET request, the server sends the resource in the message body. There are other less frequently used codes:

- **202 Accepted**: the request was accepted but may not include the resource in the response. This is useful for async processing on the server side. The server may choose to send information for monitoring.
- **204 No Content**: there is no message body in the response.
- **205 Reset Content**: indicates to the client to reset its document view.
- **206 Partial Content**: indicates that the response only contains partial content. Additional headers indicate the exact range and content expiration information.

3xx: Redirection

This requires the client to take additional action. The most common use-case is to jump to a different URL in order to fetch the resource.

- **301 Moved Permanently**: the resource is now located at a new URL.
- **303 See Other**: the resource is temporarily located at a new URL. The **Location** response header contains the temporary URL.
- **304 Not Modified**: the server has determined that the resource has not changed and the client should use its cached copy. This relies on the fact that the client is sending **Etag** (Entity Tag) information that is a hash of the content. The server compares this with its own computed **Etag** to check for modifications.

4xx: Client Error

These codes are used when the server thinks that the client is at fault, either by requesting an invalid resource or making a bad request. The most popular code in this class is **404 Not Found**, which I think everyone will identify with. 404 indicates that the resource is invalid and does not exist on the server. The other codes in this class include:

- **400 Bad Request**: the request was malformed.
- **401 Unauthorized**: request requires authentication. The client can repeat the request with the **Authorization** header. If the client already included the **Authorization** header, then the credentials were wrong.
- **403 Forbidden**: server has denied access to the resource.
- **405 Method Not Allowed**: invalid HTTP verb used in the request line, or the server does not support that verb.
- **409 Conflict**: the server could not complete the request because the client is trying to modify a resource that is newer than the client's timestamp. Conflicts arise mostly for PUT requests during collaborative edits on a resource.

5xx: Server Error

This class of codes is used to indicate a server failure while processing the request. The most commonly used error code is **500 Internal Server Error**. The others in this class are:

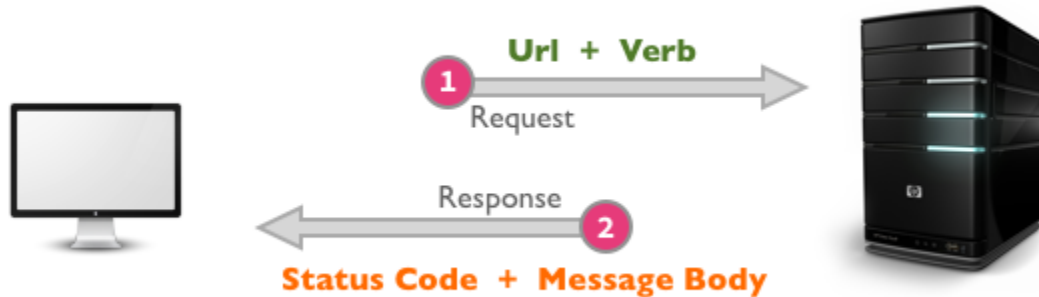
- **501 Not Implemented**: the server does not yet support the requested functionality.

For more notes visit <https://collegenote.pythonanywhere.com>

- **503 Service Unavailable:** this could happen if an internal system on the server has failed or the server is overloaded. Typically, the server won't even respond and the request will timeout.

Request and Response Message Formats

So far, we've seen that *URLs, verbs and status codes* make up the fundamental pieces of an HTTP request/response pair.



Let's now look at the content of these messages. The HTTP specification states that a request or response message has the following generic structure:

```
1 message = <start-line>
2     *(<message-header>)
3     CRLF
4     [<message-body>]
5
6 <start-line> = Request-Line | Status-Line
7 <message-header> = Field-Name ':' Field-Value
```

It's mandatory to place a new line between the message headers and body. The message can contain one or more headers, of which are broadly classified into:

- **General headers:** headers shared by for both request and response messages.

```
1 general-header = Cache-Control
2                 | Connection
3                 | Date
4                 | Pragma
5                 | Trailer
6                 | Transfer-Encoding
7                 | Upgrade
8                 | Via
9                 | Warning
```

- **Request specific headers:** Will be explained in Message-Format section.
- **Response specific headers:** Will be explained in Message-Format section.
- **Entity headers:**
Request and Response messages may also include entity headers to provide meta-information about the content (aka Message Body or Entity). These headers include:

```
01 entity-header = Allow
02                | Content-Encoding
03                | Content-Language
04                | Content-Length
05                | Content-Location
06                | Content-MD5
07                | Content-Range
08                | Content-Type
09                | Expires
10                | Last-Modified
```

All of the **Content-** prefixed headers provide information about the structure, encoding and size of the message body. Some of these headers need to be present if the entity is part of the message.

Request Format

The request message has the same generic structure as above, except for the request line which looks like:

```
1 Request-Line = Method SP URI SP HTTP-Version CRLF
2 Method = "OPTIONS"
3           | "HEAD"
4           | "GET"
5           | "POST"
6           | "PUT"
7           | "DELETE"
8           | "TRACE"
```

SP is the space separator between the tokens. HTTP-Version is specified as *"HTTP/1.1"* and then followed by a new line. Thus, a typical request message might look like:

```
1 GET /articles/http-basics HTTP/1.1
2 Host: www.articles.com
3 Connection: keep-alive
4 Cache-Control: no-cache
5 Pragma: no-cache
6 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

Note the request line followed by many request headers. The **Host** header is mandatory for HTTP/1.1 clients. **GET** requests do not have a message body, but **POST** requests can contain the post data in the body.

The request headers act as modifiers of the request message. The complete list of known request headers is not too long, and is provided below. Unknown headers are treated as entity-header fields.

```
01 request-header = Accept
02                | Accept-Charset
03                | Accept-Encoding
04                | Accept-Language
05                | Authorization
06                | Expect
07                | From
08                | Host
```

09	If-Match
10	If-Modified-Since
11	If-None-Match
12	If-Range
13	If-Unmodified-Since
14	Max-Forwards
15	Proxy-Authorization
16	Range
17	Referer
18	TE
19	User-Agent

The Accept prefixed headers indicate the acceptable media-types, languages and character sets on the client. From, Host, Referrer and User-Agent identify details about the client that initiated the request. The If- prefixed headers are used to make a request more conditional, and the server returns the resource only if the condition matches. Otherwise, it returns a 304 Not Modified. The condition can be based on a timestamp or an **ETag** (a hash of the entity).

Response Format

The response format is similar to the request message, except for the status line and headers. The status line has the following structure:

1 Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

- HTTP-Version is sent as HTTP/1.1
- The Status-Code is one of the many statuses discussed earlier.
- The Reason-Phrase is a human-readable version of the status code.

A typical status line for a successful response might look like so:

1 HTTP/1.1 200 OK

The response headers are also fairly limited, and the full set is given below:

1	response-header = Accept-Ranges
2	Age
3	ETag
4	Location
5	Proxy-Authenticate
6	Retry-After
7	Server
8	Vary
9	WWW-Authenticate

- Age is the time in seconds since the message was generated on the server.
- ETag is the MD5 hash of the entity and used to check for modifications.
- Location is used when sending a redirection and contains the new URL.
- Server identifies the server generating the message.

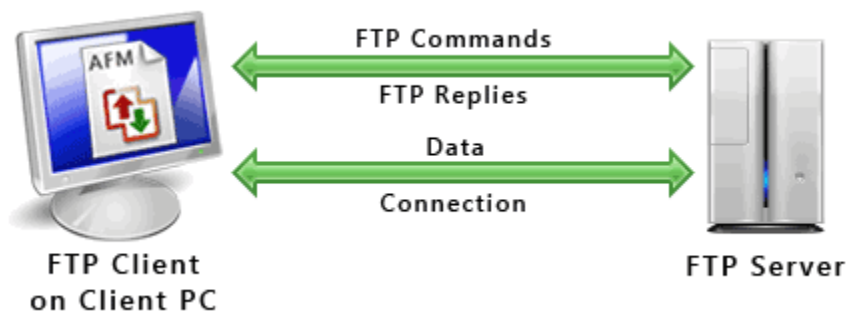
It's been a lot of theory guys; let's play with some tools to view HTTP traffic like chrome/webkit inspector in class.

File Transfer Protocol (FTP)

The **FTP** is a standard network protocol used to transfer computer files from one host to another host over a TCP-based network, such as the Internet. FTP is built on client-server architecture and uses separate control and data connections between the client and the server. FTP users may authenticate themselves using a clear-text sign-in protocol, normally in the form of a username and password, but can connect anonymously if the server is configured to allow it. Once a FTP connection is established, you can use it to send, receive, delete, rename or move files.

How does FTP work?

When you want to copy files between two computers that are on the same local network, often you can simply "share" a drive or folder, and copy the files the same way you would copy files from one place to another on your own PC. What if you want to copy files from one computer to another that is halfway around the world? You would probably use your Internet connection. However, for security reasons, it is very uncommon to share folders over the Internet. File transfers over the Internet use special techniques, of which one of the oldest and most widely-used is FTP. Transferring files from a client computer to a server computer is called "**uploading**" and transferring from a server to a client is "**downloading**".



The FTP client establishes a connection to a remote FTP server in the **active** or **passive** mode. Passive mode is used when the client is behind a firewall and cannot accept TCP connections. Depending on the server settings, the client connects to the server anonymously or with a user name and password. Separate control and data connections are initiated in parallel between the client and the server. Once connected, the client sends and/or receives single files or groups of files. The files are transferred in either **stream mode**, **block mode** or **compressed mode**. The client closes the connection once the server indicates the end of the data transfer.

FTP and Internet Connections

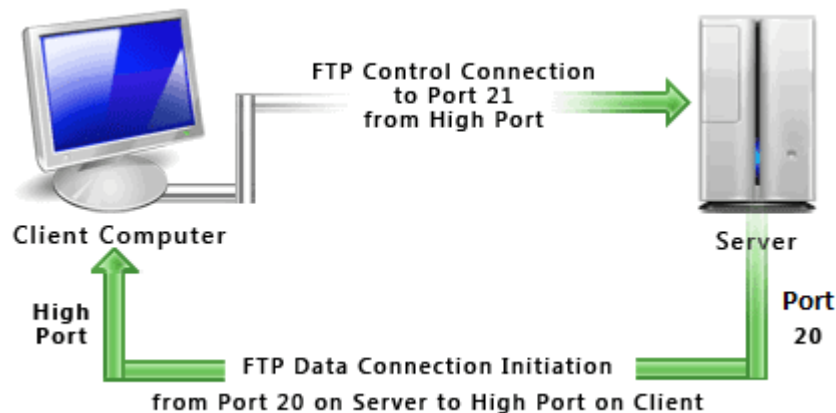
FTP uses one connection for commands and the other for sending and receiving data. FTP has a standard port number on which the FTP server "listens" for connections. A port is a "logical connection point" for communicating using the Internet Protocol (IP). The standard port number used by FTP servers is 21 and is used only for sending commands. Since port 21 is used exclusively for sending commands, this port is referred to as a **command port**. For example, to get a list of folders and files present on the FTP server, the FTP Client issues a "LIST" command. The FTP server then sends a list of all folders and files back to the FTP Client. So what about the internet connection used to send and receive data? The port that is used for transferring data is referred to as a **data port**. The number of the data port will vary depending on the "mode" of the connection.

Active and Passive Connection Mode

The FTP server may support **Active** or **Passive** connections, or both. In an Active FTP connection, the client opens a port and listens and the server actively connects to it. In a Passive FTP connection, the server opens a port and listens (passively) and the client connects to it.

Most FTP client programs select passive connection mode by default because server administrators prefer it as a safety measure. Firewalls generally block connections that are "initiated" from the outside.

If you are connecting to the FTP server using **Active mode** of connection you must set your firewall to accept connections to the port that your FTP client will open. However, many Internet service providers block incoming connections to all ports above 1024. Active FTP servers generally use **port 20 as their data port**.



It's a good idea to use **Passive mode** to connect to an FTP server. Most FTP servers support the Passive mode.

What are the different data representation modes on FTP?

Data transferred over FTP is sent in ASCII, binary, EBCDIC or local modes. Most of the FTP clients automatically determine the data transfer mode based on the contents or extension of the file. Audio, video and image files are generally transferred in binary mode, whereas HTML, script and text files are transferred in the ASCII mode. Computers with identical setups transfer data in the local mode, and hosts using the EBCDIC character set use the EBCDIC mode for data transfer.

What are the uses of FTP?

The most common use of FTP is to upload and download files, such as web page files, to a server. Websites use it in anonymous mode to power downloads. Some companies distribute their software updates using FTP.

What are the disadvantages of FTP?

FTP is not a secure protocol. FTP sends and receives all data in clear text and is, hence, vulnerable to packet capture or sniffing, port stealing, spoof attacks, bounce attacks, brute force attacks and user name hijacking. FTPS, not to be confused with SFTP, is an extension of the standard FTP that allows the FTP clients to request an encrypted session.

What are the popular FTP clients and servers?

For more notes visit <https://collegenote.pythonanywhere.com>

Cerberus FTP and **Complete FTP** is a couple of commercial FTP servers. **FileZilla** is an open-source freeware FTP server that also provides a free FTP client. Some of the other popular FTP clients are WinSCP, Transmit, FireFTP and Cyberduck.

FTP is a widely used method to transfer files over a network. FTP clients today have excellent graphical user interfaces. File transfers are seamless and just a matter of drag and drop.

Unit 3: The Client Tier

Introduction to XML

In short:

- XML stands for **EXtensible Markup Language**
- **Most Hyped Technology** during the late 90s and the current decade
- XML is a **markup language** much like HTML
- XML was designed to **describe data**, not to display data as HTML
- XML tags are not predefined. You **must define your own tags**
- XML is designed to be **self-descriptive**
- XML is a **W3C Recommendation** since February 10, 1998.

The essence of XML is in its name: Extensible Markup Language.

Extensible

XML is extensible. It lets you define your own tags, the order in which they occur, and how they should be processed or displayed. Another way to think about extensibility is to consider that XML allows all of us to extend our notion of what a document is: it can be a file that lives on a file server, or it can be a transient piece of data that flows between two computer systems (as in the case of Web Services).

Markup

The most recognizable feature of XML is its tags, or elements (to be more accurate). In fact, the elements you'll create in XML will be very similar to the elements you've already been creating in your HTML documents. However, XML allows you to define your own set of tags.

Language

XML is a language that's very similar to HTML. It's much more flexible than HTML because it allows you to create your own custom tags. However, it's important to realize that XML is not just a language. XML is a meta-language: a language that allows us to create or define other languages. For example, with XML we can create other languages, such as RSS, MathML (a mathematical markup language), and even tools like XSLT.

Moral: XML is a software- and hardware-independent tool for carrying information especially in WEB.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The first line is the XML declaration. It defines the XML version (1.0). The next line describes the **root element** of the document (like saying: "this document is a note"). The next 4 lines describe 4 **child elements** of the root (to, from, heading, and body) and finally the last line defines the end of the root element.

XML Versions

There are two current versions of XML. The first (**XML 1.0**) was initially defined in 1998. It has undergone minor revisions since then, without being given a new version number, and is currently in its fifth edition, as published on November 26, 2008. It is widely implemented and still recommended for general use. The second (**XML 1.1**) was initially published on February 4, 2004, the same day as XML 1.0 Third Edition, and is currently in its second edition, as published on August 16, 2006. There has been discussion of an XML 2.0, although no organization has announced plans for work on such a project.

The Difference between XML and HTML

XML is not replacement for HTML rather both were designed with different goals:

- XML was designed to describe data, with focus on what data is
- HTML was designed to display data, with focus on how data looks
- HTML is about displaying information, while XML is about carrying information.

XML Does Not DO Anything

Maybe it is a little hard to understand, but XML does not DO anything. The very first example above is a note to Tove, from Jani, stored as XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
</messages>
```

The note above is quite self descriptive. It has sender and receiver information, it also has a heading and a message body. But still, this XML document does not DO anything. It is just information wrapped in tags. Someone must write a piece of software to send, receive or display it.

With XML You Invent Your Own Tags

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document. I.e. the XML language has no predefined tags.

How Can XML be used? [Benefits]

XML is used in many aspects of web development, often to simplify data storage and sharing.

1. XML Separates Data from HTML
2. XML Simplifies Data Sharing
3. XML Simplifies Data Transport

4. XML Simplifies Platform Changes
5. XML Makes Your Data More Available on diverse applications
6. Internet Languages Written in XML: Several Internet languages are written in XML. Here are some examples:
 - XHTML
 - XML Schema
 - SVG
 - WSDL
 - RSS

XML Tree [XML Documents Form a Tree Structure]

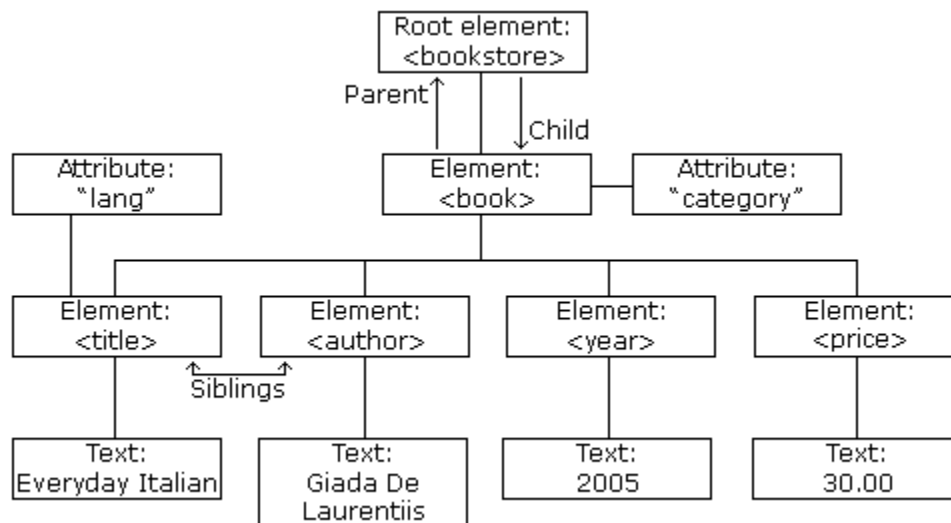
XML documents form a tree structure that starts at "the root" and branches to "the leaves". XML documents must contain a root element. This element is "the parent" of all other elements. The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree.

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. Parent elements have children. Children on the same level are called siblings (brothers or sisters). All elements can have text content and attributes (just like in HTML).

Example:



The image above represents one book in the XML below:

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
```

For more notes visit <https://collegenote.pythonanywhere.com>

```
</book>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

The root element in the example is <bookstore>. All <book> elements in the document are contained within <bookstore>. The <book> element has 4 children: <title>,< author>, <year>, <price>.

XML Syntax

The syntax rules of XML are very simple and logical. The rules are easy to learn, and easy to use.

All XML Elements Must Have a Closing Tag: In HTML, some elements do not have to have a closing tag:

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

In XML, it is illegal to omit the closing tag. All elements **must** have a closing tag:

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

XML Tags are Case Sensitive: XML tags are case sensitive. The tag <Letter> is different from the tag <letter>.

XML Elements Must be Properly Nested: In HTML, you might see improperly nested elements:

```
<b><i>This text is bold and italic</b></i>
```

In XML, all elements **must** be properly nested within each other:

```
<b><i>This text is bold and italic</i></b>
```

XML Documents Must Have a Root Element: XML documents must contain one element that is the **parent** of all other elements. This element is called the **root** element.

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML Attribute Values Must be Quoted: XML elements can have attributes in name/value pairs just like in HTML.

In XML, the attribute values must always be quoted.

Study the two XML documents below. The first one is incorrect, the second is correct:

```
<note date=12/11/2007>
  <to>Tove</to>
  <from>Jani</from>
</note>
```

```
<note date="12/11/2007">
  <to>Tove</to>
  <from>Jani</from>
</note>
```

The error in the first document is that the date attribute in the note element is not quoted.

Entity References

Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element. This will generate an XML error:

```
<message>if salary < 1000 then</message>
```

To avoid this error, replace the "<" character with an **entity reference**:

```
<message>if salary &lt; 1000 then</message>
```

There are 5 predefined entity references in XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Note: Only the characters "<" and "&" are strictly illegal in XML. The greater than character is legal, but it is a good habit to replace it.

Comments in XML

The syntax for writing comments in XML is similar to that of HTML.

```
<!-- This is a comment -->
```

White-space is Preserved in XML

HTML truncates multiple white-space characters to one single white-space:

HTML:	Hello Tove
Output:	Hello Tove

With XML, the white-space in a document is not truncated.

XML Stores New Line as LF

- Windows applications store a new line as: carriage return and line feed (**CR+LF**).
- Unix and Mac OSX uses **LF**.
- Old Mac systems uses **CR**.
- XML stores a new line as **LF**.

Well Formed XML

XML documents that conform to the syntax rules above are said to be "Well Formed" XML documents.

XML Elements

An XML document contains XML Elements.

What is an XML Element?

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

An element can contain:

- other elements
- text
- attributes
- or a mix of all of the above...

```
<bookstore>
<book category="CHILDREN">
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
<book category="WEB">
  <title>Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

In the example above, <bookstore> and <book> have **element contents**, because they contain other elements. <book> also has an **attribute** (category="CHILDREN"). <title>, <author>, <year>, and <price> have **text content** because they contain text.

Empty XML Elements

An alternative syntax can be used for XML elements with no content: Instead of writing a book element (with no content) like this:

For more notes visit <https://collegenote.pythonanywhere.com>

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

It can be written like this:

```
<book />
```

This sort of element syntax is called self-closing.

XML Attributes

XML elements can have attributes, just like HTML. Attributes provide additional information about an element. In HTML, attributes provide additional information about elements:

```

```

```
<a href="demo.asp">
```

Attributes often provide information that is not a part of the data. In the example below, the file type is irrelevant to the data, but can be important to the software that wants to manipulate the element:

```
<file type="gif">computer.gif</file>
```

XML Elements vs. Attributes

Take a look at these examples:

```
<person sex="female">
```

```
<firstname>Anna</firstname>
```

```
<lastname>Smith</lastname>
```

```
</person>
```

```
<person>
```

```
<sex>female</sex>
```

```
<firstname>Anna</firstname>
```

```
<lastname>Smith</lastname>
```

```
</person>
```

In the first example sex is an attribute. In the last, sex is an element. Both examples provide the same information. There are **no rules** about when to use attributes or when to use elements. Attributes are handy in HTML. In XML, my advice (☺) is to avoid them. Use elements instead this makes parsing xml uniform.

My proper way

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
<note date="10/01/2008">
```

```
<to>Tove</to>
```

```
<from>Jani</from>
```

```
<heading>Reminder</heading>
```

```
<body>Don't forget me this weekend!</body>
```

```
</note>
```

A date element is used in the second example:

```
<note>
```

```
<date>10/01/2008</date>
```

```
<to>Tove</to>
```

```
<from>Jani</from>
```

```
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

An expanded date element is used in the third: (THIS IS MY FAVORITE):

```
<note>
  <date>
    <day>10</day>
    <month>01</month>
    <year>2008</year>
  </date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Avoid XML Attributes?

Some of the problems with using attributes are:

- attributes cannot contain multiple values (elements can)
- attributes cannot contain tree structures (elements can)
- attributes are not easily expandable (for future changes)

Attributes are difficult to read and maintain. Use elements for data. **Use attributes for information that is not relevant to the data.**

Don't end up like this:

```
<note day="10" month="01" year="2008" to="Tove" from="Jani" heading="Reminder" body="Don't
forget me this weekend!">
</note>
```

XML Attributes for Metadata

Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML. This example demonstrates this:

```
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```


The id attributes above are for identifying the different notes. It is not a part of the note itself. What I'm trying to say here is that metadata (data about data) should be stored as attributes, and the data itself should be stored as elements.

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>
```

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="http://www.w3schools.com/furniture">
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

XML Encoding

XML documents can contain international characters, like Norwegian or French etc. To avoid errors, you should specify the encoding used, or save your XML files as UTF-8.

Character Encoding

Character encoding defines a unique binary code for each different character used in a document. In computer terms, character encoding are also called character set, character map, code set, and code page.

Unicode

Unicode is an industry standard for character encoding of text documents. It defines (nearly) every possible international character by a name and a number.

Unicode has two variants: UTF-8 and UTF-16.

UTF = **U**niversal character set **T**ransformation **F**ormat.

UTF-8 uses 1 byte (8-bits) to represent characters in the ASCII set, and two or three bytes for the rest.

UTF-16 uses 2 bytes (16 bits) for most characters, and four bytes for the rest.

UTF-8 - The Web Standard

UTF-8 is the standard character encoding on the web.

UTF-8 is the default character encoding for HTML5, CSS, JavaScript, PHP, SQL, and XML.

XML Encoding

The first line in an XML document is called the **prolog**:

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

The prolog is optional. Normally it contains the XML version number. It can also contain information about the encoding used in the document. This prolog specifies UTF-8 encoding:

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

The XML standard states that all XML software must understand both UTF-8 and UTF-16. UTF-8 is the default for documents without encoding information. In addition, most XML software systems understand encodings like ISO-8859-1, Windows-1252, and ASCII.

XML Errors

Most often, XML documents are created on one computer, uploaded to a server on a second computer, and displayed by a browser on a third computer. If the encoding is not correctly interpreted by all the three computers, the browser might display meaningless text, or you might get an error message.

DTD (Document Type Definition)

The purpose of a DTD (Document Type Definition) is to define the legal building blocks of an XML document. A DTD defines the document structure with a list of legal elements and attributes.

Why Use a DTD?

- With a DTD, each of your XML files can carry a description of its own format.
- With a DTD, independent groups of people can agree to use a standard DTD for interchanging data.
- Your application can use a standard DTD to verify that the data you receive from the outside world is valid.
- You can also use a DTD to verify your own data.

A DTD can be declared inline inside an XML document, or as an external reference.

Internal DTD Declaration

If the DTD is declared inside the XML file, it should be wrapped in a DOCTYPE definition with the following syntax:

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

Example 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>
```

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 contains 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"

External DTD Declaration

If the DTD is declared in an external file, it should be wrapped in a DOCTYPE definition with the following syntax:

```
<!DOCTYPE root-element SYSTEM "filename">
```

This is the same XML document as above, but with 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 this 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)>
```

The Building Blocks of XML Documents

Seen from a DTD point of view, all XML documents (and HTML documents) are made up by the following building blocks:

- Elements
- Attributes
- Entities
- PCDATA
- CDATA

PCDATA

PCDATA means parsed character data. Think of character data as the text found between the start tag and the end tag of an XML element. **PCDATA is text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup.** Tags inside the text will be treated as markup and entities will be expanded. However, parsed character data should not contain any &, <, or > characters; these need to be represented by the & < and > entities, respectively.

CDATA

CDATA means character data. **CDATA is text that will NOT be parsed by a parser.** Tags inside the text will NOT be treated as markup and entities will not be expanded.

DTD Elements

In a DTD, elements are declared with an ELEMENT declaration.

Declaring Elements

In a DTD, XML elements are declared with an element declaration with the following syntax:

```
<!ELEMENT element-name category>
```

or

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

Empty Elements

Empty elements are declared with the category keyword EMPTY:

```
<!ELEMENT element-name EMPTY>
```

Example:

```
<!ELEMENT br EMPTY>
```

XML example:

```
<br />
```

Elements with Parsed Character Data

Elements with only parsed character data are declared with #PCDATA inside parentheses:

```
<!ELEMENT element-name (#PCDATA)>
```

Example:

```
<!ELEMENT from (#PCDATA)>
```

Elements with any Contents

Elements declared with the category keyword ANY, can contain any combination of parsable data:

```
<!ELEMENT element-name ANY>
```

Example:

```
<!ELEMENT note ANY>
```

Elements with Children (sequences)

Elements with one or more children are declared with the name of the children elements inside parentheses:

```
<!ELEMENT element-name (child1)>
```

or

```
<!ELEMENT element-name (child1,child2,...)>
```

Example:

```
<!ELEMENT note (to,from,heading,body)>
```

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the "note" element is:

```
<!ELEMENT note (to,from,heading,body)>
```

```
<!ELEMENT to (#PCDATA)>
```

```
<!ELEMENT from (#PCDATA)>
```

For more notes visit <https://collegenote.pythonanywhere.com>

```
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Declaring Only One Occurrence of an Element

```
<!ELEMENT element-name (child-name)>
```

Example:

```
<!ELEMENT note (message)>
```

The example above declares that the child element "message" must occur once, and only once inside the "note" element.

Declaring Minimum One Occurrence of an Element

```
<!ELEMENT element-name (child-name+)>
```

Example:

```
<!ELEMENT note (message+)>
```

The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.

Declaring Zero or More Occurrences of an Element

```
<!ELEMENT element-name (child-name*)>
```

Example:

```
<!ELEMENT note (message*)>
```

The * sign in the example above declares that the child element "message" can occur zero or more times inside the "note" element.

Declaring Zero or One Occurrences of an Element

```
<!ELEMENT element-name (child-name?)>
```

Example:

```
<!ELEMENT note (message?)>
```

The ? sign in the example above declares that the child element "message" can occur zero or one time inside the "note" element.

Declaring either/or Content

Example:

```
<!ELEMENT note (to,from,header,(message|body))>
```

The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

Declaring Mixed Content

Example:

```
<!ELEMENT note (#PCDATA|to|from|header|message)*>
```

The example above declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

DTD Attributes

In the DTD, XML element attributes are declared with an ATTLIST declaration. An attribute declaration has the following syntax:

```
<!ATTLIST element-name attribute-name attribute-type default-value>
```

As you can see from the syntax above, the ATTLIST declaration defines the element which can have the attribute, the name of the attribute, the type of the attribute, and the default attribute value. The **attribute-type** can have the following values:

Value	Explanation
CDATA	The value is character data
(eval eval ..)	The value must be an enumerated value
ID	The value is an unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is predefined

The **attribute-default-value** can have the following values:

Value	Explanation
#DEFAULT value	The attribute has a default value
#REQUIRED	The attribute value must be included in the element
#IMPLIED	The attribute does not have to be included
#FIXED value	The attribute value is fixed

Attribute declaration example

DTD example:

```
<!ELEMENT square EMPTY>  
<!ATTLIST square width CDATA "0">
```

XML example:

```
<square width="100"></square>
```

In the above example the element square is defined to be an empty element with the attributes width of type CDATA. The width attribute has a default value of 0.

Default attributes value

Syntax:

```
<!ATTLIST element-name attribute-name CDATA "default-value">
```

DTD example:

```
<!ATTLIST payment type CDATA "check">
```

XML example:

```
<payment type="check">
```

Specifying a default value for an attribute, assures that the attribute will get a value even if the author of the XML document didn't include it.

Implied attribute

Syntax:

```
<!ATTLIST element-name attribute-name attribute-type #IMPLIED>
```

DTD example:

```
<!ATTLIST contact fax CDATA #IMPLIED>
```

XML example:

```
<contact fax="555-667788">
```

Use an implied attribute if you don't want to force the author to include an attribute and you don't have an option for a default value either.

Required attribute

Syntax:

```
<!ATTLIST element-name attribute_name attribute-type #REQUIRED>
```

DTD example:

```
<!ATTLIST person number CDATA #REQUIRED>
```

XML example:

```
<person number="5677">
```

Use a required attribute if you don't have an option for a default value, but still want to force the attribute to be present.

Fixed attribute value

Syntax:

```
<!ATTLIST element-name attribute-name attribute-type #FIXED "value">
```

DTD example:

```
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```


XML example:

```
<sender company="Microsoft">
```

Use a fixed attribute value when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error.

Enumerated attribute values

Syntax:

```
<!ATTLIST element-name attribute-name (eval|eval|..) default-value>
```

DTD example:

```
<!ATTLIST payment type (check|cash) "cash">
```

XML example:

```
<payment type="check">
```

or

```
<payment type="cash">
```

Use enumerated attribute values when you want the attribute values to be one of a fixed set of legal values.

DTD Examples

1. Newspaper article DTD

```
<!DOCTYPE NEWSPAPER [  
  
<!ELEMENT NEWSPAPER (ARTICLE+)>  
<!ELEMENT ARTICLE (HEADLINE, BYLINE, LEAD, BODY, NOTES)>  
<!ELEMENT HEADLINE (#PCDATA)>  
<!ELEMENT BYLINE (#PCDATA)>  
<!ELEMENT LEAD (#PCDATA)>  
<!ELEMENT BODY (#PCDATA)>  
<!ELEMENT NOTES (#PCDATA)>  
  
<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>  
<!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>  
<!ATTLIST ARTICLE DATE CDATA #IMPLIED>  
<!ATTLIST ARTICLE EDITION CDATA #IMPLIED>  
  
<!ENTITY NEWSPAPER "Vervet Logic Times">  
<!ENTITY PUBLISHER "Vervet Logic Press">  
<!ENTITY COPYRIGHT "Copyright 1998 Vervet Logic Press">  
  
>]
```

2. TV schedule DTD

```
<!DOCTYPE TVSCHEDULE [  

```

For more notes visit <https://collegenote.pythonanywhere.com>

```
<!ELEMENT TVSCHEDULE (CHANNEL+)>
<!ELEMENT CHANNEL (BANNER,DAY+)>
<!ELEMENT BANNER (#PCDATA)>
<!ELEMENT DAY (DATE,(HOLIDAY|PROGRAMSLOT+)+)>
<!ELEMENT HOLIDAY (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT PROGRAMSLOT (TIME,TITLE,DESCRIPTION?)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>

<!ATTLIST TVSCHEDULE NAME CDATA #REQUIRED>
<!ATTLIST CHANNEL CHAN CDATA #REQUIRED>
<!ATTLIST PROGRAMSLOT VTR CDATA #IMPLIED>
<!ATTLIST TITLE RATING CDATA #IMPLIED>
<!ATTLIST TITLE LANGUAGE CDATA #IMPLIED>
]>
```

XML Schema (XSD)

XML Schema is an XML-based alternative to DTD which describes the structure of an XML document. The XML Schema language is also referred to as XML Schema Definition (**XSD**). XML Schema became a W3C's recommendation in 02. May 2001.

What is an XML Schema?

The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

An XML Schema:

- defines elements that can appear in a document
- defines attributes that can appear in a document
- defines which elements are child elements
- defines the order of child elements
- defines the number of child elements
- defines whether an element is empty or can include text
- defines data types for elements and attributes
- defines default and fixed values for elements and attributes

XML Schemas are the Successors of DTDs

XML Schemas are (and will be) used in most Web applications as a replacement for DTDs. Reasons for replacement include: extensible to future additions, richer and more powerful than DTDs, support data types and support namespaces.

Example before discussion:

A Simple XML Document

Look at this simple XML document called "note.xml":

```
<?xml version="1.0"?>
<note>
```

For more notes visit <https://collegenote.pythonanywhere.com>

```
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

We have already written a DTD for this xml. Its turn now for XSD:

The following example is an XML Schema file called "note.xsd" that defines the elements of the XML document above ("note.xml"):

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.a.com"
xmlns=http://www.a.com" elementFormDefault="qualified">
```

```
<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>
```

Let's elaborate XSD document above (note.xsd):

The <schema> Element

The <schema> element is the root element of every XML Schema:

```
<?xml version="1.0"?>
<xs:schema>
  ...
  ...
</xs:schema>
```

The <schema> element may contain some attributes. A schema declaration often looks something like this:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.a.com"
xmlns="http://www.a.com"
elementFormDefault="qualified">
  ...
  ...
</xs:schema>
```

The following fragment:

xmlns:xs="http://www.w3.org/2001/XMLSchema" indicates that the elements and data types used in the schema come from the "http://www.w3.org/2001/XMLSchema" namespace. It also specifies that

For more notes visit <https://collegenote.pythonanywhere.com>

the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be prefixed with **xs**:

This fragment:

targetNamespace="http://www.a.com" indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.a.com" namespace.

This fragment:

xmlns="http://www.w3schools.com" indicates that the default namespace is "http://www.a.com".

This fragment:

elementFormDefault="qualified" indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

Referencing a Schema in an XML Document

This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note xmlns="http://www.a.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.a.com note.xsd">

  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The following fragment:

xmlns="http://www.a.com" specifies the default namespace declaration. This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.a.com" namespace.

Once you have the XML Schema Instance namespace available: **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**, you can use the *schemaLocation* attribute. This attribute has two values, separated by a space. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace:

```
xsi:schemaLocation="http://www.w3schools.com note.xsd"
```

XSD: Simple Types

XML Schemas define the elements of your XML files. A simple element is an XML element that contains only text. It cannot contain any other elements or attributes.

However, the "only text" restriction is quite misleading. The text can be of many different types. It can be one of the types included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself.

You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

Defining a Simple Element

The syntax for defining a simple element is:

For more notes visit <https://collegenote.pythonanywhere.com>

```
<xs:element name="xxx" type="yyy"/>
```

where xxx is the name of the element and yyy is the data type of the element.

XML Schema has a lot of built-in data types. The most common types are:

```
xs:string  
xs:decimal  
xs:integer  
xs:boolean  
xs:date  
xs:time
```

Example

Here are some XML elements:

```
<lastname>Refsnes</lastname>  
<age>36</age>  
<dateborn>1970-03-27</dateborn>
```

And here are the corresponding simple element definitions:

```
<xs:element name="lastname" type="xs:string"/>  
<xs:element name="age" type="xs:integer"/>  
<xs:element name="dateborn" type="xs:date"/>
```

Default and Fixed Values for Simple Elements

Simple elements may have a default value OR a fixed value specified. A default value is automatically assigned to the element when no other value is specified. In the following example the default value is "red":

```
<xs:element name="color" type="xs:string" default="red"/>
```

A fixed value is also automatically assigned to the element, and you cannot specify another value. In the following example the fixed value is "red":

```
<xs:element name="color" type="xs:string" fixed="red"/>
```

XSD Attribute

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the **attribute itself is always declared as a simple type**.

The syntax for defining an attribute is:

```
<xs:attribute name="xxx" type="yyy"/>
```

where xxx is the name of the attribute and yyy specifies the data type of the attribute.

Example

Here is an XML element with an attribute:

```
<lastname lang="EN">Smith</lastname>
```

And here is the corresponding attribute definition:

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

Default and Fixed Values for Attributes

Attributes may have a default value OR a fixed value specified. A default value is automatically assigned to the attribute when no other value is specified. In the following example the default value is "EN":

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
```

A fixed value is also automatically assigned to the attribute, and you cannot specify another value. In the following example the fixed value is "EN":

```
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

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

Restrictions on Content

When an XML element or attribute has a data type defined, it puts restrictions on the element's or attribute's content. If an XML element is of type "xs:date" and contains a string like "Hello World", the element will not validate. With XML Schemas, you can also add your own restrictions to your XML elements and attributes. These restrictions are called **facets**.

Facets/XML Restrictions

Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called **facets**.

Restrictions for Datatypes

Constraint	Description
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
Length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)

minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
Pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
Whitespace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

Restrictions on Values

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Set of Values

To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint. The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Series of Values

The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Element "initials" accepts value is THREE of the UPPERCASE letters from a to z:

```
<xs:element name="initials">
```

...

```
<xs:pattern value="[A-Z][A-Z][A-Z]"/>
...
</xs:element>
```

Other Restrictions on a Series of Values

The example below defines an element called "letter" with a restriction. The acceptable value is zero or more occurrences of lowercase letters from a to z:

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The next example also defines an element called "letter" with a restriction. The acceptable value is one or more pairs of letters, each pair consisting of a lower case letter followed by an upper case letter. For example, "sToP" will be validated by this pattern, but not "Stop" or "STOP" or "stop":

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z][A-Z]+" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The next example defines an element called "gender" with a restriction. The only acceptable value is male OR female:

```
<xs:element name="gender">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="male|female"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The next example defines an element called "password" with a restriction. There must be exactly eight characters in a row and those characters must be lowercase or uppercase letters from a to z, or a number from 0 to 9:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z0-9]{8}" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on Whitespace Characters

To specify how whitespace characters should be handled, we would use the `whiteSpace` constraint. This example defines an element called "address" with a restriction. The `whiteSpace` constraint is set to "preserve", which means that the XML processor WILL NOT remove any white space characters:

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The `whiteSpace` constraint is set to "replace", which means that the XML processor WILL REPLACE all white space characters (line feeds, tabs, spaces, and carriage returns) with spaces:

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="replace"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

This example also defines an element called "address" with a restriction. The `whiteSpace` constraint is set to "collapse", which means that the XML processor WILL REMOVE all white space characters (line feeds, tabs, spaces, carriage returns are replaced with spaces, leading and trailing spaces are removed, and multiple spaces are reduced to a single space):

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="collapse"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on Length

To limit the length of a value in an element, we would use the `length`, `maxLength`, and `minLength` constraints.

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

This example defines another element called "password" with a restriction. The value must be minimum five characters and maximum eight characters:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
</xs:restriction>  
</xs:simpleType>  
</xs:element>
```

XSD Complex Elements

A complex element contains other elements and/or attributes.

What is a Complex Element?

A complex element is an XML element that contains other elements and/or attributes.

There are four kinds of complex elements:

- **empty elements**
<product pid="1345"/>
- **elements that contain only other elements**
<employee>
 <firstname>John</firstname>
 <lastname>Smith</lastname>
</employee>
- **elements that contain only text**
<food type="dessert">Ice cream</food>
- **elements that contain both other elements and text**
<description>
 It happened on <date lang="norwegian">03.03.99</date>
</description>

Hey! Each of these elements may contain attributes as well!

How to Define a Complex Element

Look at this complex XML element, "employee", which contains only other elements:

```
<employee>  
  <firstname>John</firstname>  
  <lastname>Smith</lastname>  
</employee>
```

We can define a complex element in an XML Schema two different ways:

1. The "employee" element can be declared directly by naming the element, like this:

```
<xs:element name="employee">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="firstname" type="xs:string"/>  
      <xs:element name="lastname" type="xs:string"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>
```

If you use the method described above, only the "employee" element can use the specified complex type. Note that the child elements, "firstname" and "lastname", are surrounded by the <sequence> **indicator**. This means that the child elements must appear in the same order as they are declared.

2. The "employee" element can have a type attribute that refers to the name of the complex type to use:

```
<xs:element name="employee" type="personinfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

If you use the method described above, several elements can refer to the same complex type, like this:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>
```

```
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

You can also base a complex element on an existing complex element and add some elements, like this:

```
<xs:element name="employee" type="fullpersoninfo"/>
```

```
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Complex Types with Mixed Content

An XML element, "letter" that contains both text and other elements:

```
<letter>
  Dear Mr.<name>John Smith</name>.
  Your order <orderid>1032</orderid>
```

```
will be shipped on <shipdate>2001-07-13</shipdate>.  
</letter>
```

The following schema declares the "letter" element:

```
<xs:element name="letter">  
  <xs:complexType mixed="true">  
    <xs:sequence>  
      <xs:element name="name" type="xs:string"/>  
      <xs:element name="orderid" type="xs:positiveInteger"/>  
      <xs:element name="shipdate" type="xs:date"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>
```

Hey! To enable character data to appear between the child-elements of "letter", the mixed attribute must be set to "true". The <xs:sequence> tag means that the elements defined (name, orderid and shipdate) must appear in that order inside a "letter" element.

We could also give the complexType element a name, and let the "letter" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="letter" type="lettertype"/>  
  
<xs:complexType name="lettertype" mixed="true">  
  <xs:sequence>  
    <xs:element name="name" type="xs:string"/>  
    <xs:element name="orderid" type="xs:positiveInteger"/>  
    <xs:element name="shipdate" type="xs:date"/>  
  </xs:sequence>  
</xs:complexType>
```

XSD Indicators

We can control HOW elements are to be used in documents with indicators. There are seven indicators:

- Order indicators:
 - All
 - Choice
 - Sequence
- Occurrence indicators:
 - maxOccurs
 - minOccurs
- Group indicators:
 - Group name
 - attributeGroup name

Order Indicators

Order indicators are used to define the order of the elements.

All Indicator

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Choice Indicator

The <choice> indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="employee"/>
      <xs:element name="member" type="member"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Sequence Indicator

The <sequence> indicator specifies that the child elements must appear in a specific order:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Occurrence Indicators

Occurrence indicators are used to define how often an element can occur.

maxOccurs Indicator

The <maxOccurs> indicator specifies the maximum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The example above indicates that the "child_name" element can occur a minimum of one time (the default value for minOccurs is 1) and a maximum of ten times in the "person" element.

minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
        minOccurs="0" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The example above indicates that the "child_name" element can occur a minimum of zero times and a maximum of ten times in the "person" element.

Hey! For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) the default value for maxOccurs and minOccurs is 1.

Hey! To allow an element to appear an unlimited number of times, use the maxOccurs="unbounded" statement:

A working example:

An XML file called "Myfamily.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="family.xsd">

  <person>
    <full_name>Hege Refsnes</full_name>
    <child_name>Cecilie</child_name>
  </person>

  <person>
    <full_name>Tove Refsnes</full_name>
    <child_name>Hege</child_name>
    <child_name>Stale</child_name>
    <child_name>Jim</child_name>
    <child_name>Borge</child_name>
  </person>

  <person>
    <full_name>Stale Refsnes</full_name>
  </person>

</persons>
```

The XML file above contains a root element named "persons". Inside this root element we have defined three "person" elements. Each "person" element must contain a "full_name" element and it can contain up to five "child_name" elements.

Here is the schema file "family.xsd":

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:element name="persons">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="person" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="full_name" type="xs:string"/>
            <xs:element name="child_name" type="xs:string"
              minOccurs="0" maxOccurs="5"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Group Indicators

Group indicators are used to define related sets of elements.

Element Groups

Element groups are defined with the group declaration, like this:

```
<xs:group name="groupname">
```

```
  ...
```

```
</xs:group>
```

You must define an all, choice, or sequence element inside the group declaration. The following example defines a group named "persongroup", that defines a group of elements that must occur in an exact sequence:

```
<xs:group name="persongroup">
```

```
  <xs:sequence>
```

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

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

```
    <xs:element name="birthday" type="xs:date"/>
```

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

```
</xs:group>
```

After you have defined a group, you can reference it in another definition, like this:

```
<xs:group name="persongroup">
```

```
  <xs:sequence>
```

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

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

```
    <xs:element name="birthday" type="xs:date"/>
```

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

```
</xs:group>
```

```
<xs:element name="person" type="personinfo"/>
```

```
<xs:complexType name="personinfo">  
  <xs:sequence>  
    <xs:group ref="persongroup"/>  
    <xs:element name="country" type="xs:string"/>  
  </xs:sequence>  
</xs:complexType>
```

Attribute Groups

Attribute groups are defined with the attributeGroup declaration, like this:

```
<xs:attributeGroup name="groupname">
```

...

```
</xs:attributeGroup>
```

The following example defines an attribute group named "personattrgroup":

```
<xs:attributeGroup name="personattrgroup">  
  <xs:attribute name="firstname" type="xs:string"/>  
  <xs:attribute name="lastname" type="xs:string"/>  
  <xs:attribute name="birthday" type="xs:date"/>  
</xs:attributeGroup>
```

After you have defined an attribute group, you can reference it in another definition, like this:

```
<xs:attributeGroup name="personattrgroup">  
  <xs:attribute name="firstname" type="xs:string"/>  
  <xs:attribute name="lastname" type="xs:string"/>  
  <xs:attribute name="birthday" type="xs:date"/>  
</xs:attributeGroup>
```

```
<xs:element name="person">  
  <xs:complexType>  
    <xs:attributeGroup ref="personattrgroup"/>  
  </xs:complexType>  
</xs:element>
```

The <any> Element

The <any> element enables us to extend the XML document with elements not specified by the schema. By using the <any> element we can extend (after <lastname>) the content of "person" with any element:

```
<xs:element name="person">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="firstname" type="xs:string"/>  
      <xs:element name="lastname" type="xs:string"/>  
      <xs:any minOccurs="0"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>
```


For more notes visit <https://collegenote.pythonanywhere.com>

Now we want to extend the "person" element with a "children" element. In this case we can do so, even if the author of the schema above never declared any "children" element.

Look at this schema file, called "children.xsd":

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com" xmlns="http://www.w3schools.com"
elementFormDefault="qualified">

<xs:element name="children">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="childname" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "children.xsd":

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

<persons xmlns="http://www.microsoft.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.microsoft.com family.xsd http://www.w3schools.com children.xsd">

<person>
  <firstname>Hege</firstname>
  <lastname>Refsnes</lastname>
  <children>
    <childname>Cecilie</childname>
  </children>
</person>

<person>
  <firstname>Stale</firstname>
  <lastname>Refsnes</lastname>
</person>

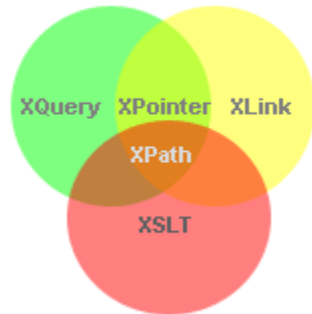
</persons>
```

The XML file above is valid because the schema "family.xsd" allows us to extend the "person" element with an optional element after the "lastname" element. The <any> and <anyAttribute> elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XPath

XPath is used to navigate through elements and attributes in an XML document. XPath is a major element in W3C's XSLT standard - and XQuery and XPointer are both built on XPath expressions.

What is XPath?



- XPath is a syntax for defining parts of an XML document
- XPath uses path expressions to navigate in XML documents
- XPath contains a library of standard functions
- XPath is a major element in XSLT
- XPath is a W3C recommendation

XPath Terminology

Nodes

In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes. XML documents are treated as trees of nodes. The topmost element of the tree is called the root element.

Let's begin with example:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
    <price>29.99</price>
  </book>
  <book>
    <title lang="en">Learning XML</title>
    <price>39.95</price>
  </book>
</bookstore>
```

Selecting Nodes

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	Description
<i>Nodename</i>	Selects all nodes with the name " <i>nodename</i> "
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node

..	Selects the parent of the current node
@	Selects attributes

In the table below we have listed some path expressions and the result of the expressions (for bookstore example)

Path Expression	Result
Bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore Note: If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

Predicates

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets. In the table below we have listed some path expressions with predicates and the result of the expressions:

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element. Note: In IE 5,6,7,8,9 first node is [0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath: <i>In JavaScript: <code>xml.setProperty("SelectionLanguage","XPath");</code></i>
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have an attribute named lang

	with a value of 'en'
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML elements.

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
/bookstore/*	Selects all the child nodes of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have any attribute

XQuery

XQuery is a language for finding and extracting elements and attributes from XML documents. It is to XML what SQL is to database tables and is designed to query XML data.

- XQuery is **the** language for querying XML data
- XQuery for XML is like SQL for databases
- XQuery is built on XPath expressions
- XQuery is supported by all major databases
- XQuery is a W3C Recommendation since 2007.

Here is an example of a question that XQuery could solve:

"Select all CD records with a price less than \$10 from the CD collection stored in the XML document called cd_catalog.xml"

XQuery can be used to:

- Extract information to use in a Web Service

- Generate summary reports
- Transform XML data to XHTML
- Search Web documents for relevant information

XQuery Example

- for \$x in doc("books.xml")/bookstore/book
 where \$x/price>30
 order by \$x/title
 return \$x/title

Let's start with example:

"books.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>

<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>

<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
```

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

Functions

XQuery uses functions to extract data from XML documents. The `doc()` function is used to open the "books.xml" file:

```
doc("books.xml")
```

Path Expressions

XQuery uses path expressions to navigate through elements in an XML document. The following path expression is used to select all the title elements in the "books.xml" file:

```
doc("books.xml")/bookstore/book/title
```

The XQuery above will extract the following:

```
<title lang="en">Everyday Italian</title>  
<title lang="en">Harry Potter</title>  
<title lang="en">XQuery Kick Start</title>  
<title lang="en">Learning XML</title>
```

Predicates

XQuery uses predicates to limit the extracted data from XML documents. The following predicate is used to select all the book elements under the bookstore element that have a price element with a value that is less than 30:

```
doc("books.xml")/bookstore/book[price<30]
```

The XQuery above will extract the following:

```
<book category="CHILDREN">  
  <title lang="en">Harry Potter</title>  
  <author>J K. Rowling</author>  
  <year>2005</year>  
  <price>29.99</price>  
</book>
```

FLWOR

FLWOR is an acronym for "For, Let, Where, Order by, Return". Explaining for example below:

The **for** clause selects all book elements under the bookstore element into a variable called \$x.

The **let clause** simply declares a variable and gives it a value:

```
let $maxCredit := 3000  
let $overdrawnCustomers := //customer[overdraft > $maxCredit]  
return count($overdrawnCustomers)
```

The **where** clause selects only book elements with a price element with a value greater than 30.

The **order by** clause defines the sort-order. Will be sort by the title element.

The **return** clause specifies what should be returned. Here it returns the title elements.

Example 1:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

Above xquery generates following output:

```
<title lang="en">Learning XML</title>
<title lang="en">XQuery Kick Start</title>
```

Example 2:

```
for $x in doc("books.xml")/bookstore/book
return if ($x/@category="CHILDREN")
then <child>{data($x/title)}</child>
else <adult>{data($x/title)}</adult>
```

Output:

```
<adult>Everyday Italian</adult>
<child>Harry Potter</child>
<adult>XQuery Kick Start</adult>
<adult>Learning XML</adult>
```

FLOWR and HTML (Present the Result In an HTML List)

Look at the following XQuery FLWOR expression:

```
for $x in doc("books.xml")/bookstore/book/title
order by $x
return $x
```

The expression above will select all the title elements under the book elements that are under the bookstore element, and return the title elements in alphabetical order. Now we want to list all the book-titles in our bookstore in an HTML list. We add and tags to the FLWOR expression:

```
<ul>
{
for $x in doc("books.xml")/bookstore/book/title
order by $x
return <li>{$x}</li>
}
</ul>
```

The result of the above will be:

```
<ul>
<li><title lang="en">Everyday Italian</title></li>
<li><title lang="en">Harry Potter</title></li>
<li><title lang="en">Learning XML</title></li>
<li><title lang="en">XQuery Kick Start</title></li>
</ul>
```

Now we want to eliminate the title element, and show only the data inside the title element:

```
<ul>
{
for $x in doc("books.xml")/bookstore/book/title
```

```
order by $x
return <li>{data($x)}</li>
}
</ul>
```

The result will be (an HTML list):

```
<ul>
<li>Everyday Italian</li>
<li>Harry Potter</li>
<li>Learning XML</li>
<li>XQuery Kick Start</li>
</ul>
```

XSL/XSLT

XSL stands for EXTensible Stylesheet Language, and is a style sheet language for XML documents. XSLT stands for XSL Transformations. In this section you will learn how to use XSLT to transform XML documents into other formats, like XHTML.

It Started with XSL

The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Stylesheet Language.

CSS = Style Sheets for HTML

HTML uses predefined tags, and the meaning of each tag is **well understood**.

The <table> tag in HTML defines a table - and a browser knows **how to display it**.

Adding styles to HTML elements are simple. Telling a browser to display an element in a special font or color is easy with CSS.

XSL = Style Sheets for XML

XML does not use predefined tags (we can use any tag-names we like), and therefore the meaning of each tag is **not well understood**.

A <table> tag could mean an HTML table, a piece of furniture, or something else - and a browser **does not know how to display it**.

XSL describes how the XML document should be displayed!

XSL - More Than a Style Sheet Language

XSL consists of three parts:

- XSLT - a language for transforming XML documents
- XPath - a language for navigating in XML documents
- XSL-FO - a language for formatting XML documents

What is XSLT?

- XSLT stands for XSL Transformations
- XSLT is the most important part of XSL
- XSLT transforms an XML document into another XML document
- All major browsers have support for XSLT.
- XSLT uses XPath to navigate in XML documents
- XSLT is a W3C Recommendation since 16 November 1999

XSLT = XSL Transformations

XSLT is the most important part of XSL and is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.

With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

A common way to describe the transformation process is to say that XSLT transforms an XML source-tree into an XML result-tree.

XSLT Uses XPath

XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents.

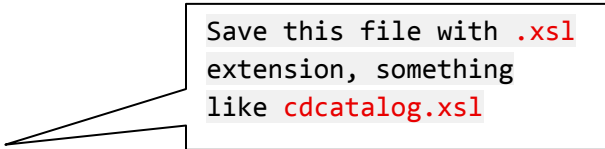
How does it Work?

In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

XSLT Example

```
<?xml version="1.0"?>
<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 bgcolor="#9acd32">
        <th>Title</th>
        <th>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>
```



Save this file with `.xsl` extension, something like `cdcatalog.xsl`

Hey! `<xsl:stylesheet>` and `<xsl:transform>` are completely synonymous and either can be used.

And we can reference this XSL style sheet to xml document as:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
  .
  .
</catalog>
```

The `<xsl:template>` Element

A template contains rules to apply when a specified node is matched. The **match** attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document. The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).

The `<xsl:value-of>` Element

The `<xsl:value-of>` element can be used to extract the value of an XML element and add it to the output stream of the transformation.

```
<td><xsl:value-of select="catalog/cd/title"/></td>
<td><xsl:value-of select="catalog/cd/artist"/></td>
```

Hey! The **select** attribute, in the example above, contains an XPath expression.

The `<xsl:for-each>` Element

The XSL `<xsl:for-each>` element can be used to select every XML element of a specified node-set.

Filtering the Output

We can also filter the output from the XML file by adding a criterion to the select attribute in the `<xsl:for-each>` element.

```
<xsl:for-each select="catalog/cd[artist='Bob Dylan']">
```

Legal filter operators are:

- = (equal)
- != (not equal)
- < less than
- > greater than

Example:

```
<xsl:for-each select="catalog/cd[artist='Bob Dylan']">
  <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
  </tr>
</xsl:for-each>
```

XSLT `<xsl:sort>` Element

The `<xsl:sort>` element is used to sort the output. To sort the output, simply add an `<xsl:sort>` element inside the `<xsl:for-each>` element in the XSL file:

```
<xsl:for-each select="catalog/cd">
  <xsl:sort select="artist"/>
  <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
  </tr>
</xsl:for-each>
```

XSLT `<xsl:if>` Element

The `<xsl:if>` element is used to put a conditional test against the content of the XML file. To add a conditional test, add the `<xsl:if>` element inside the `<xsl:for-each>` element in the XSL file:

```
<xsl:if test="expression">
  ...some output if the expression is true...
</xsl:if>
```

Example:

```
<xsl:for-each select="catalog/cd">
  <xsl:if test="price > 10">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
      <td><xsl:value-of select="price"/></td>
    </tr>
  </xsl:if>
</xsl:for-each>
```

XSLT `<xsl:choose>` Element

The `<xsl:choose>` element is used in conjunction with `<xsl:when>` and `<xsl:otherwise>` to express multiple conditional tests.

Syntax

```
<xsl:choose>
  <xsl:when test="expression">
    ... some output ...
  </xsl:when>
  <xsl:otherwise>
    ... some output ....
  </xsl:otherwise>
</xsl:choose>
```

To insert a multiple conditional test against the XML file, add the `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>` elements to the XSL file:

Example

```
<xsl:for-each select="catalog/cd">
  <tr>
    <td><xsl:value-of select="title"/></td>
    <xsl:choose>
      <xsl:when test="price > 10">
        <td bgcolor="#ff00ff">
          <xsl:value-of select="artist"/></td>
        </xsl:when>
        <xsl:otherwise>
          <td><xsl:value-of select="artist"/></td>
        </xsl:otherwise>
      </xsl:choose>
    </tr>
  </xsl:for-each>
```

Another Example

Here is another example that contains two <xsl:when> elements:

```
<?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 bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
      <tr>
        <td><xsl:value-of select="title"/></td>
        <xsl:choose>
          <xsl:when test="price > 10">
            <td bgcolor="#ff00ff">
              <xsl:value-of select="artist"/></td>
            </xsl:when>
            <xsl:when test="price > 9">
              <td bgcolor="#cccccc">
                <xsl:value-of select="artist"/></td>
            </xsl:when>
            <xsl:otherwise>
              <td><xsl:value-of select="artist"/></td>
            </xsl:otherwise>
          </xsl:choose>
        </tr>
      </xsl:for-each>
    </table>
  </body>
```

For more notes visit <https://collegenote.pythonanywhere.com>

```
</html>  
</xsl:template>  
  
</xsl:stylesheet>
```

Unit 4 Server Tier

Web Server Concepts

A **web server** is a computer system that processes requests via HTTP, the basic network protocol used to distribute information on the World Wide Web. The term can refer either to the entire system, or specifically to the software that accepts and supervises the HTTP requests. The most common use of web servers is to host websites, but there are other uses such as gaming, data storage, running enterprise applications, handling email, FTP, or other web uses.

The **primary function of a web server** is to store, process and deliver web pages to clients. Pages delivered are most frequently HTML documents, which may include images, style sheets and scripts in addition to text content. A user agent, commonly a web browser or web crawler, initiates communication by making a request for a specific resource using HTTP and the server responds with the content of that resource or an error message if unable to do so. The resource is typically a real file on the server's secondary storage. While the primary function is to serve content, a full implementation of HTTP also includes ways of receiving content from clients. This feature is used for submitting web forms, including uploading of files.

Many generic web servers also support server-side scripting using Active Server Pages (ASP), PHP, or other scripting languages. This means that the behavior of the web server can be scripted in separate files, while the actual server software remains unchanged. Usually, this function is used to create HTML documents **dynamically** ("on-the-fly") as opposed to returning **static documents**. The former is primarily used for retrieving and/or modifying information from databases. The latter is typically much faster and more easily cached but cannot deliver dynamic content.

Web server features

- **Virtual hosting:** to serve many web sites using one IP address
- **Large file support:** to be able to serve files whose size is greater than 2 GB
- **Bandwidth throttling:** to limit the speed of responses in order to not saturate the network and to be able to serve more clients
- **Server-side scripting:** to generate dynamic web pages, still keeping web server and website implementations separate from each other

Kernel-mode and user-mode web servers

A web server can be either implemented into the OS kernel, or in user space (like other regular applications). An **in-kernel web server** (like Microsoft IIS on Windows or TUX on GNU/Linux) will usually work faster, because, as part of the system, it can directly use all the hardware resources it needs, such as non-paged memory, CPU time-slices, network adapters, or buffers.

Web servers that run in **user-mode** have to ask the system for permission to use more memory or more CPU resources.

Load limits

A web server (program) has defined load limits, because it can handle only a limited number of concurrent client connections (usually between 2 and 80,000, by default between 500 and 1,000) per IP address (and TCP port) and it can serve only a certain maximum number of requests per second depending on:

- its own settings,
- the HTTP request type,
- whether the content is static or dynamic,
- whether the content is cached, and
- The hardware and software limitations of the OS of the computer on which the web server runs.

When a web server is near to or over its limit, it becomes unresponsive.

Causes of overload

At any time web servers can be overloaded because of:

- **Too much legitimate web traffic:** Thousands or even millions of clients connecting to the web site in a short interval, e.g., Slashdot effect.
- **Distributed Denial of Service attacks:** A denial-of-service attack (DoS attack) or distributed denial-of-service attack (DDoS attack) is an attempt to make a computer or network resource unavailable to its intended users.
- **Computer worms** that sometimes cause abnormal traffic because of millions of infected computers (not coordinated among them).
- **XSS viruses** can cause high traffic because of millions of infected browsers and/or web servers.
- **Internet bots** Traffic not filtered/limited on large web sites with very few resources (bandwidth, etc.).
- **Internet (network) slowdowns**, so that client requests are served more slowly and the number of connections increases so much that server limits are reached;
- **Web servers (computers) partial unavailability:** This can happen because of required or urgent maintenance or upgrade, hardware or software failures, back-end (e.g., database) failures, etc.; in these cases the remaining web servers get too much traffic and become overloaded.

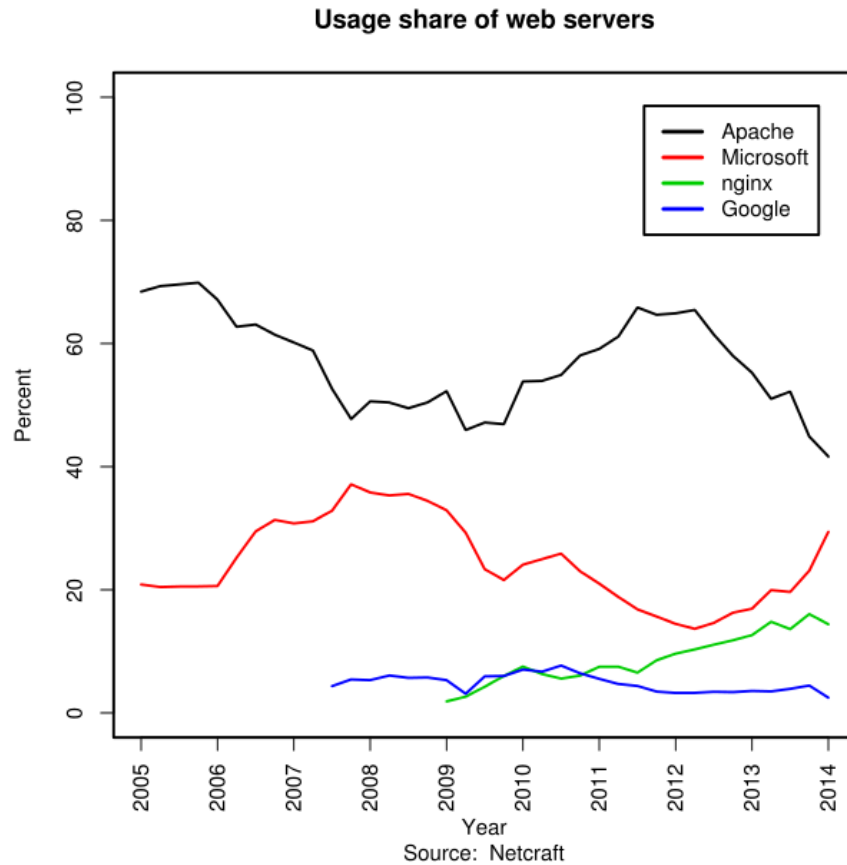
Anti-overload techniques

To partially overcome above average load limits and to prevent overload, most popular web sites use common techniques like:

- Managing network traffic, by using:
 - **Firewalls** to block unwanted traffic coming from bad IP sources or having bad patterns
 - **HTTP traffic managers** to drop, redirect or rewrite requests having bad HTTP patterns
 - **Bandwidth management and traffic shaping**, in order to smooth down peaks in network usage.
- Deploying **web cache** techniques
- **Using different domain names** to serve different (static and dynamic) content by separate web servers, i.e.:
 - <http://images.example.com>
 - <http://www.example.com>
- Using different domain names and/or computers to separate big files from small and medium sized files; the idea is to be able to fully cache small and medium sized files and to efficiently serve big or huge (over 10 - 1000 MB) files by using different settings
- Using many web servers (programs) per computer, each one bound to its own network card and IP address.
- Using many web servers (computers) that are grouped together behind a load balancer so that they act or are seen as one big web server
- Adding more hardware resources (i.e. RAM, disks) to each computer
- Tuning OS parameters for hardware capabilities and usage

Market share of major web servers

Below are the statistics of the market share of the top web servers on the Internet by April, May 2014.



Product	Vendor	April 2014	Percent	May 2014	Percent
Apache	Apache	361,853,003	37.74%	366,262,346	37.56%
IIS	Microsoft	316,843,695	33.04%	325,854,054	33.41%
nginx	NGINX, Inc.	146,204,067	15.25%	142,426,538	14.60%
GWS	Google	20,983,310	2.19%	20,685,165	2.12%

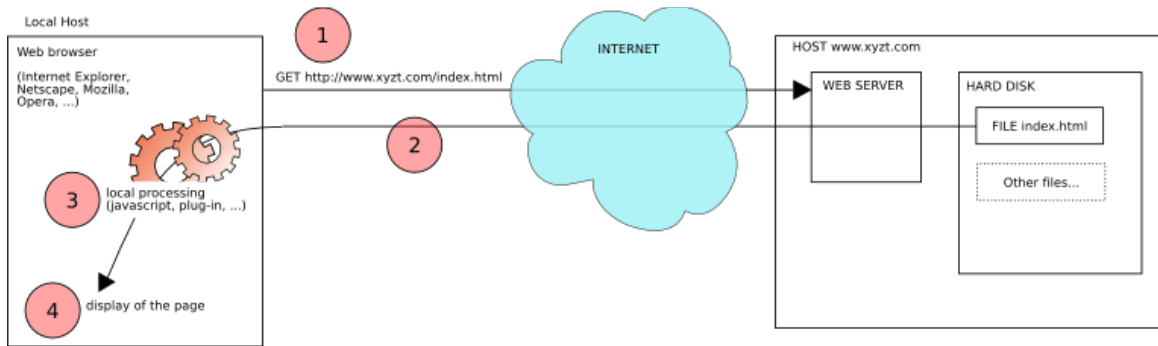
Apache, IIS and Nginx are the most used web servers on the Internet.

Creating Dynamic Content

Classical hypertext navigation, with HTML or XHTML alone, provides "static" content, meaning that the user requests a web page and simply views the page and the information on that page. However, a web page can also provide a "live", "dynamic", or "interactive" user experience. Content (text, images, form fields, etc.) on a web page can change, in response to different contexts or conditions.

There are two ways to create this kind of effect:

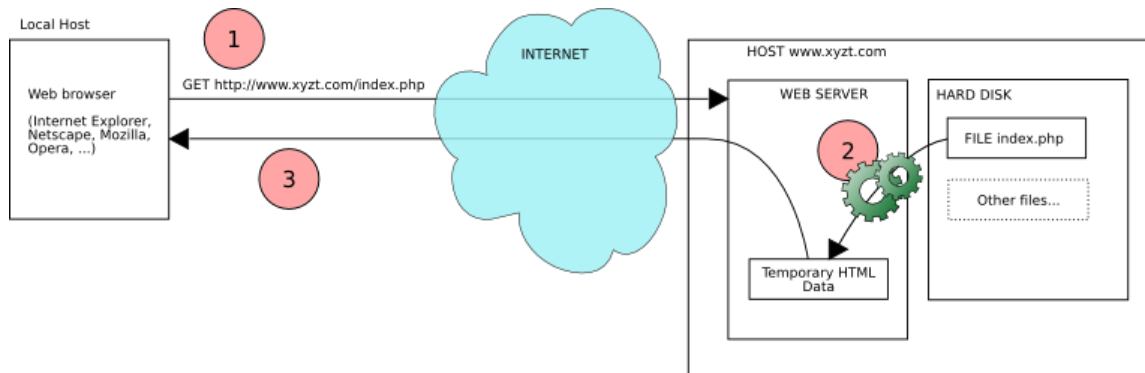
- Using **client-side scripting** to change interface behaviors *within* a specific web page, in response to mouse or keyboard actions or at specified timing events. In this case the dynamic behavior occurs within the presentation.



This shows how client-side processing occurs:

1. The web browser sends a request for an HTML file
 2. The web server sends it back
 3. The web browser receives the **index.html** file and processes javascript, plug-ins (java, audio, video,...) (this information is included in the **index.html** file)
 4. The web browser displays the page
- Using **server-side scripting** to change the supplied page source *between* pages, adjusting the sequence or reload of the web pages or web content supplied to the browser. Server responses may be determined by such conditions as data in a posted HTML form, parameters in the URL, the type of browser being used, the passage of time, or a database or server state.

The following diagram shows server-side scripting; on how a web server will handle a request for a PHP file (PHP is an example, and many other methods behave the same way - ASP, cgi-bin programs etc.):



1. The browser sends a GET request to the remote web server for a PHP file
2. The web server recognizes the **.php** extension, and does a special processing on the PHP file in order to generate a temporary file
3. The web server sends this temporary file back to the user

Important! Chapter 4 and 5 incurs specially a lab works, we will use asp.net mvc framework for all the programs we implement. Asp.net mvc is the latest (since 2009) web development framework from Microsoft and getting good responses from web world. Other frameworks like ruby on rails, django on python, groovy on rails, spring for java etc works similarly with few architectural and server-side compiler differences. Also, for our lab purpose, we stick with C# for server side processing. So altogether **asp.net mvc with c#** will be our implementation model.

State Management Overview

In traditional Web programming, all the information associated with the page and the controls on the page would be lost with each round trip. For example, if a user enters information into a text box, that information would be lost in the round trip from the browser or client device to the server.

To overcome this inherent limitation of traditional Web programming, Frameworks like ASP.NET includes several options that help you preserve data on both a per-page basis and an application-wide basis. These features are as follows:

- View state
- Control state
- Hidden fields
- Cookies
- Query strings
- Application state
- Session state
- Profile Properties

View state, control state, hidden fields, cookies, and query strings all involve storing data on the client in various ways. However, application state, session state, and profile properties all store data in memory on the server. Each option has distinct advantages and disadvantages, depending on the scenario.

The following sections describe options for state management specific to **ASP.NET** that involve storing information either in the page or on the client computer. For these options, no information is maintained on the server between round trips.

Client based state management options

View State

The ViewState property provides a dictionary object for retaining values between multiple requests for the same page. This is the default method that the page uses to preserve page and control property values between round trips.

When the page is processed, the current state of the page and controls is hashed into a string and saved in the page as a hidden field, or multiple hidden fields if the amount of data stored in the ViewState property exceeds the specified value in the MaxPageStateFieldLength property. When the page is posted back to the server, the page parses the view-state string at page initialization and restores property information in the page. You can store values in view state as well. The following example shows how to store a value in the view state.

```
[C# code]  
ViewState["color"] = "red";
```

Hidden Fields

ASP.NET allows you to store information in a HiddenField control, which renders as a standard HTML hidden field. A hidden field does not render visibly in the browser, but you can set its properties just as you can with a standard control. When a page is submitted to the server, the content of a hidden field is sent in the HTTP form collection along with the values of other controls. A hidden field acts as a repository for any page-specific information that you want to store directly in the page.

```
<asp:hiddenfield id="ExampleHiddenField"  
  value="Example Value"  
  runat="server"/>
```

Cookies

A cookie is a small amount of data that is stored either in a text file on the client file system or in-memory in the client browser session. It contains site-specific information that the server sends to the client along with page output. Cookies can be temporary (with specific expiration times and dates) or persistent.

You can use cookies to store information about a particular client, session, or application. The cookies are saved on the client device, and when the browser requests a page, the client sends the information in the cookie along with the request information. The server can read the cookie and extract its value. A typical use is to store a token (perhaps encrypted) indicating that the user has already been authenticated in your application.

The following example shows how to write a cookie.

[C# code]

```
Response.Cookies["destination"].Value = "CA";  
Response.Cookies["destination"].Expires = DateTime.Now.AddDays(1);
```

Here we are creating cookies server side with C#, we can also manipulate cookies in client using JavaScript, **please see unit 1 page no. 74-75.**

Query Strings

A query string is information that is appended to the end of a page URL. A typical query string might look like the following example:

<http://www.contoso.com/listwidgets.aspx?category=basic&price=100>

In the URL path above, the query string starts with a question mark (?) and includes two attribute/value pairs, one called "category" and the other called "price."

Query strings provide a simple but limited way to maintain state information. For example, they are an easy way to pass information from one page to another, such as passing a product number from one page to another page where it will be processed. However, some browsers and client devices impose a 2083-character limit on the length of the URL.

In order for query string values to be available during page processing, you must submit the page using an HTTP GET command. That is, you cannot take advantage of a query string if a page is processed in response to an HTTP POST command.

Server based state management options

ASP.NET offers you a variety of ways to maintain state information on the server, rather than persisting information on the client. With server-based state management, you can decrease the amount of information sent to the client in order to preserve state, however it can use costly resources on the server. The following sections describe three server-based state management features: application state, session state, and profile properties.

Application State

ASP.NET allows you to save values using application state - which is an instance of the `HttpApplicationState` class - for each active Web application. Application state is a global storage mechanism that is accessible from all pages in the Web application. Thus, application state is useful for storing information that needs to be maintained between server round trips and between requests for pages.

Application state is stored in a key/value dictionary that is created during each request to a specific URL. You can add your application-specific information to this structure to store it between page requests.

Once you add your application-specific information to application state, the server manages it. The following example shows how to assign a value in application state.

[C# code]

```
Application["WelcomeMessage"] = "Welcome to the Contoso site.";
```

Session State

ASP.NET allows you to save values by using session state - which is an instance of the `HttpSessionState` class - for each active Web-application session. Session state is similar to application state, except that it is scoped to the current browser session. If different users are using your application, each user session will have a different session state. In addition, if a user leaves your application and then returns later, the second user session will have a different session state from the first.

Session state is structured as a key/value dictionary for storing session-specific information that needs to be maintained between server round trips and between requests for pages.

You can use session state to accomplish the following tasks:

- Uniquely identify browser or client-device requests and map them to an individual session instance on the server.
- Store session-specific data on the server for use across multiple browser or client-device requests within the same session.
- Raise appropriate session management events. In addition, you can write application code leveraging these events.

Once you add your application-specific information to session state, the server manages this object. Depending on which options you specify, session information can be stored in cookies, on an out-of-process server, or on a computer running Microsoft SQL Server.

The following example shows how to save a value in session state.

[C# code]

```
Session["FirstName"] = FirstNameTextBox.Text;  
Session["LastName"] = LastNameTextBox.Text;
```

Tag libraries

Tag libraries help us write server side code within view file (html file in fact) with the help of view engines in place. ASP.NET MVC supports various view engines like Razor, Aspx, Spark etc with their own syntactical structure and own view file extension (*.cshtml for Razor with C#).

Creating dynamic content with tag libraries:

Suppose we have simple person class in C#

For more notes visit <https://collegenote.pythonanywhere.com>

```
public class Person
{
    public string Name { get; set; }
    public string Address { get; set; }
    public int Cell { get; set; }
    public bool IsMale { get; set; }
    public DateTime DOB { get; set; }
}
```

If we have collection of this person class (List<Person>) may be in-memory or database table fetched, we can render this collection in view file in MVC with Razor as:

```
@model IEnumerable<Person>

<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Address</th>
            <th>Date Of Birth</th>
            <th>Cell</th>
            <th>IsMale</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var person in Model)
        {
            <tr>
                <td>@person.Name</td>
                <td>@person.Address</td>
                <td>@person.DOB</td>
                <td>@person.Cell</td>
                <td>@person.IsMale</td>
            </tr>
        }
    </tbody>
</table>
```

Here we are using loop control to generate all people information within table rows dynamically.

Error (Exception) Handling

Error handling allows us to resolve unexpected failovers of web application for different level of audiences (User, devs, admin etc.). In asp.net mvc application, we can handle errors at different level:

1. Local level exception handling
 - i. Simple try-catch approach

```
public ActionResult TestMethod()
{
    try
    {
        //....
        return View();
    }
    catch (Exception e)
    {
        //Handle Exception;
        return View("Error");
    }
}
```

```
}  
}
```

ii. Override OnException Method in controller

```
protected override void OnException(ExceptionContext filterContext)  
{  
    Exception e = filterContext.Exception;  
    //Log Exception e  
    filterContext.ExceptionHandled=true;  
    filterContext.Result = new ViewResult()  
    {  
        ViewName = "Error"  
    };  
}
```

2. Global level exception handling

i. Using FilterConfig class

```
public class FilterConfig  
{  
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)  
    {  
        filters.Add(new HandleErrorAttribute());  
    }  
}
```

It handles all the exceptions raised by all action methods in all the controllers and return error view present inside shared folder.

→ Handling error for specific controller using HandleErrorAttribute:

```
[HandleError()]  
public class TestingController : Controller
```

OR

```
[HandleError(ExceptionType=typeof(DivideByZeroException),View="DivideZeroError")]  
[HandleError(ExceptionType = typeof(NullReferenceException), View = "NullError")]  
public class TestingController : Controller
```

→ Handle error at action level:

```
public class TestingController : Controller  
{  
    [HandleError()]  
    public ActionResult TestMethod()  
    {
```

Unit 5: Advanced server side issues

Database connectivity

For data intensive applications, we need a resilient connection to backing store. Every languages/frameworks have their own conventions and driver sets to connect to database. In .NET, ADO.NET (Active Data Object) has responsibility to connect to relational databases like Oracle, Sql server, MySql etc. ADO.NET is also a part of the .NET Framework and is used to handle data access services in .NET platform.

You can use **ADO.NET** to access data by using the new .NET Framework data providers which are:

- Data Provider for SQL Server (System.Data.SqlClient).
- Data Provider for OLEDB (System.Data.OleDb).
- Data Provider for ODBC (System.Data.Odbc).
- Data Provider for Oracle (System.Data.OracleClient).

ADO.NET is a set of classes that expose data access services to the .NET developer. The ADO.NET classes are found in System.Data.dll and are integrated with the XML classes in System.Xml.dll. There are two central components of ADO.NET classes: the DataSet, and the .NET Framework Data Provider.

Data Provider is a set of components including:

- the **Connection object** (SqlConnection, OleDbConnection, OdbcConnection, OracleConnection)
- the **Command object** (SqlCommand, OleDbCommand, OdbcCommand, OracleCommand)
- the **DataReader object** (SqlDataReader, OleDbDataReader, OdbcDataReader, OracleDataReader)
- and the **DataAdapter Object** (SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter, OracleDataAdapter).

DataSet object represents a disconnected cache of data which is made up of DataTables and DataRelations that represent the result of the command.

For above set of classes we need to use raw sql queries for **select, insert, update and delete**:

Assuming Student table in db

	Name	Data Type	Allow Nulls	Default
PK	Id	int	<input type="checkbox"/>	
	Name	nvarchar(200)	<input type="checkbox"/>	
	Address	nvarchar(500)	<input checked="" type="checkbox"/>	
	BloodGroup	varchar(10)	<input checked="" type="checkbox"/>	
	Cell	int	<input checked="" type="checkbox"/>	
	DOB	date	<input checked="" type="checkbox"/>	
			<input type="checkbox"/>	

```
CREATE TABLE [dbo].[Student] (  
    [Id] INT IDENTITY (1, 1) NOT NULL,  
    [Name] NVARCHAR (200) NOT NULL,  
    [Address] NVARCHAR (500) NULL,  
    [BloodGroup] VARCHAR (10) NULL,  
    [Cell] INT NULL,  
    [DOB] DATE NULL,  
    PRIMARY KEY CLUSTERED ([Id] ASC)  
);
```

--Selection

For more notes visit <https://collegenote.pythonanywhere.com>

```
select * from Student
```

```
--Inserting into table
```

```
insert into Student values ('Ramesh Niraula', 'ktm-89', 'AB+', 9845876, '1945-02-02');
```

```
--Updating table row
```

```
update Student set Name='Ram Niraula' where Id=2;
```

```
--delete a table row
```

```
delete from Student where Name = 'Ram Niraula';
```

We also need to provide a **connection string** for a database used. Something like:

```
public string conString = @"Provider=Microsoft.Jet.OLEDB.4.0;DataSource=..\..\PersonDatabase.mdb";
```

OR in web.config configuration file as:

```
<add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-WebApp-20141201101220;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-WebApp-20141201101220.mdf" providerName="System.Data.SqlClient" />
```

```
<add name="TestEntities" connectionString="metadata=res://*/Models.DbTest.csdl|res://*/Models.DbTest.ssdl|res://*/Models.DbTest.msl;provider=System.Data.SqlClient;provider connection string="data source=(LocalDb)\v11.0;initial catalog=DbTest;integrated security=True;pooling=False;multipleactiveresultsets=True;application name=EntityFramework";" providerName="System.Data.EntityClient" />
```

Today's database development in .NET world uses ORM (Object Relational Mapping) tools like Entity Framework (EF), NHibernate etc. These tools provide productivity to a developer abstracting ADO.NET data access methods. We have already used EF in our basic projects using entity data model of entity framework (EF) database-first where EF created context and model classes as:

```
public partial class TestEntities : DbContext
{
    public TestEntities()
        : base("name=TestEntities") //TestEntities is connection string
    {
    }

    public DbSet<Student> Students { get; set; }
}
```

And student model,

```
public partial class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string BloodGroup { get; set; }
    public Nullable<int> Cell { get; set; }
    public Nullable<System.DateTime> DOB { get; set; }
}
```



```
Then it is just the matter of querying models,  
private TestEntities db = new TestEntities();  
var students = db.Students.ToList();  
var student = db.Students.Find(id);  
db.Students.Add(student);  
db.SaveChanges();  
...other operations works similarly
```

File Handling

We already did a dictionary project where we use a plain text file as a data source. Text data is fetched using API (Application Programming Interface) from **System.IO** (in .NET). This namespace provides bunch of reader/writer classes to handle files and directories.

```
StreamReader r = new StreamReader(Path.Combine(  
    System.Web.HttpContext.Current.Request.PhysicalApplicationPath,  
    "App_Data\\Data.txt"));  
var allWords = r.ReadToEnd();  
r.Dispose();
```

Instead of streamReader, we could have used BinaryReaders, TextReaders etc provided by System.IO.

Form Handling

Html forms can be handled by ASP.NET MVC pretty easily to collect some data from the user:

```
<form action="/test/addstudent" method="post">  
    <label>Name</label>  
    <input id="Name" name="Name" type="text" value="">  
    <label>Address</label>  
    <textarea id="Address" name="Address"></textarea>  
    <label>Blood Group</label>  
    <input id="BloodGroup" name="BloodGroup" type="text" value="">  
    <label>Phone Number</label>  
    <input data-val="true" data-val-number="The field Cell must be a number." id="Cell"  
name="Cell" type="text" value="">  
    <label>Date of birth</label>  
    <input data-val="true" data-val-date="The field DOB must be a date." id="DOB"  
name="DOB" type="text" value="">  
    <br />  
    <input type="submit" value="Create">  
</form>
```

OR using Html Helpers given Razor view engine, same form can be created as:

```
@using (Html.BeginForm("AddStudent", "Test", HttpVerbs.Post))  
{  
    <label>Name</label>  
    @Html.TextBoxFor(x => x.Name);  
    <label>Address</label>  
    @Html.TextBoxFor(x => x.Address);  
    <label>Blood Group</label>  
    @Html.TextBoxFor(x => x.BloodGroup);  
    <label>Phone Number</label>  
    @Html.TextBoxFor(x => x.Cell);  
    <label>Date of birth</label>
```

For more notes visit <https://collegenote.pythonanywhere.com>

```
@Html.TextBoxFor(x => x.DOB);  
  
<input type="submit" value="Create" />  
}
```

The view file containing above code must be strongly typed (Ability to accept some model) to gather form data as:

```
@model WebApp.Models.Student
```

Then in controller action,

```
[HttpPost]  
public ActionResult AddStudent(Student model)  
{  
    TestEntities database = new TestEntities();  
    database.Students.Add(model);  
    database.SaveChanges();  
    return View();  
}
```

Authentication

Authentication is the process of obtaining identification credentials such as name and password from a user and validating those credentials against some authority. If the credentials are valid, the entity that submitted the credentials is considered an authenticated identity. Once an identity has been authenticated, the authorization process determines whether that identity has access to a given resource.

ASP.NET implements authentication through authentication providers, the code modules that contain the code necessary to authenticate the requestor's credentials. The topics in this section describe the authentication providers built into ASP.NET.

Term	Definition
Windows Authentication (Integrated Windows Authentication)	In this methodology ASP.NET web pages will use local windows users and groups to authenticate and authorize resources. Provides information on how to use Windows authentication in conjunction with Microsoft Internet Information Services (IIS) authentication to secure ASP.NET applications.
Forms Authentication	Provides information on how to create an application-specific login form and perform authentication using your own code. A convenient way to work with forms authentication is to use ASP.NET membership and ASP.NET login controls, which together provide a way to collect user credentials, authenticate them, and manage them, using little or no code.
Passport authentication (OpenId authentication)	Passport authentication is based on the passport website provided by the Microsoft, Google, Facebook and Twitter. So when user logs in with credentials it will be reached to the passport website where authentication will happen. If Authentication is successful it will return a token to your website.
Anonymous access	If you do not want any kind of authentication then you will go for Anonymous access.

For more notes visit <https://collegenote.pythonanywhere.com>