

KARNATAKA STATE OPEN UNIVERSITY
MUKTHAGANGOTRI, MYSORE- 570 006

DEPARTMENT OF STUDIES IN INFORMATION TECHNOLOGY



M.Sc IN INFORMATION TECHNOLOGY
IV SEMESTER



WEB TECHNOLOGIES
MSIT- 119

MODULE: 1-4

MSIT-119:

Web Technologies

Course Design and Editorial Committee

Prof. M.G.Krishnan

Vice Chancellor

Karnataka State Open University

Mukthagangotri, Mysore – 570 006

Prof. Vikram Raj Urs

Dean (Academic) & Convener

Karnataka State Open University

Mukthagangotri, Mysore – 570 006

Head of the Department and Course Co-Ordinator

Rashmi B.S

Assistant Professor & Chairperson

DoS in Information Technology

Karnataka State Open University

Mukthagangotri, Mysore – 570 006

Course Editor

Ms. Nandini H.M

Assistant professor of Information Technology

DoS in Information Technology

Karnataka State Open University

Mukthagangotri, Mysore – 570 006

Course Writers

Dr. Vinay

Assistant Professor,

PG Department of Computer Science,

JSS College of Arts, Commerce & Science,

Ooty road, Mysore

Dr. Chethan H K

Associate Professor,

Dept of Computer Science,

Maharaja Institue of Technology,

Mysore.

Publisher

Registrar

Karnataka State Open University

Mukthagangotri, Mysore – 570 006

Developed by Academic Section, KSOU, Mysore

Karnataka State Open University, 2014

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Karnataka State Open University.

Further information on the Karnataka State Open University Programmes may be obtained from the University's Office at Mukthagangotri, Mysore – 6.

Printed and Published on behalf of Karnataka State Open University, Mysore-6 by the **Registrar (Administration)**



Karnataka State Open University

Muktagangothri, Mysore – 570 006

Master of Science in Information Technology

MSIT – 119 Web Technologies

Module 1

Unit-1	Web Fundamentals	02-45
Unit-2	Web security and Web programmers toolbox	46-70
Unit-3	Evolution of HTML	71-95
Unit-4	Hypertext and Markup languages	96-116

Module 2

Unit-5	Style Sheet	117-133
Unit-6	Web page properties and formatting	134-160
Unit-7	Tags	161-176
Unit-8	Case Study: Conflict Resolution	177-192

Module 3

Unit-9	JAVA Script introduction	194-211
Unit-10	JAVA Script: Fundamentals of programming	212-233
Unit-11	Advance JAVA script: Functions and Constructors	234-249
Unit-12	XML	250-294

Module 4

Unit-13	Introduction Perl and CGI programming	295-311
Unit-14	Perl and CGI Programming: Language Basics	312-330
Unit-15	Advance programming concepts in Perl and CGI	331-350
Unit-16	Servlets and JAVA server pages	351-397

Preface

There are many web technologies simple to complex and explaining them in detail is a primary objective of this study material. Understanding web technologies will help one to develop their own web sites. This book provides brief definitions of the major Web technologies along with reference to the external links for advance studies.

Overall structure of the study materials is organized into four modules. Each module consist of four units. Every modules is designed in such a way that it introduces one technology and discusses the merits and demerits in comparison with existing problems.

In brief, module1 discusses fundamentals of web, web browser, web servers and markup languages such as HTML and XHTML. Module 2 covers Introduces different levels of style sheets, style specification formats, selector forms and property value forms. In this module we also cover the some advance usage of tags in demonstration conflict resolution examples. In module 3 we introduce and explore one of the popular web technology called java scripts. This covers very basic programming to advanced concepts like pattern matching and expression evaluation. In the last unit of this module we have brief about XML technology and cascade style sheets. In the last module, we introduce another web technology program PERL. Perl is regarded as one of the one powerful and widely used web technology language for scripting World Wide Web. This module also covers CGI, servlets and java server pages. Some advance algorithms for pattern matching problems are also discussed. In the materials we have provided sufficient programming example to clearly demonstrate the work flow of the technologies. In the reference section we have given an external links for the readers to get addition resources on the topic.

Wish you all happy reading.

Module-1

UNIT 1:

Structure:

- 1.0 Objectives
- 1.1 Introduction
- 1.2 Fundamental of Web
- 1.3 Internet
- 1.4 World-Wide Web
- 1.5 Web Browsers
- 1.6 Web Servers
- 1.7 Uniform Resource Locator (URL)
- 1.8 Summary
- 1.9 Keywords
- 1.10 Unit-end exercises and answers
- 1.11 Suggested readings

1.0 OBJECTIVES

At the end of this unit you will be able to know:

- Understand the fundamental of web and Internet
- Explanation of World Wide Web
- Web Browsers
- Web Servers
- Understand the working of Uniform Resource Locator

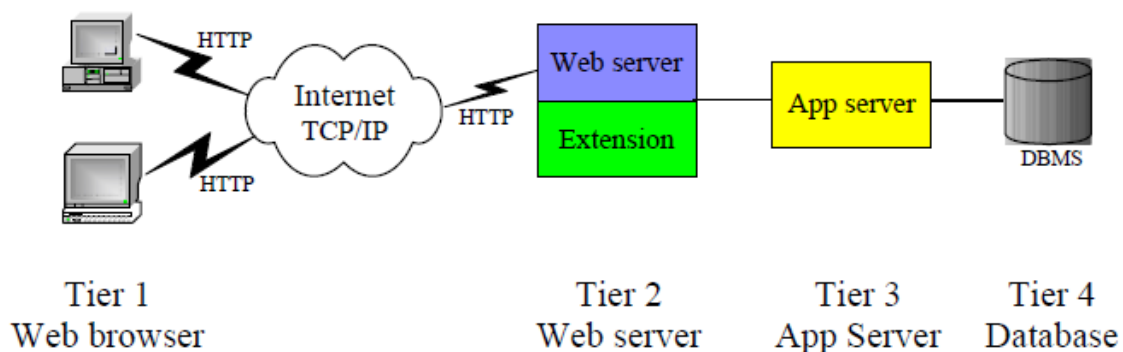
1.1 INTRODUCTION

Web servers and web browsers are communicating client-server computer programs for distributing documents and information, generally called web data, over the Internet. Web data are marked up in the HTML language for presentation and interaction with people in web browsers. Each web server uses an IP address or domain name as well as a port number for its identification. People use web browsers to send data requests to web servers with the HTTP protocol, and the web servers running on server computers either retrieve the requested data from local disks or generate the data on-the-fly, mark up the data in HTML, and send the resulting HTML files back to the web browsers to render. *Apache*, *Tomcat* and *IIS* are popular web server programs, and *IE* and *Firefox* are popular web browsers.

1.2 FUNDAMENTAL OF WEB

1.2.1 WEB ARCHITECTURE

A typical web application involves four tiers as depicted in the following web architecture figure: web browsers on the client side for rendering data presentation coded in HTML, a web server program that generates data presentation, an application server program that computes business logic, and a database server program that provides data persistency. The three types of server programs may run on the same or different server machines.



Web browsers can run on most operating systems with limited hardware or software requirement. They are the graphic user interface for the clients to interact with web applications. The basic functions of a web browser include:

- Interpret HTML markup and present documents visually;

- Support hyperlinks in HTML documents so the clicking on such a hyperlink can lead to the corresponding HTML file being downloaded from the same or another web server and presented;
- Use HTML form and the HTTP protocol to send requests and data to web applications and download HTML documents;
- Maintain cookies (name value pairs, explained later) deposited on client computers by a web application and send all cookies back to a web site if they are deposited by the web application at that web site (cookies will be further discussed later in this chapter);
- Use plug-in applications to support extra functions like playing audio-video files and running Java applets;
- Implement a *web browser sandbox* security policy: any software component (applets, JavaScript, ActiveX ...) running inside a web browser normally cannot access local clients' resources like files or keyboards, and can only communicate directly with applications on the web server from where it is downloaded.

The web server is mainly for receiving document requests and data submission from web browsers through the HTTP protocol on top of the Internet's TCP/IP layer. The main function of the web server is to feed HTML files to the web browsers. If the client is requesting a static existing file, it will be retrieved on a server hard disk and sent back to the web browser right away. If the client needs customized HTML pages like the client's bank statement, a software component, like a JSP page or a servlet class (the

"Extension" box in the web architecture figure), needs to retrieve the client's data from the database and compose a response HTML file on-the-fly.

The application server is responsible for computing the business logics of the web application, like carrying out a bank account fund transfer and computing the shortest route to drive from one city to another. If the business logic is simple or the web application is only used by a small group of clients, the application server is usually missing and business logics are computed in the web server extensions (PHP, JSP or servlet ...). But for a popular web application that generates significant computation load for serving each client, the application server will take advantage of a separate hardware server machine to run business logics more efficiently. This is a good application of the divide-and-conquer problem solving methodology.

1.2.2 UNIFORM RESOURCE LOCATORS (URL)

A web server program runs multiple web applications (sites) hosted in different folders under the web server program's document root folder. A server computer may run multiple server programs including web servers. Each server program on a server computer uses a port number, between 0 and 65535, unique on the server machine as its local identification (by default a web server uses port 80). Each server computer has an IP address, like 198.105.44.27, as its unique identifier on the Internet. Domain names, like www.pace.edu, are used as user-friendly identifications of server computers, and they are mapped to

IP addresses by a Domain Name Server (DNS). A Uniform Resource Locator (URL) is an address for uniquely identifying a web resource (like a web page or a Java object) on the Internet, and it has the following general format:

`http://domain-name:port/application/resource?query-string`

where *http* is the protocol for accessing the resource (*https* and *ftp* are popular alternative protocols standing for *secure HTTP* and *File Transfer Protocol*); *application* is a server-side folder containing all resources related to a web application; *resource* could be the name (alias or nickname) of an HTML or script/program file residing on a server hard disk; and the optional query string passes user data to the web server. An example URL is <http://www.amazon.com/computer/sale?model=dell610>.

There is a special domain name “localhost” that is normally defined as an alias of local IP address 127.0.0.1. Domain name “localhost” and IP address 127.0.0.1 are for addressing a local computer, very useful for testing web applications where the web browser and the web server are running on the same computer.

Most computers are on the Internet as well as on a local area network (LAN), like home wireless network, and they have an external IP address and a local IP address. To find out what is your computer's external IP address on the Internet, use a web browser to visit <http://whatismyip.com>. To find out what is your local (home) IP address, on Windows, run “ipconfig” in a DOS window; and on Linux, run “sudo ifconfig” in a terminal window.

1.2.3 HTML BASICS

HTML is a markup language. An HTML document is basically a text document marked up with instructions as to document logical structure and document presentation. The following is the contents of file “~/tomcat/webapps/demo/echoPost.html” in the ubuntu10 VM.

```

<html>
<head>
<body>
  <form method="post" action="http://localhost:8080/demo/echo">
    Enter your name: <input type="text" name="user"/> <br/><br/>
    <input type="submit" value="Submit"/>
    <input type="reset" value="Reset"/>
  </form>
</body>
</head>
</html>

```

An HTML *tag name* is a predefined keyword, like html, body, head, title, p, and b, all in lower-case.

A tag name is used in the form of a *start tag* or an *end tag*. A start tag is a tag name enclosed in angle brackets < and >, like <html> and <p>. An end tag is the same as the corresponding start tag except it has a forward slash / immediately before the tag name, like </html> and </p>.

An *element* consists of a start tag and a matching end tag based on the same tag name, with optional text or other elements, called *element value*, in between them. The following are some element examples:

```

<p>This is free text</p>
<p>This element has a nested <b>element</b></p>

```

While the elements can be nested, they cannot be partially nested: the end tag of an element must come after the end tags of all of its nested elements (*first starting last ending*). The following example is not a valid element because it violates the above rule:

```

<p>This is not a valid <bold>element<p><bold>

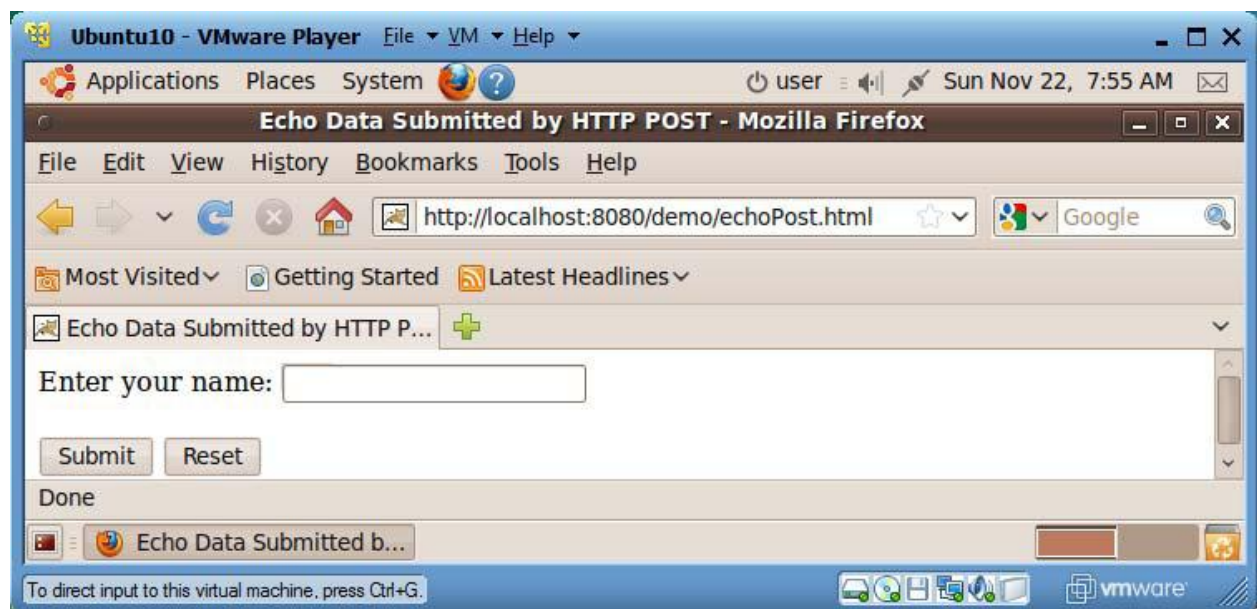
```

The *newline* character, the *tab* character and the *space* character are collectively called the *white-space characters*. A sequence of white-space characters acts like a single space for web browser's data presentation. Therefore, in normal situations, HTML document's formatting is not important (it will not change its presentation in web browsers) as long as you don't remove all white-space characters between successive words.

If an element contains no value, the start tag and the end tag can be combined into a single one as <tagName/>. As an example, we use
 to insert a line break in HTML documents.

The start tag of an element may contain one or more *attributes*, each in the form “attributeName=“attributeValue””. The above form element has two attributes: method and action.

An HTML document must contain exactly one top-level html element, which in turn contains exactly one body element. Most of the other contents are nested in the body element. If you load the above file “echoPost.html” in a web browser you will see the following:

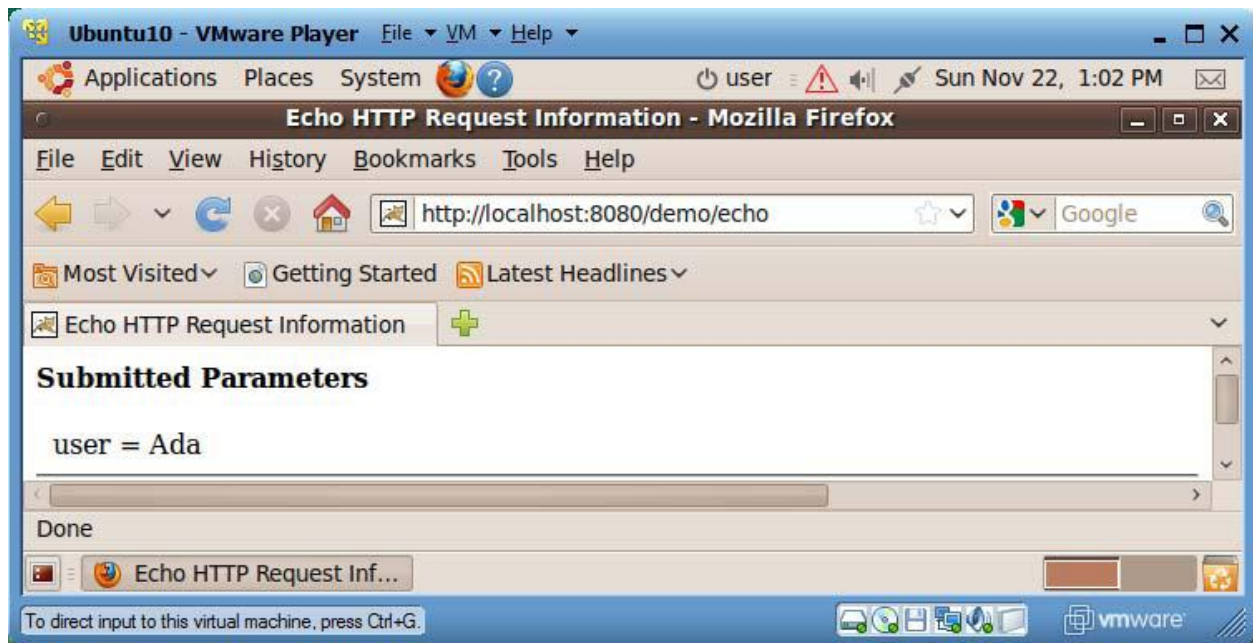


The form element is the most important mechanism for interaction between people and web applications.

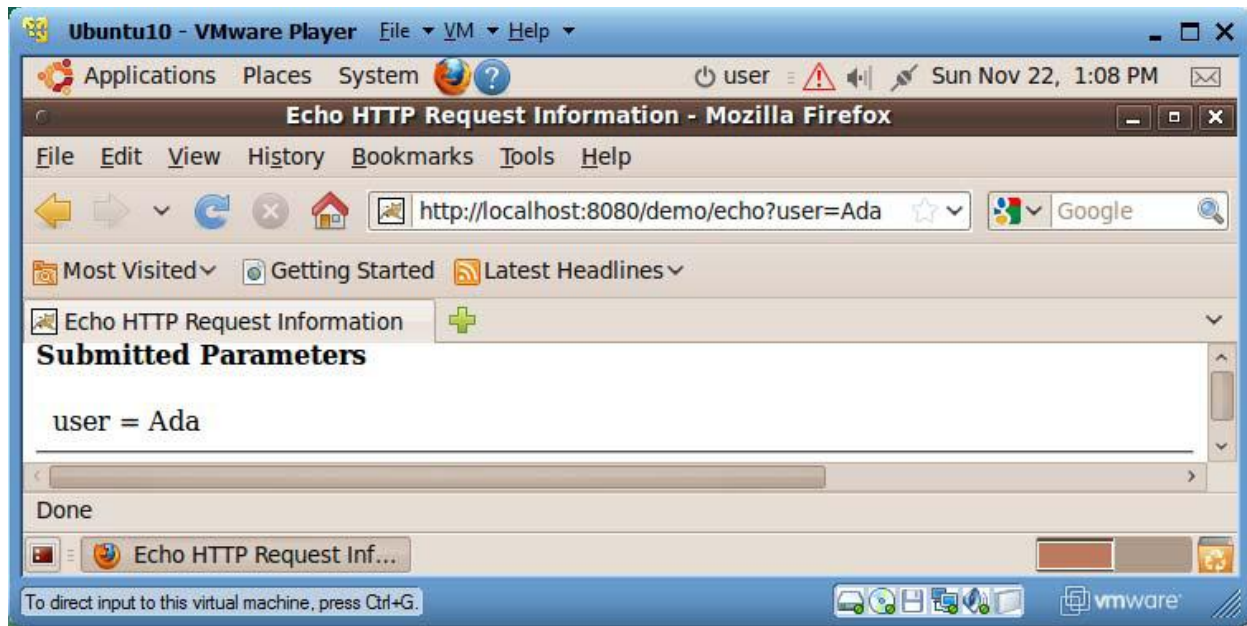
A form typically contains a few input elements and at least one submit button. A form element usually has two attributes: the method attribute for specifying HTTP method for submitting the form data to the web application (only values normally used are “get” and “post”); and the action attribute for specifying the form data submission destination, or the URL of a web application. In this example, when people click on the submit button, the form data will be sent to resource “echo” of the same web application “demo” deployed on your Ubuntu VM’s Tomcat web server, which will echo back all information the web browser sent to the web server. If the action value doesn’t specify the domain name/IP address or the web application, then the

web application from where this HTML file came from will receive the form data. The first input element of type “text” has been rendered as a text field, the second input element of type “submit” has been rendered as a submit button, and the third input element of type “reset” has been rendered as a reset button. The value attribute of the input elements determines what string will be displayed on the element’s image. The name attribute of the input element specifies the variable name with which web server programs can access what people type/enter in the element. When the submit button is clicked, the form data will be packaged as an HTTP request and sent to the web resource specified by the action attribute with the method specified by the method attribute.

If you type “Ada” in the name field and click on the submit button, you will receive the HTTP response partially displayed below.



If you load file “echoGet.html” from the same web application folder “demo”, the HTML file contents is basically the same except the method attribute for the form is changed from “post” to “get”. If you enter “Ada” in the name field and click on the submit button again, you will notice that the query string “?user=Ada” has been appended to the end of the URL. This is a major difference from HTTP POST method, and you will learn more about HTTP GET/POST soon.



An HTML file can contain hyperlinks to other web pages so users can click on them to visit different web pages. A hyperlink has the general structure of `Hyperlink Text`. The following is an example hyperlink. Since its href value is not a web page, the *welcome page* of the Google web site, which is the default page sent back if a browser visits the web site without specifying a specific interested page, will be sent back to the web browser.

```
<a href="http://www.google.com">Google</a>
```

When you click on a hyperlink, an HTTP GET request will be sent to the web server with all values to be submitted in the form of query strings.

1.2.4 HTTP PROTOCOL

Web browsers interact with web servers with a simple application-level protocol called HTTP (HyperText Transfer Protocol), which runs on top of TCP/IP network connections. When people click on the submit button of an HTML form or a hyperlink in a web browser, a TCP/IP virtual communication channel is created from the browser to the web server specified in the URL; an HTTP GET or POST request is sent through this channel to the destination web application, which retrieves data submitted by the browser user and composes an HTML file; the HTML file is sent back to the web browser as an HTTP response through the same TCP/IP channel; and then the TCP/IP channel is shut down.

The following is the HTTP POST request sent when you type “Ada” in the text field and click on the submit button of the previous file “echoPost.html”.

```
POST /demo/echo HTTP/1.1
Accept: text/html
Accept: audio/x
User-agent: Mozilla/5.0
Referer: http://localhost:8080/demo/echoPost.html
Content-length: 8 user=Ada
```

The first line, the request line, of a HTTP request is used to specify the submission type, GET or POST; the specific web resource on the web server for receiving and processing the submitted data; and the latest HTTP version that the web browser supports. As of 2010, version 1.1 is the latest HTTP specification.

The following lines, up to before the blank line, are HTTP *header lines* for declaring web browser capabilities and extra information for this submission, each of form “name: value”. The first two Accept headers declare that the web browser can process HTML files and any standard audio file formats from the web server. The User-agent header declares the software architecture of the web browser. The

Referer (yes this misspelled word is used by the HTTP standard) header specifies the URL of a web page from which this HTTP request is generated (this is how online companies like Amazon and Yahoo collect money for advertisements on their web pages from their sponsors). Any text after the blank line below the header lines is called the *entity body* of the HTTP request, which contains user data submitted through HTTP POST. The Content-length header specifies the exact number of bytes that the entity body contains. If the data is submitted through HTTP GET, the entity body will be empty and the data go to the query string of the submitting URL, as you saw earlier.

In response to this HTTP POST request, the web server will forward the submitted data to resource echo of web application demo, and the resource echo (a Java servlet) will generate dynamically an HTML page for most data it can get from the submission and let the web server send the HTML page back to the web browser as the entity body of the following HTTP response.

```
HTTP/1.1 200 OK
Server: NCSA/1.3
Mime_version: 1.0
```

Content_type: text/html

Content_length: 2000

<HTML>

.....

</HTML>

The first line, the response line, of an HTTP response specifies the latest HTTP version that the web server supports. The first line also provides a web server processing status code, the popular values of which include 200 for OK, 400 if the server doesn't understand the request, 404 if the server cannot find the requested page, and 500 for server internal error. The third entry on the first line is a brief message explaining the status code. The first two header lines declare the web server capabilities and meta-data for the returned data. In this example, the web server is based on a software architecture named "NCSA/1.3", and it supports *Multipurpose Internet Mail Extension* (MIME) specification v1.0 for web browsers to submit text or binary data with multi-parts. The last two header lines declare that the entity body contains HTML data with exactly 2000 bytes. The web browser will parse this HTTP response and present the response data.

The HTTP protocol doesn't have memory: the successive HTTP requests don't share data.

HTTP GET was initially designed for downloading static web pages from web servers, and it mainly used short query strings to specify the web page search criteria. HTTP POST was initially designed for submitting data to web servers, so it used the request entity body to send data to the web servers as a data stream, and its response normally depended on the submitted data and the submission status. While both HTTP GET and HTTP POST can send user requests to web servers and retrieve HTML pages from web servers for a web browser to present, they have the following subtle but important differences:

- HTTP GET sends data as query strings so people can read the submitted data over submitter's shoulders.
- Web servers have limited buffer size, typically 512 bytes, for accommodating query string data. If a user submits more data than that limit, either the data would be truncated, or the web server would crash, or the submitted data could potentially overwrite some computer code on the server and the server was led to run some hideous code hidden as part of the query string data. The last case is the so-called *buffer overflow*, a common way for hackers to take over the control of a server and spread virus or worms.

- By default web browsers keep (cache) a copy of the web page returned by an HTTP GET request so the future requests to the same URL can be avoided and the cached copy could be easily reused. While this can definitely improve the performance if the requested web page doesn't change, it could be disastrous if the web page or data change with time.

1.2.5 SESSION DATA MANAGEMENT

Most web applications need a user to interact with it multiple times to complete a business transaction.

For example, when you shop at Amazon, you choose one book at a time by clicking on some HTML form's submission buttons/hyperlinks in a web browser, and Amazon will process your submitted data and send you another HTML form for further shopping. A sequence of related HTTP requests between a web browser and a web application for accomplishing a single business transaction is called a session. All data specified by the user is called the session data. Session data are private so they must be protected from other users. A session normally starts when you first visit a web site in a particular day, and terminates when you pay off your purchase or shut down your web browser. Since the HTTP protocol has no memory, web applications have to use some special mechanisms to securely maintain the user session data.

Cookies

A cookie is a pair of name and value, as in (name, value). A web application can generate multiple cookies, set their life spans in terms of how many milliseconds each of them should be alive, and send them back to a web browser as part of an HTTP response. If cookies are allowed, a web browser will save all cookies on its hosting computer, along with their originating URLs and life spans. When an HTTP request is sent from a web browser of the same type on the same computer to a web site, all live cookies originated from that web site will be sent to the web site as part of the HTTP request. Therefore session data can be stored in cookies. This is the simplest approach to maintain session data. Since the web server doesn't need to commit any resources for the session data, this is the most scalable approach to support session data of large number of users. But it is not secure or efficient for cookies to go between a web browser and a web site for every HTTP request, and hackers could eavesdrop for the session data along the Internet path.

Hidden Fields

Some web users have great concern of the cookie's security implications and they disable cookie support on their web browsers. A web application can check the header fields of HTTP

requests to detect whether cookies are supported by the requesting web browser. If the cookies are disabled, the web application will normally use form hidden fields to store session data. Upon receiving submitted data through an HTTP request, the web application will generate a new HTML form for the user to continue the business transaction, and it will populate all useful session data in the new HTML form as hidden fields (input elements of type “hidden”). When the user submits the form again, all the data that the user just entered the form, as well as all data saved in the form as hidden fields, will be sent back to the web application again. Therefore this hidden fields approach for maintaining session data shares most of the advantages and disadvantages of the cookie approach.

Query Strings

Sometimes query strings can also be used to maintain small amount of session data. This is particular true for maintaining the short session IDs that will be introduced below. But since most business transactions are implemented with HTML forms, this approach is less useful.

Server-Side Session Objects

For improving the security of session data and avoiding the wasted network bandwidth for session data to move back and forth between a web browser and a web server, you can also save much of the session data on the web server as server-side *session objects*. A session object has a unique session ID for identifying a specific user. A session object is normally implemented as a hash table (lookup table) consisting of (name, value) pairs. A single cookie, hidden field of a form, or query string of a hyperlink can be used to maintain the session ID. Since session ID is a fixed size small piece of data, it will not cause much network overhead for going between a web browser and a web server for each HTTP request. For securing the session data, you need to make sure that the session ID is unique and properly protected on the client site. Since this approach stores all session data on the web server, it takes the most server resources and is relatively harder to serve large number of clients concurrently.

1.3 INTERNET

The **Internet** is a global system of interconnected computer networks that use the standard Internet protocol suite (TCP/IP) to serve several billion users worldwide. It is a *network of networks* that consists of millions of private, public, academic, business, and government networks, of local to global scope, that are linked by a broad array of electronic, wireless, and

optical networking technologies. The Internet carries an extensive range of information resources and services, such as the inter-linked hypertext documents of the World Wide Web (WWW), the infrastructure to support email, and peer-to-peer networks.

Most traditional communications media including telephone, music, film, and television are being reshaped or redefined by the Internet, giving birth to new services such as voice over Internet Protocol (VoIP) and Internet Protocol television (IPTV). Newspaper, book, and other print publishing are adapting to website technology, or are reshaped into blogging and web feeds. The Internet has enabled and accelerated new forms of human interactions through instant messaging, Internet forums, and social networking. Online shopping has boomed both for major retail outlets and small artisans and traders. Business-to-business and financial services on the Internet affect supply chains across entire industries.

1.3.1 TECHNOLOGY

Protocols

The communications infrastructure of the Internet consists of its hardware components and a system of software layers that control various aspects of the architecture. While the hardware can often be used to support other software systems, it is the design and the rigorous standardization process of the software architecture that characterizes the Internet and provides the foundation for its scalability and success. The responsibility for the architectural design of the Internet software systems has been delegated to the Internet Engineering Task Force (IETF). The IETF conducts standard-setting work groups, open to any individual, about the various aspects of Internet architecture. Resulting discussions and final standards are published in a series of publications; each called a Request for Comments (RFC), freely available on the IETF web site.

The principal methods of networking that enable the Internet are contained in specially designated RFCs that constitute the Internet Standards. Other less rigorous documents are simply informative, experimental, or historical, or document the best current practices (BCP) when implementing Internet technologies.

The Internet standards describe a framework known as the Internet protocol suite. This is a model architecture that divides methods into a layered system of protocols. The layers correspond to the environment or scope in which their services operate. At the top is

the application layer, the space for the application-specific networking methods used in software applications, e.g., a web browser program uses the client-server application model and many file-sharing systems use a peer-to-peer paradigm. Below this top layer, the transport layer connects applications on *different hosts* via the network with appropriate data exchange methods. Underlying these layers are the core networking technologies, consisting of two layers.

The internet layer enables computers to identify and locate each other via Internet Protocol (IP) addresses, and allows them to connect to one another via intermediate (transit) networks. Last, at the bottom of the architecture, is a software layer, the link layer, that provides connectivity between hosts on the same local network link, such as a local area network (LAN) or a dial-up connection. The model, also known as TCP/IP, is designed to be independent of the underlying hardware, which the model therefore does not concern itself with in any detail. Other models have been developed, such as the Open Systems Interconnection (OSI) model, but they are not compatible in the details of description or implementation; many similarities exist and the TCP/IP protocols are usually included in the discussion of OSI networking.

The most prominent component of the Internet model is the Internet Protocol (IP), which provides addressing systems (IP addresses) for computers on the Internet. IP enables internetworking and in essence establishes the Internet itself. IP Version 4 (IPv4) is the initial version used on the first generation of today's Internet and is still in dominant use. It was designed to address up to ~ 4.3 billion (10^9) Internet hosts. However, the explosive growth of the Internet has led to IPv4 address exhaustion, which entered its final stage in 2011, when the global address allocation pool was exhausted. A new protocol version, IPv6, was developed in the mid-1990s, which provides vastly larger addressing capabilities and more efficient routing of Internet traffic. IPv6 is currently in growing deployment around the world, since Internet address registries (RIRs) began to urge all resource managers to plan rapid adoption and conversion.

IPv6 is not interoperable with IPv4. In essence, it establishes a parallel version of the Internet not directly accessible with IPv4 software. This means software upgrades or translator facilities are necessary for networking devices that need to communicate on both networks. Most modern computer operating systems already support both versions of the Internet Protocol. Network infrastructures, however, are still lagging in this development. Aside from the complex array of physical connections that make up its infrastructure, the Internet is facilitated by bi- or

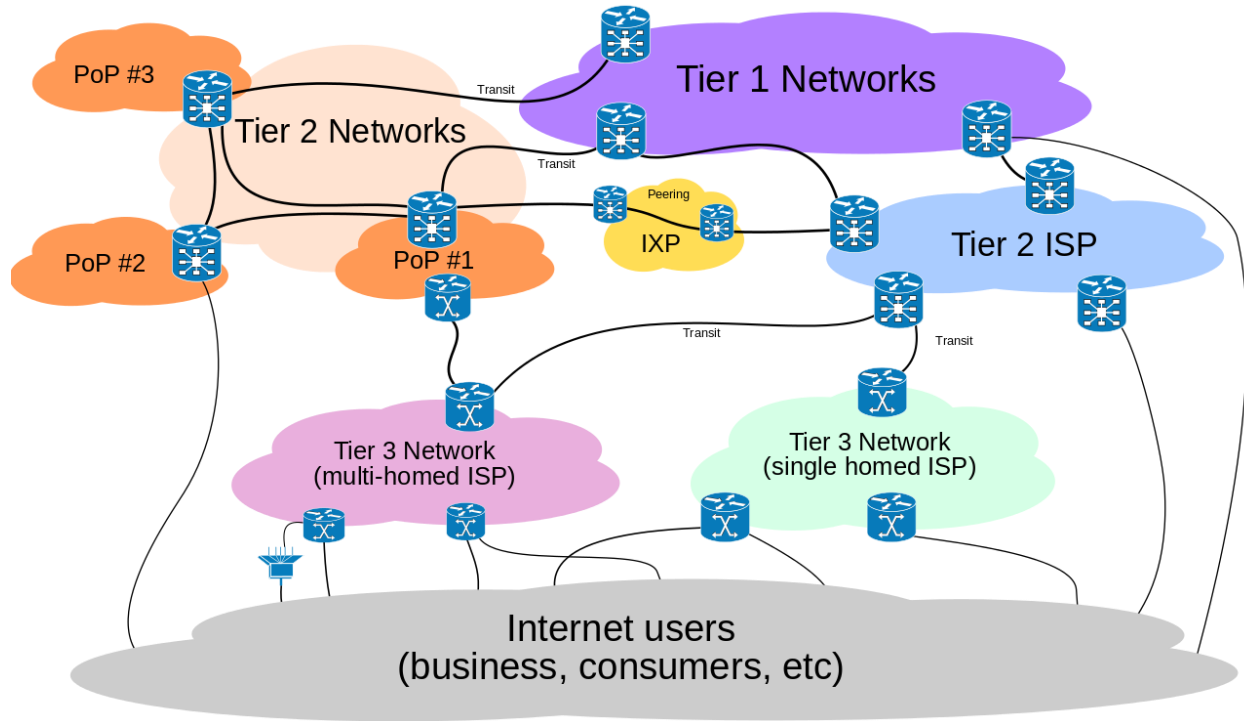
multi-lateral commercial contracts (e.g., peering agreements), and by technical specifications or protocols that describe how to exchange data over the network. Indeed, the Internet is defined by its interconnections and routing policies.

Routing

Internet service providers connect customers, which represent the bottom of the routing hierarchy, to customers of other ISPs via other higher or same-tier networks. At the top of the routing hierarchy are the Tier 1 networks, large telecommunication companies which exchange traffic directly with all other Tier 1 networks via peering agreements. Tier 2 networks buy Internet transit from other providers to reach at least some parties on the global Internet, though they may also engage in peering. An ISP may use a single upstream provider for connectivity, or implement multihoming to achieve redundancy. Internet exchange points are major traffic exchanges with physical connections to multiple ISPs.

Computers and routers use routing tables to direct IP packets to the next-hop router or destination. Routing tables are maintained by manual configuration or by routing protocols. End-nodes typically use a default route that points toward an ISP providing transit, while ISP routers use the Border Gateway Protocol to establish the most efficient routing across the complex connections of the global Internet.

Large organizations, such as academic institutions, large enterprises, and governments, may perform the same function as ISPs, engaging in peering and purchasing transit on behalf of their internal networks. Research networks tend to interconnect into large sub networks such as GEANT, GLORIAD, Internet2, and the UK's national research and education network, JANET.



1.3.2 USES OF INTERNET

The Internet allows greater flexibility in working hours and location, especially with the spread of unmetered high-speed connections. The Internet can be accessed almost anywhere by numerous means, including through mobile Internet devices. Mobile phones, datacards, handheld game consoles and cellular routers allow users to connect to the Internet wirelessly. Within the limitations imposed by small screens and other limited facilities of such pocket-sized devices, the services of the Internet, including email and the web, may be available. Service providers may restrict the services offered and mobile data charges may be significantly higher than other access methods.

Educational material at all levels from pre-school to post-doctoral is available from websites. Examples range from CBeebies, through school and high-school revision guides and virtual universities, to access to top-end scholarly literature through the likes of Google Scholar. For distance education, help with homework and other assignments, self-guided learning, whiling away spare time, or just looking up more detail on an interesting fact, it has never been easier for people to access educational information at any level from anywhere. The

Internet in general and the World in particular are important enablers of both formal and informal education.

The low cost and nearly instantaneous sharing of ideas, knowledge, and skills has made collaborative work dramatically easier, with the help of collaborative software. Not only can a group cheaply communicate and share ideas but the wide reach of the Internet allows such groups more easily to form. An example of this is the free software movement, which has produced, among other things, Linux, Mozilla Firefox, and OpenOffice.org. Internet chat, whether using an IRC chat room, an instant messaging system, or a social networking website, allows colleagues to stay in touch in a very convenient way while working at their computers during the day. Messages can be exchanged even more quickly and conveniently than via email. These systems may allow files to be exchanged, drawings and images to be shared, or voice and video contact between team members.

Content management systems allow collaborating teams to work on shared sets of documents simultaneously without accidentally destroying each other's work. Business and project teams can share calendars as well as documents and other information. Such collaboration occurs in a wide variety of areas including scientific research, software development, conference planning, political activism and creative writing. Social and political collaboration is also becoming more widespread as both Internet access and computer literacy spread.

The Internet allows computer users to remotely access other computers and information stores easily, wherever they may be. They may do this with or without computer security, i.e. authentication and encryption technologies, depending on the requirements. This is encouraging new ways of working from home, collaboration and information sharing in many industries. An accountant sitting at home can audit the books of a company based in another country, on a server situated in a third country that is remotely maintained by IT specialists in a fourth. These accounts could have been created by home-working bookkeepers, in other remote locations, based on information emailed to them from offices all over the world. Some of these things were possible before the widespread use of the Internet, but the cost of private leased lines would have made many of them infeasible in practice. An office worker away from their desk, perhaps on the other side of the world on a business trip or a holiday, can access their emails, access their data

using cloud computing, or open a remote desktop session into their office PC using a secure Virtual (VPN) connection on the Internet. This can give the worker complete access to all of their normal files and data, including email and other applications, while away from the office. It has been referred to among system administrators as the Virtual Private Nightmare, because it extends the secure perimeter of a corporate network into remote locations and its employees' homes.

Many people use the terms *Internet* and *World Wide Web*, or just the *Web*, interchangeably, but the two terms are not synonymous. The World Wide Web is only one of hundreds of services used on the Internet. The Web is a global set of documents, images and other resources, logically interrelated by hyperlinks and referenced with Uniform Resource Identifiers (URIs). URIs symbolically identifies services, servers, and other databases, and the documents and resources that they can provide. Hypertext Transfer Protocol (HTTP) is the main access protocol of the World Wide Web. services also use HTTP to allow software systems to communicate in order to share and exchange business logic and data.

World Wide Web browser software, such as Microsoft's Internet Explorer, Mozilla Firefox, Opera, Apple's Safari, and Google Chrome, lets users navigate from one web page to another via hyperlinks embedded in the documents. These documents may also contain any combination of computer data, including graphics, sounds, text, video, multimedia and interactive content that runs while the user is interacting with the page. Client-side software can include animations, games, office applications and scientific demonstrations. Through keyword-driven Internet research using search engines like Yahoo! and Google, users worldwide have easy, instant access to a vast and diverse amount of online information. Compared to printed media, books, encyclopedias and traditional libraries, the World Wide Web has enabled the decentralization of information on a large scale.

The Web has also enabled individuals and organizations to publish ideas and information to a potentially large audience online at greatly reduced expense and time delay. Publishing a web page, a blog, or building a website involves little initial cost and many cost-free services are available. Publishing and maintaining large, professional web sites with attractive, diverse and up-to-date information is still a difficult and expensive proposition, however. Many individuals and some companies and groups use *web logs* or blogs, which are largely used as easily

updatable online diaries. Some commercial organizations encourage staff to communicate advice in their areas of specialization in the hope that visitors will be impressed by the expert knowledge and free information, and be attracted to the corporation as a result.

One example of this practice is Microsoft, whose product developers publish their personal blogs in order to pique the public's interest in their work. Collections of personal web pages published by large service providers remain popular, and have become increasingly sophisticated. Whereas operations such as Angelfire and GeoCities have existed since the early days of the Web, newer offerings from, for example, Facebook and Twitter currently have large followings. These operations often brand themselves as social network services rather than simply as web page hosts.

Advertising on popular web pages can be lucrative, and e-commerce or the sale of products and services directly via the Web continues to grow.

Communication

Email is an important communications service available on the Internet. The concept of sending electronic text messages between parties in a way analogous to mailing letters or memos predates the creation of the Internet. Pictures, documents and other files are sent as email attachments. Emails can be cc-ed to multiple email addresses.

Internet telephony is another common communications service made possible by the creation of the Internet. VoIP stands for Voice-over-Internet Protocol, referring to the protocol that underlies all Internet communication. The idea began in the early 1990s with walkie-talkie-like voice applications for personal computers. In recent years many VoIP systems have become as easy to use and as convenient as a normal telephone. The benefit is that, as the Internet carries the voice traffic, VoIP can be free or cost much less than a traditional telephone call, especially over long distances and especially for those with always-on Internet connections such as cable or ADSL. VoIP is maturing into a competitive alternative to traditional telephone service. Interoperability between different providers has improved and the ability to call or receive a call from a traditional telephone is available. Simple, inexpensive VoIP network adapters are available that eliminate the need for a personal computer.

Voice quality can still vary from call to call, but is often equal to and can even exceed that of traditional calls. Remaining problems for VoIP include emergency telephone

number dialing and reliability. Currently, a few VoIP providers provide an emergency service, but it is not universally available. Older traditional phones with no "extra features" may be line-powered only and operate during a power failure; VoIP can never do so without a backup power source for the phone equipment and the Internet access devices. VoIP has also become increasingly popular for gaming applications, as a form of communication between players. Popular VoIP clients for gaming include Ventrilo and Teamspeak. Modern video game consoles also offer VoIP chat features.

Data transfer

File sharing is an example of transferring large amounts of data across the Internet. A computer file can be emailed to customers, colleagues and friends as an attachment. It can be uploaded to a website or FTP server for easy download by others. It can be put into a "shared location" or onto a file server for instant use by colleagues. The load of bulk downloads to many users can be eased by the use of "mirror" servers or peer-to-peer networks. In any of these cases, access to the file may be controlled by user authentication, the transit of the file over the Internet may be obscured by encryption, and money may change hands for access to the file. The price can be paid by the remote charging of funds from, for example, a credit card whose details are also passed – usually fully encrypted – across the Internet. The origin and authenticity of the file received may be checked by digital signatures or by MD5 or other message digests. These simple features of the Internet, over a worldwide basis, are changing the production, sale, and distribution of anything that can be reduced to a computer file for transmission. This includes all manner of print publications, software products, news, music, film, video, photography, graphics and the other arts. This in turn has caused seismic shifts in each of the existing industries that previously controlled the production and distribution of these products.

Streaming media is the real-time delivery of digital media for the immediate consumption or enjoyment by end users. Many radio and television broadcasters provide Internet feeds of their live audio and video productions. They may also allow time-shift viewing or listening such as Preview, Classic Clips and Listen Again features. These providers have been joined by a range of pure Internet "broadcasters" who never had on-air licenses. This means that an Internet-connected device, such as a computer or something more specific, can be used to access on-line media in much the same way as was previously possible only with a television or radio receiver. The range of available types of content is much wider, from specialized technical webcasts to on-

demand popular multimedia services. Podcasting is a variation on this theme, where – usually audio – material is downloaded and played back on a computer or shifted to a portable media player to be listened to on the move. These techniques using simple equipment allow anybody, with little censorship or licensing control, to broadcast audio-visual material worldwide.

Digital media streaming increases the demand for network bandwidth. For example, standard image quality needs 1 Mbit/s link speed for SD 480p, HD 720p quality requires 2.5 Mbit/s, and the top-of-the-line HDX quality needs 4.5 Mbit/s for 1080p.

Webcams are a low-cost extension of this phenomenon. While some webcams can give full-frame-rate video, the picture either is usually small or updates slowly. Internet users can watch animals around an African waterhole, ships in the Panama Canal, traffic at a local roundabout or monitor their own premises, live and in real time. Video chat rooms and video conferencing are also popular with many uses being found for personal webcams, with and without two-way sound. YouTube was founded on 15 February 2005 and is now the leading website for free streaming video with a vast number of users. It uses a flash-based web player to stream and show video files. Registered users may upload an unlimited amount of video and build their own personal profile. YouTube claims that its users watch hundreds of millions, and upload hundreds of thousands of videos daily.

1.4 WORLD WIDE WEB

The **World Wide Web** (abbreviated as **WWW** or **W3**, commonly known as **the web**) is a system of interlinked hypertext documents 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.

1.4.1 FUNCTIONS

The terms Internet and World Wide Web are often used in everyday speech without much distinction. However, the Internet and the World Wide Web are not the same. The Internet is a global system of interconnected computer networks. In contrast, the web is one of the services that run on the Internet. It is a collection of text documents and other resources, linked by hyperlinks and URLs, usually accessed by web browsers from web servers. In short, the web can be thought of as an application "running" on the Internet.

Viewing a web page on the World Wide Web normally begins either by typing the URL of the page into a web browser or by following a hyperlink to that page or resource. The web browser then initiates a series of communication messages, behind the scenes, in order to fetch and display it. The following example demonstrates how a web browser works. Consider accessing a page with the URL `http://example.org/wiki/World_Wide_Web`.

First, the browser resolves the server-name portion of the URL (*example.org*) into an Internet Protocol address using the globally distributed database known as the Domain Name System (DNS); this lookup returns an IP address such as *208.80.152.2*. The browser then requests the resource by sending an HTTP request across the Internet to the computer at that particular address. It makes the request to a particular application port in the underlying Internet Protocol Suite so that the computer receiving the request can distinguish an HTTP request from other network protocols it may be servicing such as e-mail delivery; the HTTP protocol normally uses port 80. The content of the HTTP request can be as simple as the two lines of text `GET /wiki/World_Wide_Web HTTP/1.1 Host: example.org`

The computer receiving the HTTP request delivers it to web server software listening for requests on port 80. If the web server can fulfill the request it sends an HTTP response back to the browser indicating success, which can be as simple as `HTTP/1.0 200 OK Content-Type: text/html; charset=UTF-8` followed by the content of the requested page. The Hypertext Markup Language for a basic web page looks like `<html> <head> <title>Example.org – The World Wide Web</title> </head> <body> <p>The World Wide Web, abbreviated as WWW and commonly known ...</p> </body> </html>`

The web browser parses the HTML, interpreting the markup (`<title>`, `<p>` for paragraph, and such) that surrounds the words in order to draw the text on the screen.

Many web pages use HTML to reference the URLs of other resources such as images, other embedded media, scripts that affect page behavior, and Cascading Style Sheets that affect page layout. The browser will make additional HTTP requests to the web server for these other Internet media types. As it receives their content from the web server, the browser progressively renders the page onto the screen as specified by its HTML and these additional resources.

LINKING

Most web pages contain hyperlinks to other related pages and perhaps to downloadable files, source documents, definitions and other web resources. In the underlying HTML, a hyperlink looks like `Example.org, a free encyclopedia`

Such a collection of useful, related resources, interconnected via hypertext links is dubbed a web of information. Publication on the Internet created what Tim Berners-Lee first called the WorldWideWeb (in its original CamelCase, which was subsequently discarded) in November 1990.

The hyperlink structure of the WWW is described by the webgraph: the nodes of the webgraph correspond to the web pages (or URLs) the directed edges between them to the hyperlinks.

Over time, many web resources pointed to by hyperlinks disappear, relocate, or are replaced with different content. This makes hyperlinks obsolete, a phenomenon referred to in some circles as link rot and the hyperlinks affected by it are often called dead links. The ephemeral nature of the Web has prompted many efforts to archive web sites. The Internet Archive, active since 1996, is the best known of such efforts.

1.4.2 DYNAMIC UPDATES OF WEB PAGES

To make web pages more interactive, some web applications also use JavaScript techniques such as Ajax(asynchronous JavaScript and XML). Client-side script is delivered with the page that can make additional HTTP requests to the server, either in response to user actions such as mouse movements or clicks, or based on lapsed time. The server's responses are used to modify the current page rather than creating a new page with each response, so the server needs only to provide limited, incremental information. Multiple Ajax requests can be handled at the same time, and users can interact with the page while data is being retrieved. Web pages may also regularly poll the server to check whether new information is available.

1.4.3 WWW PREFIX

Many hostnames used for the World Wide Web begin with www because of the long-standing practice of naming Internet hosts (servers) according to the services they provide. The hostname for a web server is often www, in the same way that it may be ftp for an FTP server, and news or nntp for a USENET news server. These host names appear as Domain Name

System or (DNS) subdomain names, as in `www.example.com`. The use of 'www' as a subdomain name is not required by any technical or policy standard and many web sites do not use it; indeed, the first ever web server was called `nxoc01.cern.ch`.

The use of a sub domain name is useful for load balancing incoming web traffic by creating a CNAME record that points to a cluster of web servers. Since, currently, only a sub domain can be used in a CNAME, the same result cannot be achieved by using the bare domain root.

When a user submits an incomplete domain name to a web browser in its address bar input field, some web browsers automatically try adding the prefix "www" to the beginning of it and possibly ".com", ".org" and ".net" at the end, depending on what might be missing. For example, entering 'microsoft' may be transformed to `http://www.microsoft.com/` and 'openoffice' to `http://www.openoffice.org`. This feature started appearing in early versions of Mozilla Firefox, when it still had the working title 'Firebird' in early 2003, from an earlier practice in browsers such as Lynx. It is reported that Microsoft was granted a US patent for the same idea in 2008, but only for mobile devices.

Use of the www prefix is declining as Web 2.0 web applications seek to brand their domain names and make them easily pronounceable. As the mobile web grows in popularity, services like Gmail.com, MySpace.com, Facebook.com and Twitter.com are most often discussed without adding www to the domain (or, indeed, the .com).

1.4.4 SCHEME SPECIFIERS: HTTP AND HTTPS

The scheme specifier `http://` or `https://` at the start of a web URI refers to Hypertext Transfer Protocol or HTTP Secure respectively. Unlike www, which has no specific purpose, these specify the communication protocol to be used for the request and response. The HTTP protocol is fundamental to the operation of the World Wide Web and the added encryption layer in HTTPS is essential when confidential information such as passwords or banking information are to be exchanged over the public Internet. Web browsers usually prepend `http://` to addresses too, if omitted.

1.4.5 SECURITY

The web has become criminals' preferred pathway for spreading malware. Cybercrime carried out on the web can include identity theft, fraud, espionage and intelligence gathering. Web-based vulnerabilities now outnumber traditional computer security concerns, and

as measured by Google, about one in ten web pages may contain malicious code. Most web-based attacks take place on legitimate websites, and most, as measured by Sophos, are hosted in the United States, China and Russia. The most common of all malware threats is SQL injection attacks against websites. Through HTML and URIs the web was vulnerable to attacks like cross-site scripting (XSS) that came with the introduction of JavaScript and were exacerbated to some degree by Web 2.0 and Ajax web design that favors the use of scripts. Today by one estimate, 70% of all websites are open to XSS attacks on their users.

1.5 WEB BROWSERS

A **web browser** (commonly referred to as a **browser**) is a software application for retrieving, presenting and traversing information resources on the World Wide Web. An *information resource* is identified by a Uniform Resource Identifier (URI/URL) and may be a web page, image, video or other piece of content. Hyperlinks present in resources enable users easily to navigate their browsers to related resources.

Although browsers are primarily intended to use the World Wide Web, they can also be used to access information provided by web servers in private networks or files in file systems.

The major web browsers are Google Chrome, Mozilla Firefox, Internet Explorer, Opera, and Safari.

1.5.1 BUSINESS MODELS

The ways that web browser makers fund their development costs has changed over time. The first web browser, WorldWideWeb, was a research project. Netscape Navigator was originally sold commercially, as was Opera; Netscape no longer exists and has been replaced with the free Firefox, while Opera is now downloadable free of charge.

Internet Explorer, on the other hand, was from its first release always included with the Windows operating system (and furthermore was downloadable for no extra charge), and therefore it was funded partly by the sales of Windows to computer manufacturers and direct to users. Internet Explorer also used to be available for the Mac. It is likely that releasing IE for the Mac was part of Microsoft's overall strategy to fight threats to its quasi-monopoly platform dominance - threats such as web standards and Java - by making some web developers, or at least their managers, assume that there was "no need" to develop for anything other than Internet Explorer (an assumption that later proved to be badly mistaken, with the rise of Firefox and

Chrome). In this respect, IE may have contributed to Windows and Microsoft applications sales in another way, through tricking organizations into inadvertent "lock-in" into the Microsoft platform.

Safari and Mobile Safari were likewise always included with OS X and iOS respectively, so, similarly, they were originally funded by sales of Apple computers and mobile devices, and formed part of the overall Apple experience to customers.

Today, most commercial web browsers are paid by search engine companies to make the search engine their default search engine (the most valuable prize) or to include them as another option. For example, Google pays Mozilla, the maker of Firefox, to make Google Search the default search engine in Firefox. Mozilla makes so much money from this deal that it does not need to charge users for Firefox. The reason search engine companies are willing to pay for such deals is that such decisions drive traffic their way, increasing ad revenue.

1.5.2 FUNCTION

The primary purpose of a web browser is to bring information resources to the user ("retrieval" or "fetching"), allowing them to view the information ("display", "rendering"), and then access other information ("navigation", "following links").

This process begins when the user inputs a Uniform Resource Locator (URL), for example <http://en.wikipedia.org/>, into the browser. The prefix of the URL, the Uniform Resource Identifier or URI, determines how the URL will be interpreted. The most commonly used kind of URI starts with [http:](http://) and identifies a resource to be retrieved over the Hypertext Transfer Protocol (HTTP). Many browsers also support a variety of other prefixes, such as [https:](https://) for HTTPS, [ftp:](ftp://) for the File Transfer Protocol, and [file:](file://) for local files. Prefixes that the web browser cannot directly handle are often handed off to another application entirely. For example, <mailto:> URIs is usually passed to the user's default e-mail application, and <news:> URIs is passed to the user's default newsgroup reader.

In the case of [http](http://), [https](https://), [file](file://), and others, once the resource has been retrieved the web browser will display it. HTML and associated content (image files, formatting information such as CSS, etc.) is passed to the browser's layout engine to be transformed from markup to an interactive document, a process known as "rendering". Aside from HTML, web browsers can generally display any kind of content that can be part of a web page. Most browsers can display images, audio, video, and XML files, and often have plug-ins to support Flash applications

and Java applets. Upon encountering a file of an unsupported type or a file that is set up to be downloaded rather than displayed, the browser prompts the user to save the file to disk.

Information resources may contain hyperlinks to other information resources. Each link contains the URI of a resource to go to. When a link is clicked, the browser navigates to the resource indicated by the link's target URI, and the process of bringing content to the user begins again.

1.5.3 FEATURES

Available web browsers range in features from minimal, text-based user interfaces with bare-bones support for HTML to rich user interfaces supporting a wide variety of file formats and protocols. Browsers which include additional components to support e-mail, Usenet news, and Internet Relay Chat (IRC), are sometimes referred to as "Internet suites" rather than merely "web browsers".

All major web browsers allow the user to open multiple information resources at the same time, either in different browser windows or in different tabs of the same window. Major browsers also include pop-up blockers to prevent unwanted windows from "popping up" without the user's consent.

Most web browsers can display a list of web pages that the user has bookmarked so that the user can quickly return to them. Bookmarks are also called "Favorites" in Internet Explorer. In addition, all major web browsers have some form of built-in web feed aggregator. In Firefox, web feeds are formatted as "live bookmarks" and behave like a folder of bookmarks corresponding to recent entries in the feed. In Opera, a more traditional feed reader is included which stores and displays the contents of the feed.

Furthermore, most browsers can be extended via plug-ins, downloadable components that provide additional features.

Most major web browsers have these user interface elements in common:

- Back and forward buttons to go back to the previous resource and forward respectively.
- A refresh or reload button to reload the current resource.
- A stop button to cancel loading the resource. In some browsers, the stop button is merged with the reload button.
- A home button to return to the user's home page.

- An address bar to input the Uniform Resource Identifier (URI) of the desired resource and display it.
- A search bar to input terms into a search engine. In some browsers, the search bar is merged with the address bar.
- A status bar to display progress in loading the resource and also the URI of links when the cursor hovers over them, and page zooming capability.
- The viewport, the visible area of the webpage within the browser window.

Major browsers also possess incremental find features to search within a web page.

Privacy and security: Most browsers support HTTP Secure and offer quick and easy ways to delete the web cache, cookies, and browsing history. For a comparison of the current security vulnerabilities of browsers, see comparison of web browsers.

Standards support: Early web browsers supported only a very simple version of HTML. The rapid development of proprietary web browsers led to the development of non-standard dialects of HTML, leading to problems with interoperability. Modern web browsers support a combination of standards-based and de facto HTML and XHTML, which should be rendered in the same way by all browsers.

Extensibility: A browser extension is a computer program that extends the functionality of a web browser. Every major web browser supports the development of browser extensions.

1.5.4 COMPONENTS

Web browsers are built of User Interface, Layout Engine, Rendering Engine, JavaScript interpreter, UI backend, Networking component and Data persistence component. These components achieve different functionalities of a web browser and together provide all capabilities of a web browser.

1.6 WEB SERVER

The term web server can refer to either the hardware (the computer) or the software (the computer application) that helps to deliver web content that can be accessed through the Internet.

The most common use of web servers is to host websites, but there are other uses such as gaming, data storage or running enterprise applications.

The primary function of a web server is to deliver web pages to clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP). 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, but this is not necessarily the case and depends on how the web server is implemented.

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 servers are not always used for serving the World Wide Web. They can also be found embedded in devices such as printers, routers, webcams and serving only a local network. The web server may then be used as a part of a system for monitoring and/or administering the device in question. This usually means that no additional software has to be installed on the client computer, since only a web browser is required.

1.6.1 COMMON 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 on 32 bit OS
- 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

1.6.2 PATH TRANSLATION

Web servers are able to map the path component of a Uniform Resource Locator (**URL**) into:

- A local file system resource (for static requests)
- An internal or external program name (for dynamic requests)

For a *static request* the URL path specified by the client is relative to the web server's root directory.

Consider the following URL as it would be requested by a client:

`http://www.example.com/path/file.html`

The client's user agent will translate it into a connection to `www.example.com` with the following HTTP 1.1 request:

`GET /path/file.html HTTP/1.1`

`Host: www.example.com`

The web server on `www.example.com` will append the given path to the path of its root directory. On an Apache server, this is commonly `/home/www` (On Unix machines, usually `/var/www`). The result is the local file system resource:

`/home/www/path/file.html`

The web server then reads the file, if it exists and sends a response to the client's web browser. The response will describe the content of the file and contain the file itself or an error message will return saying that the file does not exist or is unavailable.

1.6.3 TYPES OF WEB SERVERS

Kernel-mode and user-mode web servers: A web server can be either implemented into the OS kernel, or in user space.

An in-kernel web server 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. Not only do these requests to the kernel take time, but they are not always satisfied because the system reserves resources for its own usage and has the responsibility to share hardware resources with all the other running applications. Executing in user mode can also mean useless buffer copies which are another handicap for user-mode web servers.

Single-Threaded Web Server: In this type of web server's only one process sequentially handles all client connections. This requires no synchronization and does not scale i.e., it accepts only one client at a time.



Uses a selector to check for ready file descriptors and allows the file which is ready. These use a finite state machine to determine how to move to the next processing stage. There is no concept of context switching, no synchronization, and single address space. Modern Operating System does not provide adequate support for asynchronous disk operations.

1.6.4 LOAD LIMITS

A web server has defined load limits, because it can handle only a limited number of concurrent client connections per IP address 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;
- 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.

Symptoms of overload: The symptoms of an overloaded web server are:

- Requests are served with (possibly long) delays (from 1 second to a few hundred seconds).
- The web server returns an HTTP error code, such as 500, 502, 503, 504, 408, or even 404, which is inappropriate for an overload condition.

- The web server refuses or resets (interrupts) TCP connections before it returns any content.
- In very rare cases, the web server returns only a part of the requested content. This behavior can be considered a bug, even if it usually arises as a symptom of overload.

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
- Using more efficient computer programs for web servers, etc.
- Using other workarounds, especially if dynamic content is involved

1.7 WEB BROWSERS

A *URL (uniform resource locator)* is a uniform way to refer to objects and services on the Internet. Even novice users should be familiar with typing a URL, such as *http://www.htmlref.com*, in a browser dialog box, to get to a Web site. However, URLs can be used for far more than just retrieving a Web page and can be used to invoke other Internet services, such as transferring files via FTP or sending e-mail. Despite its potentially confusing collection of slashes and colons, URL syntax is designed to provide a clear, simple notation that people can easily understand. The concepts in this section will help you to better understand the syntax of URLs, which is key to linking documents in and beyond a Web site.

1.7.1 Basic Concepts

To locate any arbitrary object on the Internet, you need to find out the following information:

1. First, you need to locate and access the machine on the network on which the object resides. Locating the site might be a matter of specifying its domain name or IP address, whereas accessing the machine might require a username and password.
2. After you access the machine, you need to determine the name of the desired file, where the file is located, the position in the file as specified by a fragment identifier, and what protocol will be used to retrieve the information or access the object.

In other words, a URL describes where something is and how it will be retrieved. The where is specified by the machine name, the directory name, the filename, and potentially more.

The how is specified by the protocol (for example, HTTP). Slashes and other characters are used to separate the parts of the address into machine-readable pieces. The basic structure of the URL is shown here:

protocol://site address/directory/filename#fragmentid

The next several sections look at the individual pieces of a URL in closer detail.

1.7.2 Server Address

A document exists on some serving computer somewhere on the global Internet or within a private intranet. The first step in finding a document is to identify its server. This may be performed by a site's IP address,

http://10.0.0.1

though it is more likely that an alphanumeric domain name is employed,

`http://www.htmlref.com`

The name may be fully qualified with a machine name, a domain, an organization type, and potentially, a country code. For example,

`http://www.htmlref.com`

would specify the name of a machine called “www” in the domain htmlref, which is in the top-level COM domain. By contrast,

`http://dev.htmlref.com`

would reference a machine known as “dev” in the same domain.

Very often for primary Web sites within a domain the machine name is omitted, so we simply write

`http://htmlref.com`

However, such configuration is up to the owner of the domain. This short-hand form should be employed as most sites are reachable without a www prefix.

Historically, top-level domains such as those found in Table-1 are used.

Domain	Intended Type
.com	Commercial entities
.net	Networks
.edu	Educational institutions
.org	Non-Profit organizations
.gov	Government entities
.mil	U.S. military

Table-1: Common Top-Level Domains

However, starting around 2001, the top-level domain space expanded quite a bit. A sample of the top-level domains that have been added beyond the commonly known ones is shown in Table 2. Potentially more domains may be found at the Internet Assigned Numbers Authority (IANA) Web site (iana.org).

There is a distinct possibility that arbitrary domains could be introduced. For example, .google might be top-level domain for all Google properties. Even without this happening, the

top-level domain space is clearly a mess, and with generic domains on the horizon, the situation seems unlikely to get much better soon.

Geographic domains are particularly common outside the United States; such a domain name typically contains more information than the organization type, with a fully qualified domain name (FQDN) including a country code as well. It generally is written as follows:

machine name.domain name.domain type.country code

Zone identifiers outside the U.S. use a two-character code to indicate the country hosting the server. These include .ca for Canada, .mx for Mexico, .jp for Japan, and so on. A few examples are shown here.

www.unam.edu.mx

www.mcgill.ca

www.bbc.co.uk

www.ox.ac.uk

www.sony.co.jp

A complete list of country codes can be found at the IANA site (iana.org).

Domain	Intended Type
.aero	Business entities similar to .com
.asia	Entities in the Asia Pacific region
.biz	Business entities (similar to .com)
.coop	Cooperatives
.info	Information-oriented sites
.jobs	Job hosting sites
.mobi	Mobile device sites
.travel	Travel and tourism-related sites

Table-2: Some Newer Top-Level Domain

Within each country, the local naming authorities might create domain types at their own discretion, but these domain types can't correspond to American extensions. For example, we see that www.sony.co.jp specifies a Web server for Sony in the co zone of Japan. In this case, .co, rather than .com, indicates a commercial venture. In the United Kingdom, the educational domain space has a different name, ac. Oxford University's Web server is www.ox.ac.uk, whereby .ac indicates academic, compared to the U.S. .edu extension for education.

The United States also uses the .us extension, although it has only recently caught on outside of local government and k-12 educational environments. For example, www.sdcoe.k12.ca.us is the current address of the County Office of Education in San Diego.

However, the school district opts to use a .net domain (sandi.net), and individual high schools have even registered .com names. As in many organizations that have a choice of a regional domain, the shorter top-level domain is preferred, and unfortunately, the .com space seems to be the most desirable whether it is appropriate or not.

1.7.3 Directory

Once you reach a server, you may access a particular directory. The Web site directory that contains all others is known as the root directory and is specified with a single forward slash. So a URL like

`http://htmlref.com/`

would select the root directory of the book site. Very often users and developers will leave off the final trailing slash when referencing a directory. It is syntactically correct for it to be included, and if you don't include it, your browsers or the receiving Web server will likely add it in.

Directories may contain other directories

`http://htmlref.com/ch1/`

to arbitrary depth

`http://htmlref.com/really/long/fake/directory/path/`

However, do not assume that the Web server's operating system dictates everything; for example, URLs do not use Windows-style backslashes.

1.7.4 Filename

After you specify the server and the directory path for a document, the next step toward locating it is to specify its filename. Commonly, when a simple directory-based URL is given like

`http://htmlref.com/ch1/`

a default file in that directory, often named `index.html`, will be returned by the Web server. However, this file could be referenced directly like so:

`http://htmlref.com/ch1/index.html`

File names are arbitrary,

`http://htmlref.com/ch1/reallylongfilename.html`

and may be case sensitive, depending on the host operating system. Thus

`http://htmlref.com/ch1/reallylongfilename.html`

and

`http://htmlref.com/ch1/REALLYLONGFILENAME.html`

may reference the same object or not, depending on the operating system. Filenames may include special characters like dashes and underscores,

`http://htmlref.com/ch1/really_long_file_name.html`

`http://htmlref.com/ch1/another-really-long-file-name.html`

However, depending on the special characters used, they may be encoded (see the upcoming section “Encoding” for more information). As an example, the filename “really long file name.html” with spaces should encode as

`http://htmlref.com/ch1/another%20really%20long%20file%20name.html`

A dot separates the filename and the extension, which is a code, generally composed of three or four letters that identifies the type of information contained in the file. For example, HTML source files generally have a .htm or .html extension, CSS files, a .css extension, JavaScript files, a .js extension, JPEG images have a .jpg extension, and so on.

`http://htmlref.com/ch1/site.css`

`http://htmlref.com/ch1/bigimage.jpg`

`http://htmlref.com/ch1/jquery.js`

A file’s extension is critically important for Web applications because it is the primary indication of the information type that a file contains. However, it is possible to remove file extensions from URLs, as it is really the underlying MIME header that tells a browser what it is getting, so it might be quite possible to serve URLs like

`http://htmlref.com/ch1/listexamples`

rather than

`http://htmlref.com/ch1/listexamples.php`

Removing extensions will aid in portability and hide implementation details from end users.

1.7.5 Fragment Identifier

Besides referencing a file, it may be desirable to send a user directly to a particular point within the file. Because you can set up named links under traditional HTML and name any tag using the **id** attribute from HTML 4 onward, you can provide links directly to different points

within a file. To jump to a particular named link, the URL must include a hash symbol (#) followed by the link name, which indicates that the value is a fragment identifier. For example, given `<p id="#middle">` found in the file `fragmentids.html` in the `ch1` directory of the book support site, we would use the URL

`http://htmlref.com/ch1/fragmentids.html#middle`

Protocol

Finally, we need to specify how to retrieve information from the specified location. This is indicated in the URL by the protocol value. A protocol is the structured discussion that computers follow to negotiate resource-specific services. For example, the protocol that makes the Web possible is the Hypertext Transfer Protocol (HTTP). When you click a hyperlink in a Web document, your browser uses the HTTP protocol to contact a Web server and retrieve the appropriate document.

Protocol	Description	Example
https	Secure Sockets Layer (SSL) protocol for encrypted HTTP traffic	<code>https://yourbank.com/</code>
File	Enables a hyperlink to access a file on the local file system	<code>file:///C:/inetpub/wwwroot/ch1/fakeexample.html</code>
ftp (File Transfer Protocol)	Enables a hyperlink to download files from remote systems	<code>ftp://ftp.apple.com/</code>
telnet	Enables a hyperlink to open a telnet session on a remote host	<code>telnet://someserver.fakeexample.com</code>

Table-3: Some commonly used URL protocol's

These are the common protocols, but a variety of new protocols and URL forms are being debated all the time. We'll present a discussion of emerging URL forms toward the end of this appendix.

1.7.6 Other Features of URLs

In addition to the protocol, server address, directory, and filename, URLs often include a username and password, a port number, and potentially more. Some URLs, such as `mailto`, might even contain a different form of information altogether, such as an e-mail address rather than a server or filename.

Username and Password

FTP and telnet are protocols for authenticated services. It is also possible to make HTTP an authenticated service if you password-protect a directory or file. Authenticated services can restrict access to authorized users, and the protocols can require a username and password as parameters. A username and password precede a server name; for example, ftp syntax looks like

`ftp://username:password@server-address`

The password could be optional or unspecified in the URL, making the form simply:

`ftp://username@server-address`

Regardless of the protocol, we should avoid putting login identifiers and especially passwords in URLs. If it is not specified and the resource is protected, let the server issue a challenge so that users provide it directly.

Port

Although not often used, the communication port number in a URL also can be specified. Browsers speaking a particular protocol communicate with servers through entry points, known as ports, which generally are identified by numeric addresses. Associated with each protocol is a default port number. For example, an HTTP request defaults to port number 80. You could say

`http://htmlref.com:80/ch1/fakeexample.html`

but there is no point, as the browser will use the default port for HTTP traffic anyway. However, a server administrator can configure a server to handle protocol requests at ports other than the default numbers. Usually this occurs for experimental or secure applications. In these cases, the intended port must be explicitly addressed in a URL. For example, if we ran another server on port 8080, we would use

`http://notgoingtowork.htmlref.com:8080/ch1/fakeexample.html`

Port number-based access is not terribly user friendly, and it intrinsically provides no extra security other than obscurity.

Query String

Many URLs contain query strings indicated by the question mark (?). When a URL requests a program to be run rather than a file to be returned, a query string might be passed in the URL to indicate the various arguments to be given to the server-side program. Consider, for example,

```
http://www.htmlref.com/fakeexample/registration.php?  
Name=Matt+Folely&Age=32&Sex=male
```

In this situation, the program `registration.php` is handed a query string that has a name value set to “Matt Folely,” an Age value set to “32,” and a Sex value set to “male.” Query strings are generally encoded as discussed in the next section. Spaces in this case are mapped to the plus sign (+), while all other characters are in the %hex value form. The various name-value pairs are separated by ampersands (&). The encoding and decoding of URLs is important for Web developers to understand, and a loose attitude toward allowed encodings can quickly lead to security problems.

Encoding

Some characters may have special meaning within the context of a URL or the operating system of the server on which the resource is found. If any unsafe, reserved, or nonprintable characters occur in a URL, they must be encoded in a special form defined by the MIME type `x-www-form-urlencoded`. Failure to encode special characters may lead to errors, particularly in the presence of Web server security systems such as Web application firewalls.

The form of encoding consists of a percent sign and two hexadecimal digits corresponding to the value of the character in the ASCII character set. Only alphanumeric values and some special characters (\$ - _ . + ! * '), including parentheses, may be used in a URL; other characters should be encoded. In general, special characters such as accents, spaces, and some punctuation marks have to be encoded, depending on the character set in play.

1.7.7 URL Challenges

While we all know and use URLs, we don't necessarily understand all their little quirks. We enumerate a few of the more common challenges faced when working with URLs here.

Unclear Case Sensitivity

Are URLs case sensitive? The answer is, it depends. Domains are not case sensitive. Addresses can be written as `www.Democompany.com` or `www.DEMOCOMPANY.com`. A

browser should handle both properly. Case typically is changed for marketing or branding purposes. However, directory names and filenames following the domain name might be case sensitive, depending on the operating system that the Web server is running on. For example, UNIX systems are case sensitive, whereas Windows machines are not. Then the question arises of query string names and values. Serious trouble can ensue when you are sloppy with case. Assume URLs are case sensitive to avoid headaches.

Unclear Length Limits

How long can a URL be? The answer is unclear. Some documentation suggests low limits, around 255 or 1,024 characters. Other documents indicate there are no limits—the answer is dependent upon many factors. For example, user agents will vary with some supporting user agents and web servers, whatever a system’s maximum string length is. While others are more restrictive or have bugs that restrict URLs to a bit over 1,000 characters. Add in Web servers and security systems, which may have their own limits on allowed URL lengths, and you get the simple answer—nobody knows what the limit may be. Web page authors should assume the worst and use short URLs, 255 chars or lower if at all possible.

Persistence Concerns

Documents move around, servers change names, and documents might eventually be deleted. This is the nature of the Web, and the reason why the 404 Not Found message is so common. When users hit a broken link, they might be at a loss to determine what happened to the document and how to locate its new home. Wouldn’t it be nice if, no matter what happened, a unique identifier indicated where to get a copy of the information? Links can be maintained and errors carefully tracked, but how many developers are really that careful with their URLs?

Long, Dirty URLs

People often have to transcribe addresses. For example, the following is quite a lot to type, read to someone, or avoid not breaking across lines in an e-mail:

`http://www.democompany.com/about/press/pressdetail.php?id=7&view=screen`

Firms are already scrambling for short domain names and paths to improve the typeability of URLs, and most folks tend to omit the protocol when discussing things. Despite these minor clean-ups, many URLs are very long and “dirty,” filled with all sorts of special characters, encouraging fiddling by the mischievous.

Short, Cryptic URLs

Admittedly, URLs can get too long to reasonably type or remember. Worse yet, they may simply be too long for a 140-character Twitter message. Web developers may employ a shortened URL. For example, <http://tinyurl.com/c3l7cq> takes you to the archaic server-side image map example at <http://htmlref.com/ch7/serverimagemap.html>. The shorter URL doesn't tell us much about where we are going. We could be visiting an HTML example, a 1980s pop-video of Rick Astley, or some horrid drive-by malware download. Short URLs may save space, but they are not only cryptic but potentially dangerous. Further, we must hope that the service that powers our shortened URL lives on and that the usage data they glean from watching users traverse the link is not used for troubling ends.

Location, Not Meaning

The primary problem with URLs is that they define location rather than meaning. In other words, URLs specify where something is located on the Web, not what it is or what it's about. This might not seem to be a big deal, but it is. For example, the text of the HTML5 specification is a useful document and certainly has an address at the W3C Web site. But does it live in other places on the Internet? For certain, it can be found at its original parent, WhatWG, and is likely mirrored in a variety of locations. However, if we focus solely on the W3C server and it is unreachable, or DNS services fail to resolve the host, we are stuck if we focus on location. Rather than trying to find a particular document, wherever it might be on the Internet, Web users try to go to a particular location. Rather than talking about where something is, Web users should try to talk about *what* that something is.

1.8 SUMMARY

In this unit, we have learnt the about fundamentals of web and its importance. Also topics such as World Wide Web (WWW), Web browsers, Web servers and Uniform Resource Locators (URL's) are discussed. There are different types of Web browsers and servers available, depending on the need and requirement they are classified into several types which is discussed in this unit. This unit even argues that by separating out the discussion of Web technologies, from the more recent applications and services (social software), and attempts to understand the manifestations and adoption of these services, decision makers will find it easier to understand and act on the strategic implications.

1.9 KEYWORDS

WWW – World Wide Web

URL – Uniform Resource Locator

HTTP – Hyper Text Transfer Protocol

Fragment Identifiers, Kernel mode servers, User mode servers and Single threaded servers.

1.10 UNIT END EXERCISE AND ANSWER

1. Explain the Fundamental of Web in detail.
2. What is Internet? Explain
3. Explain World Wide Web
4. What are Web browsers? Explain in detail.
5. What are Web servers? Explain in detail.
6. What is the key role of Uniform Resource Locator? Explain

1.11 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.
5. HTML5 Black Book: Covers CSS3, Javascript, XML, XHTML, Ajax, PHP and JQuery.

UNIT 2:

Structure:

- 2.0 Objectives
- 2.1 Introduction to MIME
- 2.2 HyperText Transfer Protocol (HTTP)
- 2.3 HTTP Security
- 2.4 Web Programmers Toolbox
- 2.5 Summary
- 2.6 Keywords
- 2.7 Unit-end exercises and answers
- 2.8 Suggested readings

2.0 OBJECTIVES

Virtually all human-written Internet email and a fairly large proportion of automated email are transmitted via SMTP in MIME format. Internet email is so closely associated with the SMTP and MIME standards that it is sometimes called **SMTP/MIME** email. The content types defined by MIME standards are also of importance outside of email, such as in communication protocols like HTTP for the World Wide Web. HTTP requires that data be transmitted in the context of email-like messages, although the data most often is not actually email. MIME is specified in six linked RFC memoranda: RFC 2045, RFC 2046, RFC 2047, RFC 4288, RFC 4289 and RFC 2049, which together define the specifications.

2.1 INTRODUCTION to MIME

Multipurpose Internet Mail Extensions (MIME) is an Internet standard that extends the format of email to support:

- Text in character sets other than ASCII
- Non-text attachments
- Message bodies with multiple parts
- Header information in non-ASCII character sets

The basic Internet email transmission protocol, SMTP, supports only 7-bit ASCII characters. This effectively limits Internet email to messages which, *when transmitted*, include only the characters sufficient for writing a small number of languages, primarily English. Other languages based on the Latin alphabet typically include diacritics and are not supported in 7-bit ASCII, meaning text in these languages cannot be correctly represented in basic email.

MIME defines mechanisms for sending other kinds of information in email. These include text in languages other than English using character encodings other than ASCII, and 8-bit binary content such as files containing images, sounds, movies, and computer programs. Parts of MIME are also reused in communication protocols such as HTTP, which requires that data be transmitted in the context of email-like messages even though the data might not actually have anything to do with email, and the message body can actually be binary. Mapping messages into and out of MIME format is typically done automatically by an email client or by mail servers when sending or receiving Internet (SMTP/MIME) email.

The basic format of Internet email is defined in RFC 5322, which is an updated version of RFC 2822 and RFC 822. These standards specify the familiar formats for text email headers and body and rules pertaining to commonly used header fields such as "To:", "Subject:", "From:", and "Date:". MIME defines a collection of email headers for specifying additional attributes of a message including *content type*, and defines a set of *transfer encodings* which can be used to represent 8-bit binary data using characters from the 7-bit ASCII character set. MIME also specifies rules for encoding non-ASCII characters in email message headers, such as "Subject:", allowing these header fields to contain non-English characters.

MIME is extensible. Its definition includes a method to register new *content types* and other MIME attribute values.

The goals of the MIME definition included requiring no changes to existing email servers and allowing plain text email to function in both directions with existing clients. These goals were achieved by using additional RFC 822-style headers for all MIME message attributes and by making the MIME headers optional with default values ensuring a non-MIME message are interpreted correctly by a MIME-capable client. A simple MIME text message is therefore likely

to be interpreted correctly by a non-MIME client even if it has email headers which the non-MIME client will not know how to interpret. Similarly, if the quoted printable transfer encoding is used, the ASCII part of the message will be intelligible to users with non-MIME clients.

2.1.1 MIME Headers

MIME-Version: The presence of this header indicates the message is MIME-formatted. The value is typically "1.0" so this header appears as

MIME-Version: 1.0

According to MIME co-creator Nathaniel Borenstein, the intention was to allow MIME to change, to advance to version 2.0 and so forth, but this decision led to the opposite outcome, making it nearly impossible to create a new version of the standard.

Content-Type: This header indicates the Internet media type of the message content, consisting of a *type* and *subtype*, for example

Content-Type: text/plain

Through the use of the *multipart* type, MIME allows mail messages to have parts arranged in a tree structure where the leaf nodes are any non-multipart content type and the non-leaf nodes are any of a variety of multipart types. This mechanism supports:

- **Simple text messages using *text/plain*** (the default value for "Content-Type: ")
- Text plus attachments (*multipart/mixed* with a *text/plain* part and other non-text parts). A MIME message including an attached file generally indicates the file's original name with the "Content-disposition:" header, so the type of file is indicated both by the MIME content-type and the (usually OS-specific) filename extension
- Reply with original attached (*multipart/mixed* with a *text/plain* part and the original message as a *message/rfc822* part)
- Alternative content, such as a message sent in both plain text and another format such as HTML (*multipart/alternative* with the same content in *text/plain* and *text/html* forms)
- Image, audio, video and application (for example, *image/jpeg*, *audio/mp3*, *video/mp4*, and *application/msword* and so on)
- Many other message constructs

Content-Disposition: The original MIME specifications only described the structure of mail messages. They did not address the issue of presentation styles. The content-disposition header field was added in RFC 2183 to specify the presentation style. A MIME part can have:

- An *inline* content-disposition, which means that it should be automatically displayed when the message is displayed, or
- An *attachment* content-disposition, in which case it is not displayed automatically and requires some form of action from the user to open it.

In addition to the presentation style, the content-disposition header also provides fields for specifying the name of the file, the creation date and modification date, which can be used by the reader's mail user agent to store the attachment.

The following example is taken from RFC 2183, where the header is defined

```
Content-Disposition: attachment; filename=genome.jpeg;  
modification-date="Wed, 12 Feb 1997 16:29:51 -0500";
```

The filename may be encoded as defined by RFC 2231.

A good majority of mail user agents do not follow this prescription fully. The widely used Mozilla Thunderbird mail client makes its own decisions about which MIME parts should be automatically displayed, ignoring the *content-disposition* headers in the messages. Thunderbird prior to version 3 also sends out newly composed messages with *inline* content-disposition for all MIME parts. Most users are unaware of how to set the content-disposition to *attachment*. Many mail user agents also send messages with the file name in the *name* parameter of the *content-type* header instead of the *filename* parameter of the *content-disposition* header. This practice is discouraged – the file name should be specified either through just the *filename* parameter, or through both the *filename* and the *name* parameters.

In HTTP, the Content-Disposition: attachment response header is usually used to hint to the client to present the response body as a downloadable file. Typically, when receiving such a response, a Web browser will prompt the user to save its content as a file instead of displaying it as a page in a browser window, with the *filename* parameter suggesting the default file name.

Content-Transfer-Encoding: MIME defined a set of methods for representing binary data in formats other than ASCII text format. The *content-transfer-encoding*: MIME header has 2-sided significance:

- It indicates whether or not a binary-to-text encoding scheme has been used on top of the original encoding as specified within the Content-Type header:
 1. If such a binary-to-text encoding method has been used, it states which one.
 2. If not, it provides a descriptive label for the format of content, with respect to the presence of 8 bit or binary content.

The RFC and the IANA's list of transfer encodings define the values shown below, which are not case sensitive. Note that '7bit', '8bit', and 'binary' mean that no binary-to-text encoding on top of the original encoding was used. In these cases, the header is actually redundant for the email client to decode the message body, but it may still be useful as an indicator of what type of object is being sent. Values 'quoted-printable' and 'base64' tell the email client that a binary-to-text encoding scheme was used and that appropriate initial decoding is necessary before the message can be read with its original encoding (e.g. UTF-8).

- Suitable for use with normal SMTP:
 - **7bit** – up to 998 octets per line of the code range 1...127 with CR and LF (codes 13 and 10 respectively) only allowed to appear as part of a CRLF line ending. This is the default value.
 - **Quoted-printable** – used to encode arbitrary octet sequences into a form that satisfies the rules of 7bit. Designed to be efficient and mostly human readable when used for text data consisting primarily of US-ASCII characters but also containing a small proportion of bytes with values outside that range.
 - **base64** – used to encode arbitrary octet sequences into a form that satisfies the rules of 7bit. Designed to be efficient for non-text 8 bit and binary data. Sometimes used for text data that frequently uses non-US-ASCII characters.
- Suitable for use with SMTP servers that support the 8BITMIME SMTP extension:

- **8bit** – up to 998 octets per line with CR and LF (codes 13 and 10 respectively) only allowed to appear as part of a CRLF line ending.
- Suitable only for use with SMTP servers that support the BINARYMIME SMTP extension (RFC 3030):
 - **Binary** – any sequence of octets.

There is no encoding defined which is explicitly designed for sending arbitrary binary data through SMTP transports with the 8BITMIME extension. Thus base64 or quoted-printable must sometimes still be used. This restriction does not apply to other uses of MIME such as Web Services with MIME attachments or MTOM.

2.1.2 ENCODED WORD

Since RFC 2822, conforming message header names and values should be ASCII characters; values that contain non-ASCII data should use the MIME **encoded-word** syntax instead of a literal string. This syntax uses a string of ASCII characters indicating both the original character encoding (the "*charset*") and the content-transfer-encoding used to map the bytes of the charset into ASCII characters.

The form is: "*=?charset?encoding?encoded text?=*".

- *Charset* may be any character set registered with IANA. Typically it would be the same charset as the message body.
- *Encoding* can be either "Q" denoting Q-encoding that is similar to the quoted-printable encoding, or "B" denoting base64 encoding.
- *Encoded text* is the Q-encoded or base64-encoded text.
- An *encoded-word* may not be more than 75 characters long, including *charset*, *encoding*, *encoded text*, and delimiters. If it is desirable to encode more text than will fit in an *encoded-word* of 75 characters, multiple *encoded-words* (separated by CRLF SPACE) may be used.

Difference between Q-encoding and quoted-printable

The ASCII codes for the question mark ("?",) and equals sign ("=") may not be represented directly as they are used to delimit the encoded-word. The ASCII code for space may

not be represented directly because it could cause older parsers to split up the encoded word undesirably. To make the encoding smaller and easier to read the underscore is used to represent the ASCII code for space creating the side effect that underscore cannot be represented directly. Use of encoded words in certain parts of headers imposes further restrictions on which characters may be represented directly.

For example,

Subject: =?iso-8859-1?Q?=A1Hola,_se=F1or!?=

is interpreted as "Subject: ¡Hola, señor!".

The encoded-word format is not used for the names of the headers (for example Subject). These header names are always in English in the raw message. When viewing a message with a non-English email client, the header names are usually translated by the client.

2.1.3 MULTIPART MESSAGES

A MIME multipart message contains a boundary in the "Content-Type: " header; this boundary, which must not occur in any of the parts, is placed between the parts, and at the beginning and end of the body of the message, as follows:

MIME-Version: 1.0

Content-Type: multipart/mixed; boundary=frontier

This is a message with multiple parts in MIME format.

--frontier

Content-Type: text/plain

This is the body of the message.

--frontier

Content-Type: application/octet-stream

Content-Transfer-Encoding: base64

PGh0bWw+CiAgPGhlYWQ+CiAgPC9oZWfkPgogIDxib2R5PgogICAgPHA+VGhpcyBpcyB0a

GUg Ym9keSBvZiB0aGUgbWVzc2FnZS48L3A+CiAgPC9ib2R5Pgo8L2h0bWw+Cg==

--frontier--

Each part consists of its own content header (zero or more *Content-* header fields) and a body. Multipart content can be nested. The content-transfer-encoding of a multipart type must always be "7bit", "8bit" or "binary" to avoid the complications that would be posed by multiple levels of decoding. The multipart block as a whole does not have a charset; non-ASCII characters in the part headers are handled by the Encoded-Word system, and the part bodies can have charsets specified if appropriate for their content-type.

Multipart subtypes: The MIME standard defines various multipart-message subtypes, which specify the nature of the message parts and their relationship to one another. The subtype is specified in the "Content-Type" header of the overall message. For example, a multipart MIME message using the digest subtype would have its Content-Type set as "multipart/digest".

The RFC initially defined 4 subtypes: mixed, digest, alternative and parallel. A minimally compliant application must support mixed and digest; other subtypes are optional. Applications must treat unrecognised subtypes as "multipart/mixed". Additional subtypes, such as signed and form-data, have since been separately defined in other RFCs.

The following is a list of the most commonly used subtypes; it is not intended to be a comprehensive list.

Mixed: Multipart/mixed is used for sending files with different "Content-Type" headers inline (or as attachments). If sending pictures or other easily readable files, most mail clients will display them inline (unless otherwise specified with the "Content-disposition" header). Otherwise it will offer them as attachments. The default content-type for each part is "text/plain".

Digest: Multipart/digest is a simple way to send multiple text messages. The default content-type for each part is "message/rfc822".

Message: A message/rfc822 part contains an email message, including any headers. *Rfc822* is a misnomer, since the message may be a full MIME message. This is used for digests as well as for email forwarding.

Alternative: The multipart/alternative subtype indicates that each part is an "alternative" version of the same (or similar) content, each in a different format denoted by its "Content-Type" header. The formats are ordered by how faithful they are to the original, with the least faithful first and the most faithful last. Systems can then choose the "best" representation they are capable of processing; in general, this will be the last part that the system can understand, although other factors may affect this.

Since a client is unlikely to want to send a version that is less faithful than the plain text version, this structure places the plain text version (if present) first. This makes life easier for users of clients that do not understand multipart messages.

Most commonly, multipart/alternative is used for email with two parts, one plain text (text/plain) and one HTML (text/html). The plain text part provides backwards compatibility while the HTML part allows use of formatting and hyperlinks. Most email clients offer a user option to prefer plain text over HTML; this is an example of how local factors may affect how an application chooses which "best" part of the message to display.

While it is intended that each part of the message represent the same content, the standard does not require this to be enforced in any way. At one time, anti-spam filters would only examine the text/plain part of a message,^[citation needed] because it is easier to parse than the text/html part. But spammers eventually took advantage of this, creating messages with an innocuous-looking text/plain part and advertising in the text/html part. Anti-spam software eventually caught up on this trick, penalizing messages with very different text in a multipart/alternative message.

Related: A multipart/related is used to indicate that each message part is a component of an aggregate whole. It is for compound objects consisting of several inter-related components - proper display cannot be achieved by individually displaying the constituent parts. The message consists of a root part (by default, the first) which reference other parts inline, which may in turn reference other parts. Message parts are commonly referenced by the "Content-ID" part header. The syntax of a reference is unspecified and is instead dictated by the encoding or protocol used in the part.

One common usage of this subtype is to send a web page complete with images in a single message. The root part would contain the HTML document, and use image tags to reference images stored in the latter parts.

Report: *Multipart/report* is a message type that contains data formatted for a mail server to read. It is split between a text/plain (or some other content/type easily readable) and a message/delivery-status, which contains the data formatted for the mail server to read.

Signed: A multipart/signed message is used to attach a digital signature to a message. It has exactly two body parts, a body part and a signature part. The whole of the body part, including

mime headers, is used to create the signature part. Many signature types are possible, like "application/pgp-signature" (RFC 3156) and "application/pkcs7-signature" (S/MIME).

Encrypted: A multipart/encrypted message has two parts. The first part has control information that is needed to decrypt the application/octet-stream second part. Similar to signed messages, there are different implementations which are identified by their separate content types for the control part. The most common types are "application/pgp-encrypted" (RFC 3156) and "application/pkcs7-mime" (S/MIME).

Form Data: As its name implies, multipart/form-data is used to express values submitted through a form. Originally defined as part of HTML 4.0, it is most commonly used for submitting files via HTTP.

Mixed-Replace: The content type multipart/x-mixed-replace was developed as part of a technology to emulate server push and streaming over HTTP.

All parts of a mixed-replace message have the same semantic meaning. However, each part invalidates - "replaces" - the previous parts as soon as it is received completely. Clients should process the individual parts as soon as they arrive and should not wait for the whole message to finish.

Originally developed by Netscape, it is still supported by Mozilla, Firefox, Chrome,^[6] Safari, and Opera, but traditionally ignored by Microsoft. It is commonly used in IP cameras as the MIME type for MJPEG streams.

byteranges: The multipart/byterange is used to represent non-contiguous byte ranges of a single message. It is used by HTTP when a server returns multiple byte ranges and is defined in RFC 2616.

2.2 HYPERTEXT TRANSFER PROTOCOL (HTTP)

The **Hypertext Transfer Protocol (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.

2.2.1 HTTP session

An HTTP session is a sequence of network request-response transactions. An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server. An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own. The body of this message is typically the requested resource, although an error message or other information may also be returned.

2.2.1.1 Request methods: HTTP defines methods to indicate the desired action to be performed on the identified resource. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. Often, the resource corresponds to a file or the output of an executable residing on the server. The HTTP/1.0 specification defined the GET, POST and HEAD methods and the HTTP/1.1 specification added 5 new methods: OPTIONS, PUT, DELETE, TRACE and CONNECT. By being specified in these documents their semantics are well known and can be depended upon. Any client can use any method and the server can be configured to support any combination of methods. If a method is unknown to an intermediate it will be treated as an unsafe and non-idempotent method. There is no limit to the number of methods that can be defined and this allows for future methods to be specified without breaking existing infrastructure.

- **GET**

Requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

- **HEAD**

Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.

- **POST**

Requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, as examples, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database.

- **PUT**

Requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.

- **DELETE**

Deletes the specified resource.

- **TRACE**

Echoes back the received request so that a client can see what (if any) changes or additions have been made by intermediate servers.

- **OPTIONS**

Returns the HTTP methods that the server supports for the specified URL. This can be used to check the functionality of a web server by requesting '*' instead of a specific resource.

- **CONNECT**

Converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.

- **PATCH**

Is used to apply partial modifications to a resource.

HTTP servers are required to implement at least the GET and HEAD methods and, whenever possible, also the OPTIONS method.

2.2.1.2 Safe methods: Some methods (for example, HEAD, GET, OPTIONS and TRACE) are defined as safe, which means they are intended only for information retrieval and should not change the state of the server. In other words, they should not have side effects, beyond relatively harmless effects such as logging, caching, the serving of banner advertisements or incrementing a web counter. Making arbitrary GET requests without regard to the context of the application's state should therefore be considered safe. By contrast, methods such as POST, PUT and DELETE are intended for actions that may cause side effects either on the server, or external side effects such as financial transactions or transmission of email. Such methods are therefore not usually used by conforming web robots or web crawlers; some that do not conform tend to make requests without regard to context or consequences. Despite the prescribed safety of GET requests, in practice their handling by the server is not technically

limited in any way. Therefore, careless or deliberate programming can cause non-trivial changes on the server. This is discouraged, because it can cause problems for Web caching, search engines and other automated agents, which can make unintended changes on the server.

2.2.1.3 Idempotent methods and web applications: Methods PUT and DELETE are defined to be idempotent, meaning that multiple identical requests should have the same effect as a single request. Methods GET, HEAD, OPTIONS and TRACE, being prescribed as safe, should also be idempotent, as HTTP is a stateless protocol. In contrast, the POST method is not necessarily idempotent, and therefore sending an identical POST request multiple times may further affect state or cause further side effects (such as financial transactions). In some cases this may be desirable, but in other cases this could be due to an accident, such as when a user does not realize that their action will result in sending another request, or they did not receive adequate feedback that their first request was successful. While web browsers may show alert dialog boxes to warn users in some cases where reloading a page may re-submit a POST request, it is generally up to the web application to handle cases where a POST request should not be submitted more than once. Note that whether a method is idempotent is not enforced by the protocol or web server.

2.2.1.4 Security: Implementing methods such as TRACE, TRACK and DEBUG are considered potentially insecure by some security professionals because attackers can use them to gather information or bypass security controls during attacks. Security software tools such as Tenable Nessus and Microsoft UrlScan Security Tool report on the presence of these methods as being security issues. TRACK and DEBUG are not valid HTTP 1.1 verbs.

2.2.2 Status codes

In HTTP/1.0 and since, the first line of the HTTP response is called the status line and includes a numeric status code (such as "404") and a textual reason phrase (such as "Not Found"). The way the user agent handles the response primarily depends on the code and secondarily on the response headers. Custom status codes can be used since, if the user agent encounters a code it does not recognize, it can use the first digit of the code to determine the general class of the response.

Also, the standard reason phrases are only recommendations and can be replaced with "local equivalents" at the web developer's discretion. If the status code indicated a problem, the

user agent might display the reason phrase to the user to provide further information about the nature of the problem. The standard also allows the user agent to attempt to interpret the reason phrase, though this might be unwise since the standard explicitly specifies that status codes are machine-readable and reason phrases are human-readable. HTTP status code is primarily divided into five groups for better explanation of request and responses between client and server as named: Informational 1XX, Successful 2XX, Redirection 3XX, Client Error 4XX and Server Error 5XX.

2.2.3 Persistent connections

In HTTP/0.9 and 1.0, the connection is closed after a single request/response pair. In HTTP/1.1 a keep-alive-mechanism was introduced, where a connection could be reused for more than one request. Such persistent connections reduce request latency perceptibly, because the client does not need to re-negotiate the TCP 3-Way-Handshake connection after the first request has been sent. Another positive side effect is that in general the connection becomes faster with time due to TCP's slow-start-mechanism.

Version 1.1 of the protocol also made bandwidth optimization improvements to HTTP/1.0. For example, HTTP/1.1 introduced chunked transfer encoding to allow content on persistent connections to be streamed rather than buffered. HTTP pipelining further reduces lag time, allowing clients to send multiple requests before waiting for each response. Another improvement to the protocol was byte serving, where a server transmits just the portion of a resource explicitly requested by a client.

2.2.4 HTTP session state

HTTP is a stateless protocol. A stateless protocol does not require the HTTP server to retain information or status about each user for the duration of multiple requests. However, some web applications implement states or server side sessions using for instance HTTP cookies or Hidden variables within web forms.

2.2.5 Encrypted connections

The most popular way of establishing an encrypted HTTP connection is HTTP Secure.

Two other methods for establishing an encrypted HTTP connection also exist, called Secure Hypertext Transfer Protocol and the HTTP/1.1 Upgrade header. Browser support, for these latter two, is, however, nearly non-existent, so HTTP Secure is the dominant method of establishing an encrypted HTTP connection.

2.2.6 Request message

The request message consists of the following:

- A request line.
- Request Headers, such as Accept-Language: en
- An empty line.
- An optional message body.

The request line and headers must all end with <CR><LF>. The empty line must consist of only <CR><LF> and no other whitespace. In the HTTP/1.1 protocol, all headers except Host are optional.

A request line containing only the path name is accepted by servers to maintain compatibility with HTTP clients before the HTTP/1.0 specification in RFC 1945.

2.2.7 Response message

The response message consists of the following:

- A Status-Line
- Response Headers, such as Content-Type: text/html
- An empty line
- An optional message body

The Status-Line and headers must all end with <CR><LF> (a carriage return followed by a line feed). The empty line must consist of only <CR><LF> and no other whitespace.

2.3 HTTP SECURITY

Hypertext Transfer Protocol Secure (HTTPS) is a communications protocol for secure communication over a computer network, with especially wide deployment on the Internet. Technically, it is not a protocol in and of itself; rather, it is the

result of simply layering the Hypertext Transfer Protocol (HTTP) on top of the SSL/TLS protocol, thus adding the security capabilities of SSL/TLS to standard HTTP communications.

The security of HTTPS is therefore that of the underlying TLS, which uses long term public and secret keys to exchange a short term session key to encrypt the data flow between client and server. An important property in this context is perfect forward secrecy (PFS), so the short term session key cannot be derived from the long term asymmetric secret key; however, PFS is not widely adopted.

In its popular deployment on the internet, **HTTPS** provides authentication of the web site and associated web server that one is communicating with, which protects against man-in-the-middle attacks. Additionally, it provides bidirectional encryption of communications between a client and server, which protects against eavesdropping and tampering with and/or forging the contents of the communication. In practice, this provides a reasonable guarantee that one is communicating with precisely the web site that one intended to communicate with, as well as ensuring that the contents of communications between the user and site cannot be read or forged by any third party.

Historically, HTTPS connections were primarily used for payment transactions on the World Wide Web, e-mail and for sensitive transactions in corporate information systems. In the late 2000s and early 2010s, HTTPS began to see widespread use for protecting page authenticity on all types of websites, securing accounts and keeping user communications, identity and browsing private.

A site must be completely hosted over HTTPS, without having some of its contents loaded over HTTP, or the user will be vulnerable to some attacks and surveillance. For example, having scripts etc. loaded insecurely on an HTTPS page makes the user vulnerable to attacks. Also having only a certain page that contains sensitive information (such as a log-in page) of a website loaded over HTTPS, while having the rest of the website loaded over plain HTTP, will expose the user to attacks. On a site that has sensitive information somewhere on it, every time that site is accessed with HTTP instead of HTTPS the user and the session will get exposed. Similarly, cookies on a site served through HTTPS have to have the secure attribute enabled.

2.3.1 Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.1 as described by this document. The discussion does not include definitive solutions to the problems revealed, though it does make some suggestions for reducing security risks.

2.3.1.1 Personal Information

HTTP clients are often privy to large amounts of personal information (e.g. the user's name, location, mail address, passwords, encryption keys, etc.), and **SHOULD** be very careful to prevent unintentional leakage of this information via the HTTP protocol to other sources. We very strongly recommend that a convenient interface be provided for the user to control dissemination of such information, and that designers and implementers be particularly careful in this area. History shows that errors in this area often create serious security and/or privacy problems and generate highly adverse publicity for the implementer's company.

Abuse of Server Log Information: A server is in the position to save personal data about a user's requests which might identify their reading patterns or subjects of interest. This information is clearly confidential in nature and its handling can be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

Transfer of Sensitive Information: Like any generic data transfer protocol, HTTP cannot regulate the content of the data that is transferred, nor is there any a priori method of determining the sensitivity of any particular piece of information within the context of any given request. Therefore, applications **SHOULD** supply as much control over this information as possible to the provider of that information. Four header fields are worth special mention in this context: Server, Via, Referer and from.

Revealing the specific software version of the server might allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Implementers **SHOULD** make the Server header field a configurable option.

Proxies which serve as a portal through a network firewall SHOULD take special precautions regarding the transfer of header information that identifies the hosts behind the firewall. In particular, they SHOULD remove, or replace with sanitized versions, any via fields generated behind the firewall.

The Referer header allows reading patterns to be studied and reverse links drawn. Although it can be very useful, its power can be abused if user details are not separated from the information contained in the Referer. Even when the personal information has been removed, the Referer header might indicate a private document's URI whose publication would be inappropriate.

The information sent in the From field might conflict with the user's privacy interests or their site's security policy, and hence it SHOULD NOT be transmitted without the user being able to disable, enable, and modify the contents of the field. The user MUST be able to set the contents of this field within a user preference or application defaults configuration.

The User-Agent or Server header fields can sometimes be used to determine that a specific client or server have a particular security hole which might be exploited. Unfortunately, this same information is often used for other valuable purposes for which HTTP currently has no better mechanism.

Encoding Sensitive Information in URI's: Because the source of a link might be private information or might reveal an otherwise private information source, it is strongly recommended that the user be able to select whether or not the Referer field is sent. For example, a browser client could have a toggle switch for browsing openly/anonymously, which would respectively enable/disable the sending of Referer and from information. Clients SHOULD NOT include a Referer header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol. Authors of services which use the HTTP protocol SHOULD NOT use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the Request-URI. Many existing servers, proxies, and user agents will log the request URI in some place where it might be visible to third parties. Servers can use POST-based form submission instead.

Privacy Issues Connected to Accept Headers: Accept request-headers can reveal information about the user to all servers which are accessed. The Accept-Language header in

particular can reveal information the user would consider to be of a private nature, because the understanding of particular languages is often strongly correlated to the membership of a particular ethnic group. User agents who offer the option to configure the contents of an Accept-Language header to be sent in every request are strongly encouraged to let the configuration process include a message which makes the user aware of the loss of privacy involved.

An approach that limits the loss of privacy would be for a user agent to omit the sending of Accept-Language headers by default, and to ask the user whether or not to start sending Accept-Language headers to a server if it detects, by looking for any Vary response-header fields generated by the server, that such sending could improve the quality of service.

Elaborate user-customized accept header fields sent in every request, in particular if these include quality values, can be used by servers as relatively reliable and long-lived user identifiers. Such user identifiers would allow content providers to do click-trail tracking, and would allow collaborating content providers to match cross-server click-trails or form submissions of individual users. In environments where proxies are used to enhance privacy, users agents ought to be conservative in offering accept header configuration options to end users. As an extreme privacy measure, proxies could filter the accept headers in relayed requests. General purpose user agents who provide a high degree of header configurability **SHOULD** warn users about the loss of privacy which can be involved.

Attacks Based On File and Path Names: Implementations of HTTP origin servers **SHOULD** be careful to restrict the documents returned by HTTP requests to be only those that were intended by the server administrators. If an HTTP server translates HTTP URIs directly into file system calls, the server **MUST** take special care not to serve files that were not intended to be delivered to HTTP clients. For example, UNIX, Microsoft Windows, and other operating systems use "..." as a path component to indicate a directory level above the current one. On such a system, an HTTP server **MUST** disallow any such construct in the Request-URI if it would otherwise allow access to a resource outside those intended to be accessible via the HTTP server. Similarly, files intended for reference only internally to the server (such as access control files, configuration files, and script code) **MUST** be protected from inappropriate retrieval, since they might contain sensitive information. Experience has shown that minor bugs in such HTTP server implementations have turned into security risks.

2.3.2 DNS Spoofing

Clients using HTTP rely heavily on the Domain Name Service, and are thus generally prone to security attacks based on the deliberate mis-association of IP addresses and DNS names. Clients need to be cautious in assuming the continuing validity of an IP number/DNS name association.

In particular, HTTP clients **SHOULD** rely on their name resolver for confirmation of an IP number/DNS name association, rather than caching the result of previous host name lookups. Many platforms already can cache host name lookups locally when appropriate, and they **SHOULD** be configured to do so. It is proper for these lookups to be cached, however, only when the TTL (Time To Live) information reported by the name server makes it likely that the cached information will remain useful.

If HTTP clients cache the results of host name lookups in order to achieve a performance improvement, they **MUST** observe the TTL information reported by DNS.

If HTTP clients do not observe this rule, they could be spoofed when a previously-accessed server's IP address changes. As network renumbering is expected to become increasingly common, the possibility of this form of attack will grow. Observing this requirement thus reduces this potential security vulnerability.

This requirement also improves the load-balancing behavior of clients for replicated servers using the same DNS name and reduces the likelihood of a user's experiencing failure in accessing sites which use that strategy.

2.3.3 Location Headers and Spoofing

If a single server supports multiple organizations that do not trust one another, then it **MUST** check the values of Location and Content-Location headers in responses that are generated under control of said organizations to make sure that they do not attempt to invalidate resources over which they have no authority.

2.3.4 Content-Disposition Issues

RFC 1806, from which the often implemented Content-Disposition header in HTTP is derived, has a number of very serious security considerations. Content-Disposition is not part of

the HTTP standard, but since it is widely implemented, we are documenting its use and risks for implementers.

2.3.5 Authentication Credentials and Idle Clients

Existing HTTP clients and user agents typically retain authentication information indefinitely. HTTP/1.1 does not provide a method for a server to direct clients to discard these cached credentials. This is a significant defect that requires further extensions to HTTP. Circumstances under which credential caching can interfere with the application's security model include but are not limited to:

- Clients which have been idle for an extended period following which the server might wish to cause the client to reprompt the user for credentials.
- Applications which include a session termination indication (such as a 'logout' or 'commit' button on a page) after which the server side of the application 'knows' that there is no further reason for the client to retain the credentials.

This is currently under separate study. There is a number of work-around to parts of this problem, and we encourage the use of password protection in screen savers, idle time-outs, and other methods which mitigate the security problems inherent in this problem. In particular, user agents which cache credentials are encouraged to provide a readily accessible mechanism for discarding cached credentials under user control.

2.3.6 Proxies and Caching

By their very nature, HTTP proxies are men-in-the-middle, and represent an opportunity for man-in-the-middle attacks. Compromise of the systems on which the proxies run can result in serious security and privacy problems. Proxies have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers. A compromised proxy, or a proxy implemented or configured without regard to security and privacy considerations, might be used in the commission of a wide range of potential attacks.

Proxy operators should protect the systems on which proxies run as they would protect any system that contains or transports sensitive information. In particular, log information

gathered at proxies often contains highly sensitive personal information, and/or information about organizations. Log information should be carefully guarded, and appropriate guidelines for use developed and followed.

Caching proxies provide additional potential vulnerabilities, since the contents of the cache represent an attractive target for malicious exploitation. Because cache contents persist after an HTTP request is complete, an attack on the cache can reveal information long after a user believes that the information has been removed from the network. Therefore, cache contents should be protected as sensitive information.

Proxy implementers should consider the privacy and security implications of their design and coding decisions, and of the configuration options they provide to proxy operators.

Users of a proxy need to be aware that they are no trust worthier than the people who run the proxy; HTTP itself cannot solve this problem.

The judicious use of cryptography, when appropriate, may suffice to protect against a broad range of security and privacy attacks. Such cryptography is beyond the scope of the HTTP/1.1 specification.

2.4 WEB PROGRAMMERS TOOLBOX

Client Side	Server Side
Helper Applications	CGI scripts and programs
Netscape Plug-ins	Server API Programs <ul style="list-style-type: none">• ISAPI• NSAPI• Apache Modules
ActiveX Controls	
Java Applets	Java Servlets
Scripting Languages <ul style="list-style-type: none">• JavaScript• VBScript	Server-side scripting <ul style="list-style-type: none">• Active Server Pages(ASP)• ColdFusion• PHP

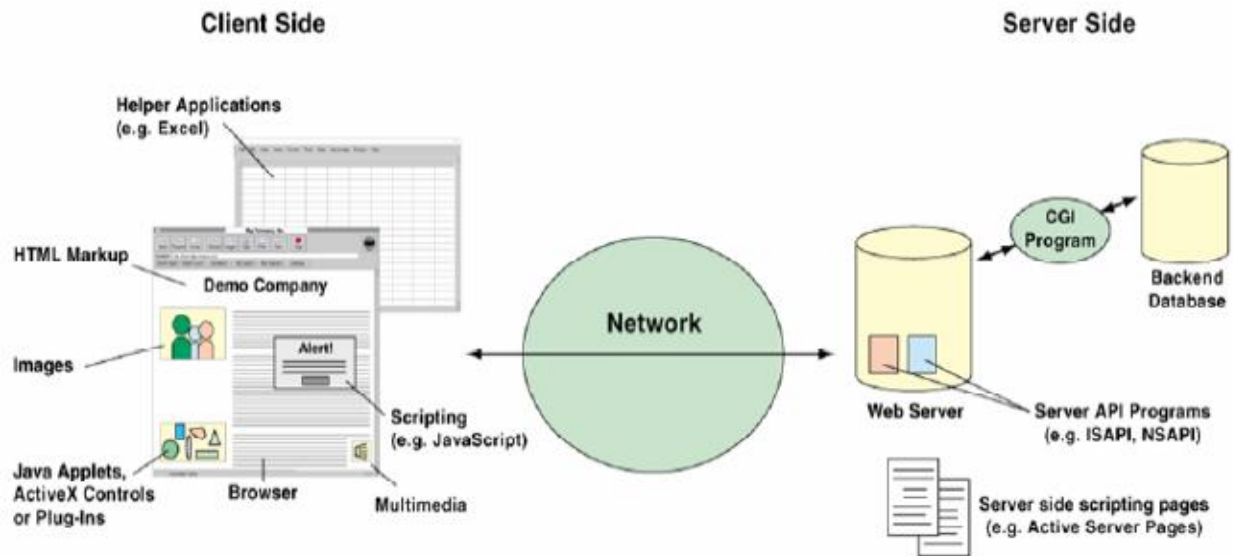


Figure: Web programming Client and Server Side

In the above figure, Client side develops the Web Program by using several tools, some of the tools are Java Applets, ActiveX Controls, Browser, Multimedia etc. The network between the Client side and Server side is linked using this Network. It carries the message from the Client to Server and vice-versa. The Server has the details of all the files which is stored in Backend Database. The CGI Program is the gateway between the Web server and Backend Database. Web is a client server environment built over a network. Web browser typically belongs to the client side since the browser is developed by using the several software. The data which is being used in the browser belongs to the server side since the server holds all the details of the address which is entered in the browser by the client. In some ways Web server is a file server, but it could be more of an application server when interactivity is added. Typically a question arises whether the computing in the web should run in CLIENT SIDE OR SERVER SIDE? The answer to this is it can belong to any of the sides depending on the interactivity from the client side.

Client versus Server

- Client side programming can be problematic because you don't have a great deal of control over what the user has
 - Screen size

- Java/ JavaScript support
- Browser differences
- Hardware
- Server side programming can be problematic because it puts all the responsibility on the server and can cause performance problems
- The best approach is a balance with nothing being too heavily rely on particularly client side wise. Everything should be done in a defensive manner.

CGI Basics

CGI stands for Common Gateway Interface. The most basic server-side way to add interactivity to a Web site is through a CGI program. CGI is not just the program, it is the way things are interfaces. CGI specifies how data should be passed in from a Web page and back out.

CGI Process

- 1) User submits form
- 2) Form is sent to server and eventually CGI program
- 3) CGI program processes data and responds back
- 4) Web server passes CGI response to client

Here Step 2 and the end of Step 3 is the most important steps because the data goes back and forth in these two steps.

2.5 SUMMARY

In this unit, we have learnt the about Introduction of MIME and its importance. MIME allows messages to contain multiple objects. When multiple objects are in a MIME message, they are represented in a form called a body part. A body part has a header and a body, so it makes sense to speak about the body of a body part. Also, body parts can be nested in bodies that contain one or multiple body parts. Also topics such as HyperText Transfer Protocol and its security issues have been discussed here. The different types of Web programmer's toolbox are also discussed in detail.

2.6 KEYWORDS

MIME - Multipurpose Internet Mail Extensions,

DNS – Domain Name System,

Encrypted Connection, Persistent connections, Content-Transfer-Encoding

2.7 UNIT END EXERCISE AND ANSWERS

1. Expand MIME? Explain it in detail
 2. What is the full form of HTTP? Explain HTTP in detail
 3. Explain HTTP security
 4. What are the different Web programmer's toolboxes? Explain
-

2.8 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

UNIT 3:

Structure:

- 3.0 Objectives
- 3.1 Introduction to XHTML
- 3.2 Origin and Evolution of HTML and XHTML
- 3.3 Basic Syntax
- 3.4 Standard XHTML Document Structure
- 3.5 Summary
- 3.6 Keywords
- 3.7 Unit-end exercises and answers
- 3.8 Suggested readings

3.0 OBJECTIVES

At the end of this unit you will be able to know:

- Understanding HTML and XHTML
- Origin, different evolution and basic syntax of HTML and XHTML
- Standard XHTML document structure

3.1 INTRODUCTION TO XHTML

Extensible HyperText Markup Language (XHTML):

XHTML (Extensible HyperText Markup Language) is a family of XML markup languages that mirror or extend versions of the widely used Hypertext Markup Language (HTML), the language in which web pages are written.

While HTML (prior to HTML5) was defined as an application of Standard Generalized Markup Language (SGML), a very flexible markup language framework, XHTML is an

application of XML, a more restrictive subset of SGML. Because XHTML documents need to be well-formed, they can be parsed using standard XML parsers—unlike HTML, which requires a lenient HTML-specific parser.

An XHTML document that conforms to an XHTML specification is said to be *valid*. Validity assures consistency in document code, which in turn eases processing, but does not necessarily ensure consistent rendering by browsers. A document can be checked for validity with the W3C Markup Validation Service. In practice, many web development programs provide code validation based on the W3C standards.

Root element

The root element of an XHTML document must be `html`, and must contain an `xmlns` attribute to associate it with the XHTML namespace. The namespace URI for XHTML is `http://www.w3.org/1999/xhtml`. The example tag below additionally features an `xml:lang` attribute to identify the document with a natural language:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
```

DOCTYPE's

In order to validate an XHTML document, a Document Type Declaration, or *DOCTYPE*, may be used. A DOCTYPE declares to the browser the Document Type Definition (DTD) to which the document conforms. A Document Type Declaration should be placed before the root element.

The system identifier part of the DOCTYPE, which in these examples is the URL that begins with `http://`, need only point to a copy of the DTD to use, if the validator cannot locate one based on the public identifier (the other quoted string). It does not need to be the specific URL that is in these examples; in fact, authors are encouraged to use local copies of the DTD files when possible. The public identifier, however, must be character-for-character the same as in the examples.

XML declaration

A character encoding may be specified at the beginning of an XHTML document in the XML declaration when the document is served using the `application/xhtml+xml` MIME type. (If an XML document lacks encoding specification, an XML parser assumes that the encoding is UTF-8 or UTF-16, unless the encoding has already been determined by a higher protocol.)

For example:

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

The declaration may be optionally omitted because it declares as its encoding the default encoding. However, if the document instead makes use of XML 1.1 or another character encoding, a declaration is necessary. Internet Explorer prior to version 7 enters quirks mode, if it encounters an XML declaration in a document served as text/html.

3.2 ORIGIN AND EVOLUTION OF HTML AND XHTML

3.2.1 HTML (Hypertext Markup Language): HyperText Markup Language is the main markup language for creating web pages and other information that can be displayed in a web browser.

In 1980, physicist Tim Berners-Lee, who was a contractor at CERN, proposed and prototyped ENQUIRE, a system for CERN researchers to use and share documents. In 1989, Berners-Lee wrote a memo proposing an Internet-based hypertext system. Berners-Lee specified HTML and wrote the browser and server software in the later 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. Except for the hyperlink tag, these were strongly influenced by SGMLguid, an in-house SGML-based documentation format at CERN. Eleven of these elements still exist in HTML 4.

HyperText Markup Language is a markup language that web browsers use to interpret and compose text, images and other material into visual or audible web pages. Default characteristics for every item of HTML markup are defined in the browser, and these characteristics can be altered or enhanced by the web page designer's additional use of CSS. Many of the text elements are found in the 1988 ISO technical report TR 9537 *Techniques for using SGML*, which in turn covers the features of early text formatting languages such as that used by the RUNOFF command developed in the early 1960s for the CTSS (Compatible Time-Sharing System) operating system: these formatting commands were derived from the commands used by typesetters to manually format documents. However, the SGML concept of generalized markup is based on elements rather than merely print effects, with also the

separation of structure and markup; HTML has been progressively moved in this direction with CSS.

Berners-Lee considered HTML to be an application of SGML. It was formally defined as such by the Internet Engineering Task Force (IETF) with the mid-1993 publication of the first proposal for an HTML specification: "Hypertext Markup Language (HTML)" Internet-Draft by Berners-Lee and Dan Connolly, which included an SGML Document Type Definition to define the grammar. The draft expired after six months, but was notable for its acknowledgment of the NCSA Mosaic browser's custom tag for embedding in-line images, reflecting the IETF's philosophy of basing standards on successful prototypes. Similarly, Dave Raggett's competing Internet-Draft, "HTML+ (Hypertext Markup Format)", from late 1993, suggested standardizing already-implemented features like tables and fill-out forms.

After the HTML and HTML+ drafts expired in early 1994, the IETF created an HTML Working Group, which in 1995 completed "HTML 2.0", the first HTML specification intended to be treated as a standard against which future implementations should be based.

Further development under the auspices of the IETF was stalled by competing interests. Since 1996, the HTML specifications have been maintained, with input from commercial software vendors, by the World Wide Web Consortium (W3C). However, in 2000, HTML also became an international standard (ISO/IEC 15445:2000). HTML 4.01 was published in late 1999, with further errata published through 2001. In 2004 development began on HTML5 in the Web Hypertext Application Technology Working Group (WHATWG), which became a joint deliverable with the W3C in 2008.

HTML versions timeline

November 24, 1995

HTML 2.0 was published as IETF RFC 1866 . Supplemental RFCs added capabilities:

- November 25, 1995: RFC 1867 (form-based file upload)
- May 1996: RFC 1942 (tables)
- August 1996: RFC 1980 (client-side image maps)
- January 1997: RFC 2070 (internationalization)

January 1997

HTML 3.2 was published as a W3C Recommendation. It was the first version developed and standardized exclusively by the W3C, as the IETF had closed its HTML Working Group in September 1996.

Initially code-named "Wilbur", HTML 3.2 dropped math formulas entirely, reconciled overlap among various proprietary extensions and adopted most of Netscape's visual markup tags. Netscape's blink element and Microsoft's marquee element were omitted due to a mutual agreement between the two companies. A markup for mathematical formulas similar to that in HTML was not standardized until 14 months later in MathML.

December 1997

HTML 4.0 was published as a W3C Recommendation . It offers three variations:

- Strict, in which deprecated elements are forbidden,
- Transitional, in which deprecated elements are allowed,
- Frameset, in which mostly only frame related elements are allowed ;

Initially code-named "Cougar", HTML 4.0 adopted many browser-specific element types and attributes, but at the same time sought to phase out Netscape's visual markup features by marking them as deprecated in favor of style sheets. HTML 4 is an SGML application conforming to ISO 8879 – SGML.

April 1998

HTML 4.0 was reissued with minor edits without incrementing the version number.

December 1999

HTML 4.01 was published as a W3C Recommendation. It offers the same three variations as HTML 4.0 and its last errata were published May 12, 2001.

May 2000

ISO/IEC 15445:2000 was published as an ISO/IEC international standard. In the ISO this standard falls in the domain of the ISO/IEC JTC1/SC34 (ISO/IEC Joint Technical Committee 1, Subcommittee 34 – Document description and processing languages).

As of mid-2008, HTML 4.01 and ISO/IEC 15445:2000 are the most recent versions of HTML. Development of the parallel, XML-based language XHTML occupied the W3C's HTML Working Group through the early and mid-2000s.

HTML draft version timeline

October 1991

HTML Tags, an informal CERN document listing 18 HTML tags, was first mentioned in public.

June 1992

First informal draft of the HTML DTD, with seven subsequent revisions (July 15, August 6, August 18, November 17, November 19, November 20, November 22)

November 1992

HTML DTD 1.1 (the first with a version number, based on RCS revisions, which start with 1.1 rather than 1.0), an informal draft

June 1993

Hypertext Markup Language was published by the IETF IIR Working Group as an Internet-Draft (a rough proposal for a standard). It was replaced by a second version one month later, followed by six further drafts published by IETF itself that finally led to HTML 2.0 in RFC1866

November 1993

HTML+ was published by the IETF as an Internet-Draft and was a competing proposal to the Hypertext Markup Language draft. It expired in May 1994.

April 1995 (authored March 1995)

HTML 3.0 was proposed as a standard to the IETF, but the proposal expired five months later without further action. It included many of the capabilities that were in Raggett's HTML+ proposal, such as support for tables, text flow around figures and the display of complex mathematical formulas.

W3C began development of its own Arena browser as a test bed for HTML 3 and Cascading Style Sheets, but HTML 3.0 did not succeed for several reasons. The draft was considered very large at 150 pages and the pace of browser development, as well as the number of interested parties, had outstripped the resources of the IETF. Browser vendors, including

Microsoft and Netscape at the time, chose to implement different subsets of HTML 3's draft features as well as to introduce their own extensions to it. These included extensions to control stylistic aspects of documents, contrary to the "belief [of the academic engineering community] that such things as text color, background texture, font size and font face were definitely outside the scope of a language when their only intent was to specify how a document would be organized." Dave Raggett, who has been a W3C Fellow for many years has commented for example, "To a certain extent, Microsoft built its business on the Web by extending HTML features."

January 2008

HTML5 was published as a Working Draft by the W3C.

Although its syntax closely resembles that of SGML, HTML5 has abandoned any attempt to be an SGML application and has explicitly defined its own "html" serialization, in addition to an alternative XML-based XHTML5 serialization.

May 2011

On 14 February 2011, the W3C extended the charter of its HTML Working Group with clear milestones for HTML5. In May 2011, the working group advanced HTML5 to "Last Call", an invitation to communities inside and outside W3C to confirm the technical soundness of the specification. The W3C is developing a comprehensive test suite to achieve broad interoperability for the full specification by 2014, which is now the target date for Recommendation.

3.2.2 XHTML (EXtensible HyperText Markup Language): XHTML (Extensible HyperText Markup Language) is a family of XML markup languages that mirror or extend versions of the widely used Hypertext Markup Language (HTML), the language in which web pages are written.

XHTML 1.0

December 1998 saw the publication of a W3C Working Draft entitled *Reformulating HTML in XML*. This introduced Voyager, the codename for a new markup language based on HTML 4, but adhering to the stricter syntax rules of XML. By February 1999 the name of the specification had changed to *XHTML 1.0. The Extensible HyperText Markup Language*, and in

January 2000 it was officially adopted as a W3C Recommendation. There are three formal DTDs for XHTML 1.0, corresponding to the three different versions of HTML 4.01:

- **XHTML 1.0 Strict** is the XML equivalent to strict HTML 4.01, and includes elements and attributes that have not been marked deprecated in the HTML 4.01 specification. As of May 25, 2011, XHTML 1.0 Strict is the document type used for the homepage of the website of the World Wide Web Consortium.
- **XHTML 1.0 Transitional** is the XML equivalent of HTML 4.01 Transitional, and includes the presentational elements (such as center, font and strike) excluded from the strict version.
- **XHTML 1.0 Frameset** is the XML equivalent of HTML 4.01 Frameset, and allows for the definition of frameset documents—a common Web feature in the late 1990s.

The second edition of XHTML 1.0 became a W3C Recommendation in August 2002.

Modularization of XHTML

Modularization provides an abstract collection of components through which XHTML can be sub settled and extended. The feature is intended to help XHTML extend its reach onto emerging platforms, such as mobile devices and Web-enabled televisions. The initial draft of *Modularization of XHTML* became available in April 1999, and reached Recommendation status in April 2001.

The first modular XHTML variants were XHTML 1.1 and XHTML Basic 1.0.

In October 2008 *Modularization of XHTML* was superseded by *XHTML Modularization 1.1*, which adds an XML Schema implementation. It was itself superseded by a second edition in July 2010.

XHTML 1.1: Module-based XHTML

XHTML 1.1 evolved out of the work surrounding the initial Modularization of XHTML specification. The W3C released a first draft in September 1999; Recommendation status was reached in May 2001. The modules combined within XHTML 1.1 effectively recreate XHTML 1.0 Strict, with the addition of ruby annotation elements (ruby, rbc, rtc, rb, rt and rp) to better support East-Asian languages. Other changes include removal of the name attribute from the a and map elements, and (in the first edition of the language) removal of the lang attribute in favour of xml:lang.

Although XHTML 1.1 is largely compatible with XHTML 1.0 and HTML 4, in August 2002 the Working Group issued a formal Note advising that it should not be transmitted with the HTML media type. With limited browser support for the alternate application/xhtml+xml media type, XHTML 1.1 proved unable to gain widespread use. In January 2009 a second edition of the document (XHTML Media Types - Second Edition) was issued, relaxing this restriction and allowing XHTML 1.1 to be served as text/html.

A second edition of XHTML 1.1 was issued on 23 November 2010, which addresses various errata and adds an XML Schema implementation not included in the original specification.

Of all the versions of XHTML, XHTML Basic 1.0 provides the fewest features. With XHTML 1.1, it is one of the two first implementations of modular XHTML. In addition to the Core Modules (Structure, Text, Hypertext, and List), it implements the following abstract modules: Base, Basic Forms, Basic Tables, Image, Link, Metainformation, Object, Style Sheet, and Target.

XHTML Basic 1.1 replaces the Basic Forms Module with the Forms Module, and adds the Intrinsic Events, Presentation, and Scripting modules. It also supports additional tags and attributes from other modules. This version became a W3C recommendation on 29 July 2008.

The current version of XHTML Basic is 1.1 Second Edition (23 November 2010), in which the language is re-implemented in the W3C's XML Schema language. This version also supports the lang attribute.

XHTML-Print: XHTML-Print, which became a W3C Recommendation in September 2006, is a specialized version of XHTML Basic designed for documents printed from information appliances to low-end printers.

XHTML Mobile Profile

XHTML Mobile Profile (abbreviated XHTML MP or XHTML-MP) is a third-party variant of the W3C's XHTML Basic specification. Like XHTML Basic, XHTML was developed for information appliances with limited system resources.

In October 2001, a limited company called the Wireless Application Protocol Forum began adapting XHTML Basic for WAP 2.0, the second major version of the Wireless

Application Protocol. WAP Forum based their DTD on the W3C's Modularization of XHTML, incorporating the same modules the W3C used in XHTML Basic 1.0—except for the Target Module. Starting with this foundation, the WAP Forum replaced the Basic Forms Module with a partial implementation of the Forms Module, added partial support for the Legacy and Presentation modules, and added full support for the Style Attribute Module.

In 2002, the WAP Forum was subsumed into the Open Mobile Alliance (OMA), which continued to develop XHTML Mobile Profile as a component of their OMA Browsing Specification.

XHTML Mobile Profile 1.1

To this version, finalized in 2004, the OMA added partial support for the Scripting Module, and partial support for Intrinsic Events. XHTML MP 1.1 is part of v2.1 of the OMA Browsing Specification (1 November 2002).

XHTML Mobile Profile 1.2

This version, finalized 27 February 2007, expands the capabilities of XHTML MP 1.1 with full support for the Forms Module and OMA Text Input Modes. XHTML MP 1.2 is part of v2.3 of the OMA Browsing Specification (13 March 2007).

XHTML Mobile Profile 1.3

XHTML MP 1.3 (finalized on 23 September 2008) uses the XHTML Basic 1.1 document type definition, which includes the Target Module. Events in this version of the specification are updated to DOM Level 3 specifications (i.e., they are platform- and language-neutral).

XHTML 1.2

The XHTML 2 Working Group considered the creation of a new language based on XHTML 1.1. If XHTML 1.2 was created, it would include WAI-ARIA and role attributes to better support accessible web applications, and improved Semantic Web support through RDFa. The input mode attribute from XHTML Basic 1.1, along with the target attribute (for specifying frame targets) might also be present. The XHTML2 WG had not been chartered to carry out the development of XHTML1.2. Since the W3C announced that it does not intend to recharter the XHTML2 WG, and closed the WG in December 2010, this means that XHTML 1.2 proposal would not eventuate.

XHTML 2.0

Between August 2002 and July 2006, the W3C released eight Working Drafts of XHTML 2.0, a new version of XHTML able to make a clean break from the past by discarding the requirement of backward compatibility. This lack of compatibility with XHTML 1.x and HTML 4 caused some early controversy in the web developer community. Some parts of the language (such as the role and RDFa attributes) were subsequently split out of the specification and worked on as separate modules, partially to help make the transition from XHTML 1.x to XHTML 2.0 smoother. A ninth draft of XHTML 2.0 was expected to appear in 2009, but on July 2, 2009, the W3C decided to let the XHTML2 Working Group charter expire by that year's end, effectively halting any further development of the draft into a standard. Instead, XHTML 2.0 and its related documents were released as W3C Notes.

New features to have been introduced by XHTML 2.0 included:

- HTML forms were to be replaced by XForms, an XML-based user input specification allowing forms to be displayed appropriately for different rendering devices.
- HTML frames were to be replaced by XFrames.
- The DOM Events were to be replaced by XML Events, which uses the XML Document Object Model.
- A new list element type, the `nl` element type, were to be included to specifically designate a list as a navigation list. This would have been useful in creating nested menus, which are currently created by a wide variety of means like nested unordered lists or nested definition lists.
- Any element was to be able to act as a hyperlink, e. g., `<li href="articles.html">Articles`, similar to XLink. However, XLink itself is not compatible with XHTML due to design differences.
- Any element was to be able to reference alternative media with the `src` attribute, e. g., `<p src="lbridge.jpg" type="image/jpeg">London Bridge</p>` is the same as `<object src="lbridge.jpg" type="image/jpeg"><p>London Bridge</p></object>`.
- The `alt` attribute of the `img` element was removed: alternative text was to be given in the content of the `img` element, much like the `object` element, e. g., `HMS Audacious`.

- A single heading element (h) was added. The level of these headings was determined by the depth of the nesting. This would have allowed the use of headings to be infinite, rather than limiting use to six levels deep.
- The remaining presentational elements i, b and tt, still allowed in XHTML 1.x (even Strict), were to be absent from XHTML 2.0. The only somewhat presentational elements remaining were to be sup and sub for superscript and subscript respectively, because they have significant non-presentational uses and are required by certain languages. All other tags were meant to be semantic instead (e. g. strong for **strong emphasis**) while allowing the user agent to control the presentation of elements via CSS (e.g. rendered as boldface text in most visual browsers, but possibly rendered with changes of tone in a text-to-speech reader, larger + italic font per rules in a user-end stylesheet, etc.).
- The addition of RDF triple with the property and about attributes to facilitate the conversion from XHTML to RDF/XML.

XHTML5

HTML5 initially grew independently of the W3C, through a loose group of browser manufacturers and other interested parties calling themselves the WHATWG, or Web Hypertext Application Technology Working Group. The WHATWG announced the existence of an open mailing list in June 2004, along with a website bearing the strapline “Maintaining and evolving HTML since 2004.” The key motive of the group was to create a platform for dynamic web applications; they considered XHTML 2.0 to be too document-centric, and not suitable for the creation of internet forum sites or online shops.

In April 2007, the Mozilla Foundation and Opera Software joined Apple in requesting that the newly re-chartered HTML Working Group of the W3C adopt the work, under the name of HTML5. The group resolved to do this the following month, and the First Public Working Draft of HTML5 was issued by the W3C in January 2008. The most recent W3C Working Draft was published in January 2011.

HTML5 has both a regular text/html serialization and an XML serialization, which is known as XHTML5. In addition to the markup language, the specification includes a number of application programming interfaces. The Document Object Model is extended with APIs for editing, drag-and-drop, and data storage and network communication.

The language is more compatible with HTML 4 and XHTML 1.x than XHTML 2.0, due to the decision to keep the existing HTML form elements and events model. It adds many new elements not found in XHTML 1.x, however, such as section and aside.

The most recent draft includes WAI-ARIA support.

XHTML 1.0 became a World Wide Web Consortium (W3C) Recommendation on January 26, 2000. XHTML 1.1 became a W3C Recommendation on May 31, 2001. XHTML5 is undergoing development as of September 2009, as part of the HTML5 specification.

3.3 BASIC SYNTAX

Elements are defined by tags (markers). The tag format is as follows:

- Opening tag: `<name [/]>`
This indicates that the it is the beginning of the content.
- Closing tag: `</name>`
This indicates the closing of the content being entered in the tag.

The content of a tag appears between its opening tag and the closing tag. It is not mandatory that all tag must have content. If a tag has no content, its form is `<name />`. If a tag has a attributes, they appear between its name and the right bracket. The tag name and the attribute name must be written in lowercase letters. Every tag that has content must have a closing tag. The tag must be properly nested or it may end up in mixing up the contents. Some of the commonly used tags are: `<html>`, `<head>`, `<title>`, and `<body>`.

Comments are also the basic syntax to make the programmer understand what exactly an instruction does. Some of the comment form is: `<!-- ... -->`. Browsers ignore comments, unrecognized tags, line breaks, multiple spaces and tabs (Explained in the section 3.4). Tags are suggestions to the browser, even if they are recognized by the browser.

3.4 STANDARD XHTML DOCUMENT STRUCTURE

A basic XHTML document consists of the following main parts:

- The *DOCTYPE* (DTD)
- *html* document root

- *xmlns* attribute for the *html* element
- *head* element with a child *title* element
- *body* element

Example: Standard XHTML Document Structure:

```
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>...</title>
  </head>
  <body>...</body>
</html>
```

This section covers basic XHTML elements, such as those used to create new paragraphs, headers, page breaks, and comments.

NOTE:

1. The indentation in the markup examples is for readability. Indentation is not required, but is recommended to make your markup easier for humans to read.
2. Blocks of XHTML markup that are highlighted in a bold-faced, red font represent the part of an example that is being emphasized.

3.4.1 Opening declarations

Head and body: The head section defines the title of the page and other information that isn't usually visible to the viewer of your page, but is useful to search engines. The body section contains the content of the Web page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
```

```

        </head>

        <body>

        </body>

</html>

```

Encoding: All markup documents should specify the character encoding in which they should be displayed. The most accepted character encoding for XHTML markup is UTF-8 (8-bit Unicode Transformation Format). Briefly, Unicode is an industry standard allowing computers to consistently represent and manipulate text expressed in any of the world's writing systems. The meta element is used to define the character encoding. In the example markup below, the http-equiv attribute indicates that the given meta element is specifying information about the content of the markup. The content attribute indicates that the markup is text-based HTML with a character set of UTF-8. Typically you can simply copy the meta element shown below directly into your XHTML document and not worry about the meaning of the various attribute values.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="content-type" content="text/html; charset=utf-8" />

  </head>

  <body>

    //Enter the content of your interest

  </body>

</html>

```

3.4.2 Some XHTML elements

Title: Each XHTML page must have a title element. A title should be short and descriptive, because the text appears in the title bar of the browser window, and more importantly, is used in search engine results. A title cannot contain formatting, images, or links.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
```

```
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
```

```
<title>Programming is Fun!</title>
```

```
</head>
```

```
<body>
```

```
</body>
```

```
</html>
```



Headers: XHTML provides six levels of headers, h1 through h6, which can be used to break your page into sections. Think of headers as hierarchical dividers like those used in an outline. Smaller numbered headers represent bigger chunks of the document, such as chapters, and are displayed more prominently than higher numbered headers.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="content-type" content="text/html; charset=utf-8" />

    <title>Programming is Fun!</title>

  </head>

  <body>

    <h1>History of Programming</h1>

    <h2>Before Computers</h2>

    <h2>After Computers</h2>

  </body>

</html>

```

This is how the page may appear in the browser.

History of Programming
Before Computers
After Computers

Paragraphs: Paragraphs cannot simply be created by adding newlines because XHTML parsers ignore newlines or other extra white space in the markup. To start a new paragraph, use the p element.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="content-type" content="text/html; charset=utf-8" />

    <title>Programming is Fun!</title>

```

```
</head>

<body>

  <h1>History of Programming</h1>

  <p>We will now discuss the history of programming. </p>


  <h2>Before Computers</h2>

  <p>The abacus was about as fun as a bag of rocks. </p>

  <p>You might try looking up Charles Babbage on Wikipedia. </p>


  <h2>After Computers</h2>

  <p>Fun, joy, excitement, money!</p>

</body>

</html>
```

This is how the page may appear in the browser.

History of Programming

We will now discuss the history of programming.

Before Computers

The abacus was about as fun as a bag of rocks.

You might try looking up Charles Babbage on Wikipedia.

After Computers

Fun, joy, excitement, money!

Divisions: Divisions are denoted using the `div` element, and are primarily used to break text into blocks that can be formatted using cascading style sheets. Although not required, you can uniquely identify a division by using the `id` attribute, or identify classes of divisions using the `class` attribute. These attributes are helpful for applying styles. For example, you may want to format the header and introductory text in the History of Programming section differently than other level-one headers, while using the standard formatting for the subsections. The following specification accomplishes these goals:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Programming is Fun!</title>
  </head>
  <body>
    <div id="history">
      <h1>History of Programming</h1>
      <p>We will now discuss the history of programming.</p>
    </div>

    <div class="subsection">
      <h2>Before Computers</h2>
      <p>The abacus was about as fun as a bag of rocks.</p>
      <p>You might try looking up Charles Babbage on Wikipedia.</p>
    </div>

    <div class="subsection">
      <h2>After Computers</h2>
      <p>Fun, joy, excitement, money!</p>
    </div>
  </body>
</html>

```

Spans: Spans are used to identify smaller blocks of text than headers or divisions, and are primarily used to format inline text with styles. In the example below, spans are used to identify key terms and people.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Programming is Fun!</title>
  </head>
  <body>
    <div id="history">
      <h1>History of Programming</h1>
      <p>We will now discuss the <span class="keyterm">history of programming</span>.</p>
    </div>

    <div class="subsection">
      <h2>Before Computers</h2>
      <p>The <span class="keyterm">abacus</span> was about as fun as a bag of rocks.</p>
      <p>You might try looking up <span class="person">Charles Babbage</span> on
Wikipedia.</p>
    </div>

    <div class="subsection">
      <h2>After Computers</h2>
      <p><span class="keyterm">Fun</span>, joy, excitement, money!</p>
```

```
</div>
</body>
</html>
```

Line breaks: Browsers ignore line breaks in your text and automatically wrap lines. You can force the browser to break a line by using the `br` element. Line breaks are appropriate for short lines of text that should appear one after another without much space in between. In the example below, line breaks are used to format a table of contents.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Programming is Fun!</title>
  </head>
  <body>
    <div id="tableofcontents">History of Programming<br />
      Before Computers<br />
      After Computers<br />
    </div>

    <div id="history">
      <h1>History of Programming</h1>
      <p>We will now discuss the <span class="keyterm">history of programming</span>.</p>
    </div>
```



```

<div class="subsection">

  <h2>Before Computers</h2>

  <p>The <span class="keyterm">abacus</span> was about as fun as a bag of rocks.</p>

  <p>You might try looking up <span class="person">Charles Babbage</span> on
Wikipedia.</p>

</div>

<div class="subsection">

  <h2>After Computers</h2>

  <p><span class="keyterm">Fun</span>, joy, excitement, money!</p>

</div>

</body>

</html>

```

This is how the page may appear in the browser.

History of Programming
 Before Computers
 After Computers

History of Programming

We will now discuss the history of programming.

Before Computers

The abacus was about as fun as a bag of rocks.

You might try looking up Charles Babbage on Wikipedia.

After Computers

Fun, joy, excitement, money!

Comments: Comments are useful for reminding yourself or informing others about what your markup should do. Comments do not appear in the page text shown to the user. To create a comment, start with `<!--`, type your comment, then close it with `-->`. *NOTE:* Comments cannot be nested in other comments.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="content-type" content="text/html; charset=utf-8" />

    <title>Programming is Fun!</title>

  </head>

  <body>

    <!-- Ye olde table of contents -->

    <div id="tableofcontents">History of Programming<br />

      Before Computers<br />

      After Computers<br />

    </div>

    <div id="history">

      <h1>History of Programming</h1>

      <p>We will now discuss the <span class="keyterm">history of programming</span>.</p>

    </div>

    <div class="subsection"> <!-- The first subsection -->

      <h2>Before Computers</h2>

      <p>The <span class="keyterm">abacus</span> was about as fun as a bag of rocks.</p>

      <p>You might try looking up <span class="person">Charles Babbage</span> on
Wikipedia.</p>

    </div>

    <div class="subsection">

      <h2>After Computers</h2>

      <p><span class="keyterm">Fun</span>, joy, excitement, money!</p>

```

```
</div>

<!-- More stuff should probably go here -->

</body>

</html>
```

Tooltips: You can use the title attribute of most XHTML elements to add a tooltip, a small string of text that appears when the mouse cursor is positioned over a particular XHTML element for a few seconds. For example, you can add a tooltip for the table of contents section so that the user knows why the text is there.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="content-type" content="text/html; charset=utf-8" />

    <title>Programming is Fun!</title>

  </head>

  <body>

    <!-- Ye olde table of contents -->

    <div id="tableofcontents" title="Table of Contents">History of Programming<br />
      Before Computers<br />
      After Computers<br />
    </div>

    <div id="history">

      <h1>History of Programming</h1>

      <p>We will now discuss the <span class="keyterm">history of programming</span>.</p>

    </div>
```

3.5 SUMMARY

In this unit, we have learnt the about Introduction to XHTML. Origin and different evolution of HTML and XHTML have been discussed here. The basic syntax of HTML and XHTML have been discussed in detail. The Standard XHTML document structure have been discussed in detail.

3.6 KEYWORDS

HTML – HyperText Markup Language,

XHTML – Extensible HyperText Markup Language,

W3C markup, XML, URL.

3.7 UNIT END EXERCISE AND ANSWERS

- 1) Explain XHTML in detail.
- 2) Explain the origin and evolution of HTML and XHTML
- 3) What are the Basic Syntax in HTML and XHTML? Explain using the syntax
- 4) What is the Standard XHTML document structure in detail?

3.8 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

UNIT 4:

Structure:

- 4.0 Objectives
- 4.1 Introduction to Markup Language
- 4.2 Types of Markup Language
- 4.3 Basic Text Markup
- 4.4 Links and Images
- 4.5 Lists
- 4.6 Tables
- 4.7 Forms
- 4.8 Frames
- 4.9 HyperText Links
- 4.10 HTML versus XHTML
- 4.11 Summary
- 4.12 Keywords
- 4.13 Unit-end exercises and answers
- 4.14 Suggested readings

4.0 OBJECTIVES

At the end of this unit you will be able to know:

- Different types of Markup Language and Basic Text Markup
- How to use the different types of links, images, list, tables, forms, frames have been explained.
- How to give a HyperText Link to other website or other pages.
- Difference between HTML and XHTML

4.1 INTRODUCTION TO MARKUP LANGUAGE

A **(document) markup language** is a modern system for annotating a document in a way that is syntactically distinguishable from the text. The idea and terminology evolved 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".

Examples are 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, but is omitted from the version of the text that is displayed to users. Some markup languages, such as 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.

A widely used markup language is HyperText Markup Language (HTML), one of the document formats of the World Wide Web. HTML, which is an instance of SGML (though, strictly, it does not comply with all the rules of SGML), follows many of the markup conventions used in the publishing industry in the communication of printed work between authors, editors, and printers.

4.2 TYPES OF MARKUP LANGUAGE

There are three general categories of electronic markup:

4.2.1 Presentational markup

The kind of markup used by traditional word-processing systems: binary codes embedded in document text that produces the WYSIWYG effect. Such markup is usually designed to be hidden from human users, even those who are authors or editors.

4.2.2 Procedural markup

Markup is embedded in text and provides instructions for programs that are to process the text. Well-known examples include troff, LaTeX, and PostScript. It is expected that the processor will run through the text from beginning to end, following the instructions as encountered. Text with such markup is often edited with the markup visible and directly manipulated by the author. Popular procedural-markup systems usually include programming constructs, so macros or subroutines can be defined and invoked by name.

4.2.3 Descriptive markup

Markup is used to label parts of the document rather than to provide specific instructions as to how they should be processed. The objective is to decouple the inherent structure of the document from any particular treatment or rendition of it. Such markup is often described as

"semantic". An example of descriptive markup would be HTML's `<cite>` tag, which is used to label a citation.

There is considerable blurring of the lines between the types of markup. In modern word-processing systems, presentational markup is often saved in descriptive-markup-oriented systems such as XML, and then processed procedurally by implementations. The programming constructs in procedural-markup systems such as TeX may be used to create higher-level markup systems which are more descriptive, such as LaTeX.

In recent years, a number of small and largely un-standardized markup languages have been developed to allow authors to create formatted text via web browsers, for use in wikis and web forums. These are sometimes called lightweight markup languages.

Descriptive markup -- sometimes called *logical markup* or *conceptual markup* -- encourages writing in a way that describes the material conceptually, rather than visually.

4.3 BASIC TEXT MARKUP

Text Markup — Simple markup language to encode text with attributes. Explanation of the text markup is explained using the Pango Text Attribute Markup

Frequently, you want to display some text to the user with attributes applied to part of the text (for example, you might want bold or italicized words). With the base Pango interfaces, you could create a [PangoAttrList](#) and apply it to the text; the problem is that you'd need to apply attributes to some numeric range of characters, for example "characters 12-17." This is broken from an internationalization standpoint; once the text is translated, the word you wanted to italicize could be in a different position.

The solution is to include the text attributes in the string to be translated. Pango provides this feature with a small markup language. You can parse a marked-up string into the string text plus a [PangoAttrList](#) using either of [pango_parse_markup\(\)](#) or [pango_markup_parser_new\(\)](#).

A simple example of a marked-up string might be: "`Blue text` is `<i>cool</i>!`"

Pango uses #G Markup to parse this language, which means that XML features such as numeric character entities such as `©` for © can be used too.

The root tag of a marked-up document is `<markup>`, but `pango_parse_markup()` allows you to omit this tag, so you will most likely never need to use it. There are some convenience tags.

<code>b</code>	Bold
<code>big</code>	Makes font relatively larger, equivalent to <code></code>
<code>i</code>	Italic
<code>s</code>	Strikethrough
<code>sub</code>	Subscript
<code>sup</code>	Superscript
<code>small</code>	Makes font relatively smaller, equivalent to <code></code>
<code>tt</code>	Monospace font
<code>u</code>	Underline

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

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

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

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

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

4.4 LINKS AND IMAGES

Link and image markup are a form of phrase markup with a little structure added. The `<` and `>` modifiers when applied to images cause them to *float*.

Markup		HTML
<code>"text":url</code>	link	<code>text</code>
<code>"text (title text)":url</code>	link with title text	<code>text</code>
<code>!url!</code>	image	<code></code>
<code>!url (alternate text)!</code>	image with alternate text	<code></code>
<code>!imgURL!:url</code>	linked image	<code></code>

Markup		HTML
!imgURL (alt text)!:url	linked image with alternate text	<code></code>

Examples

Simple link:

`<p>See here for the Xilize2 User Guide. </p>`

See [here](#) for the Xilize2 User Guide.

Link with title:

`<p>See here for the same link with title text. </p>`

See [here](#) for the same link with title text.


Simple image:

`<p>This is a common image on this site. </p>`

This  is a common image on this site.

Image with alt and title:

`<p>This is the same image with <code>alt</code> and <code>title</code> attributes set.</p>`

This is the same image  with alt and title attributes set.

Link-with-image


There are two ways to create a link using an image rather than text. The easiest way is to just add: URL to the end of the standard image markup like this:

!imageURL!: linkURL

Compare the two following examples and look for the difference in where the title attribute ends up.

The most efficient way:

This is an image with a link ``

This is an image with a link 

An alternative:

`<p>This is an image with a link </p>`

4.5 LISTS

There are three types of list formats:

- ordered (eg. numbered lists)
- unordered (bulleted lists)
- Definition lists.

All lists are BLOCK ELEMENTS which means that you do not have to nest them inside tags pairs such as `<p></p>`.

4.5.1 Unordered List

An unordered list is a list of items. The list items are marked with bullets (typically small black circles).

An unordered list starts with the `` tag. Each list item starts with the `` tag.

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

Here is how it looks in a browser:

- Coffee
- Milk

Inside a list item you can put paragraphs, line breaks, images, links, other lists, etc.

4.5.2 Ordered List

An ordered list is also a list of items. The list items are marked with numbers. An ordered list starts with the `` tag. Each list item starts with the `` tag.

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

Here is how it looks in a browser:

1. Coffee
2. Milk

4.5.3 Definition List

A definition list is not a list of items. This is a list of terms and explanation of the terms. A definition list starts with the `<dl>` tag. Each definition-list term starts with the `<dt>` tag. Each definition-list definition starts with the `<dd>` tag.

```
<dl>
  <dt>Coffee</dt>
  <dd>Black hot drink</dd>
  <dt>Milk</dt>
  <dd>White cold drink</dd>
</dl>
```

Here is how it looks in a browser:

Coffee

Black hot drink

Milk

White cold drink

Inside a definition-list definition (the `<dd>` tag) you can put paragraphs, line breaks, images, links, other lists, etc.

Some of the table tags are:

Tag	Description
<code></code>	Defines an ordered list
<code></code>	Defines an unordered list
<code></code>	Defines a list item
<code><dl></code>	Defines a definition list
<code><dt></code>	Defines a definition term

<code><dd></code>	Defines a definition description
<code><dir></code>	Deprecated. Use <code></code> instead
<code><menu></code>	Deprecated. Use <code></code> instead

4.6 TABLES

Tables are defined with the `<table>` tag. A table is divided into rows (with the `<tr>` tag), and each row is divided into data cells (with the `<td>` tag). The letters td stands for "table data," which is the content of a data cell. A data cell can contain text, images, lists, paragraphs, forms, horizontal rules, tables, etc.

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

How it looks in a browser:

Row 1, cell 1	row 1, cell 2
Row 2, cell 1	row 2, cell 2

4.6.1 Tables and the Border Attribute

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

To display a table with borders, you will have to use the border attribute:

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

4.6.2 Headings in a Table

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

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

How it looks in a browser:

Heading	Another Heading
row 1, cell 1	row 1, cell 2
row 2, cell 1	row 2, cell 2

4.6.3 Empty Cells in a Table

Table cells with no content are not displayed very well in most browsers.

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

How it looks in a browser:

Row 1, cell 1	row 1, cell 2
Row 2, cell 1	

Some of the table tags are:

Table Tags Tag	Description
<table>	Defines a table
<th>	Defines a table header
<tr>	Defines a table row
<td>	Defines a table cell
<caption>	Defines a table caption
<colgroup>	Defines groups of table columns
<col>	Defines the attribute values for one or more columns in a table
<thead>	Defines a table head
<tbody>	Defines a table body
<tfoot>	Defines a table foot

4.7 FORMS

A form is an area that can contain form elements. Form elements are elements that allow the user to enter information (like text fields, text area fields, drop-down menus, radio buttons, checkboxes, etc.) in a form. A form is defined with the <form> tag.

```
<form>
  <input>
  <input>
</form>
```

4.7.1 Input

The most used form tag is the <input> tag. The type of input is specified with the type attribute. The most commonly used input types are explained below.

4.7.1.1 Text Fields

Text fields are used when you want the user to type letters, numbers, etc. in a form.

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

How it looks in a browser:

First name:

Last name:

4.7.1.2 Radio Buttons

Radio Buttons are used when you want the user to select one of a limited number of choices.

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

How it looks in a browser:

☐ Male

☐ Female

4.7.1.3 Checkboxes

Checkboxes are used when you want the user to select one or more options of a limited number of choices.

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

How it looks in a browser:

I have a bike: ☐

I have a car: ☐


I have an airplane: ☐

4.7.1.4 The Form's Action Attribute and the Submit Button

When the user clicks on the "Submit" button, the content of the form is sent to another file. The form's action attribute defines the name of the file to send the content to. The file defined in the action attribute usually does something with the received input.

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

How it looks in a browser:

The image shows a browser window displaying a simple web form. On the left, the text "Username:" is followed by a rectangular text input field. To the right of the input field is a button labeled "Submit".

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

Some Form Examples:

- Form with input fields and a submit button
 - This example demonstrates how to add a form to a page. The form contains two input fields and a submit button.
- Form with checkboxes
 - This form contains three checkboxes, and a submit button.
- Form with radio buttons
 - This form contains two radio buttons, and a submit button.
- Send e-mail from a form
 - This example demonstrates how to send e-mail from a form.

4.8 FRAMES

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

The disadvantages of using frames are:

- The web developer must keep track of more HTML documents
- It is difficult to print the entire page

The Frameset Tag

- The <frameset> tag defines how to divide the window into frames
- Each frameset defines a set of rows **or** columns
- The values of the rows/columns indicate the amount of screen area each row/column will occupy

The Frame Tag

The <frame> tag defines what HTML document to put into each frame

In the example below we have a frameset with two columns. The first column is set to 25% of the width of the browser window. The second column is set to 75% of the width of the browser window. The HTML document "frame_a.htm" is put into the first column, and the HTML document "frame_b.htm" is put into the second column:

```
<frameset cols="25%,75%">
    <frame src="frame_a.htm">
    <frame src="frame_b.htm">
</frameset>
```

Some of the frame tags are:

Tag	Description
<frameset>	Defines a set of frames
<frame>	Defines a sub window (a frame)
<noframes>	Defines a no frame section for browsers that do not handle frames
<iframe>	Defines an inline sub window (frame)

4.9 HYPERTEXT LINKS

Permitted Context: %text

Content Model: %text, but no nested anchors

The anchor <A> element is used to define the start and/or destination of a hypertext link. In previous versions of HTML it provided the only means for defining destination anchors within documents, but you can now use any ID attribute as a destination anchor so that links can now be made to divisions, paragraphs and most other elements.

Example:

The `World Wide Web Organization` provides information on Web related standards, mailing lists and freeware tools.

The text between the start and end tag defines the label for the link. Selecting the link takes the reader to the document specified by the HREF attribute, in this case, the W3O home page. The label can include graphics defined with IMG elements.

For FIG elements, the anchor element serves a dual role. Non-graphical user agents interpret it as a conventional text-based hypertext link, while graphical user agents interpret the anchor's SHAPE attribute as a graphical hot zone.

4.9.1 Permitted Attributes

4.9.1.1 ID

An SGML identifier used as the target for hypertext links or for naming particular elements in associated style sheets. Identifiers are NAME tokens and must be unique within the scope of the current document. This attribute supercedes the "NAME" attribute, see below.

For example, the following paragraph is defined as an anchor named "potomac":

`<P ID="potomac">`

The Potomac river flows into Boston harbour, and played an important role in opening up the hinterland to early settlers...

Elsewhere, you can define a link to this paragraph, as follows:

`Boston`

is a historic city and a thriving center of commerce and higher education.

The reader can select the link labeled "Boston" to see further information on the Boston area.

4.9.1.2 LANG

This is one of the ISO standard language abbreviations, e.g. "en.uk" for the variation of English spoken in the United Kingdom. It can be used by parsers to select language specific choices for quotation marks, ligatures and hyphenation rules etc. The language attribute is composed from the two letter language code from ISO 639, optionally followed by a period and a two letter country code from ISO 3166.

4.9.1.3 CLASS

This space separated list of SGML NAME tokens and is used to subclass tag names. By convention, the class names are interpreted hierarchically, with the most general class on the left and the most specific on the right, where classes are separated by a period. The CLASS attribute is most commonly used to attach a different style to some element, but it is recommended that where practical class names should be picked on the basis of the element's semantics, as this will permit other uses, such as restricting search through documents by matching on element class names. The conventions for choosing class names are outside the scope of this specification.

4.9.1.4 HREF

The HREF attribute implies that the anchor acts as the start of a hypertext link. The destination is designated by the value of the HREF attribute, which is expressed in the Universal Resource Identifier (URI) notation.

4.9.1.5 MD

Specifies a message digest or cryptographic checksum for the linked document designated by the HREF attribute. It is used when you want to be sure that a linked object is indeed the same one that the author intended, and hasn't been modified in any way. For instance, MD="md5:jV2OfH+nnXHU8bnkPAad/mSQITDZ", which specifies an MD5 checksum encoded as a base64 character string. The MD attribute is generally allowed for all elements which support URI based links.

4.9.1.6 NAME

This attribute is used to define a named anchor for use as the destination of hypertext links. For example, the following defines an anchor than can be used as the destination of a jump into a description of the Boston area.

The Potomac river flows into Boston harbour.

4.9.1.7 SHAPE

This attribute is used within figures to define shaped hotzones for graphical hypertext links. Full details of how to use this feature will be given with the description of the figure element. The attribute value is a string taking one of the following forms:

"default"

Used to define a default link for the figure background.

"circle x, y, r"

Where x and y define the center and r specifies the radius.

"rect x, y, w, h"

Where x, y define the upper left corner and w, h define the width and height respectively

"polygon x1, y1, x2, y2, ..."

Given n pairs of x, y coordinates, the polygon is closed by a line linking the n'th point to the first. Intersecting polygons use the non-zero winding number rule to determine if a point lies inside the polygon.

If a pointer event occurs in a region where two or more shapes overlap, the distance from the point to the center of gravity of each of the overlapping shapes is computed and the closest one chosen. This feature is useful when you want lots of closely spaced hot zones, for example over points on a map, as it allows you to use simple shapes without worrying about overlaps.

4.9.1.8 TITLE

This is informational only and describes the object specified with the HREF attribute. It can be used for object types that don't possess titles, such as graphics, plain text and Gopher menus.

4.9.1.9 REL

Used to describe the relationship of the linked object specified with the HREF attribute. The set of relationship names is not part of this specification, although "Path" and "Node" are reserved for future use with hypertext paths or guided tours. The REL attribute can be used to support search for links serving particular relationships.

4.9.1.10 REV

This defines a reverse relationship. A link from document A to document B with *REV=relation* expresses the same relationship as a link from B to A with *REL=relation*. *REV=made* is sometimes used to identify the document author, either the author's email address with a *mailto* URI, or a link to the author's home page. Tables of contents can use anchors with *REV="ToC"* to allow software to insert page numbers when printing hypertext documents. The plain text version of this specification was generated in this way!

4.10 HTML VERSUS XHTML

The argument about whether to use HTML4.01 or XHTML1 is one that comes up time and time again. Not so long ago, most people were advising the use of XHTML1 almost without question, in the belief that it's little more than a newer implementation of HTML and therefore somehow automatically the 'better' option (specifically, XHTML1 is a reformulation of HTML4 as an application of XML). However, many people who once recommended using XHTML1 have since changed their minds on the topic, including some SitePoint authors who made very strong arguments as to why examples in their books should be presented in HTML 4.01 rather than XHTML 1. Others have decided to move on to using HTML5 instead, even though the W3C has not finalized this particular language's specification. That said, it is clear now that HTML5 is where the Web is headed (XHTML2, the proposed successor to XHTML 1 & XHTML 1.1, has effectively been canned with the working group being disbanded at the end of 2009). Until such time as the HTML5 specification is solidified and has strong support across the major browsers, the safest option is to use HTML4.01 or XHTML1 (which from this point on we'll refer to simply as HTML and XHTML without the version numbers).

It seems that we need to clarify what HTML and XHTML are, what their differences are, and why one or the other should be used.

The first thing you should realize is that using HTML is not wrong as long as you specify that you're using HTML with the appropriate doctype, and the HTML you use is valid for that doctype. If you want to use HTML 4.01, no one can stop you! Ignore anyone who tells you that XHTML is the only way to go, and that using HTML 4.01 is somehow backwards. That said, you should be aware of the differences between HTML and XHTML, as these may affect your choice of markup.

4.10.1 Syntactic difference between HTML and XHTML

The following list details the main differences between XHTML and HTML. Most of them are related to syntax differences, although there are some less obvious variations that you may not be aware of:

- XHTML is choosier than HTML—there are some elements that absolutely must appear in the XHTML markup, but which may be omitted if you're using HTML 4 and earlier versions. These

elements include the `html`, `head`, and `body` elements (although why you'd want to omit any of them is a mystery to me). In addition, every element you use in XHTML must have both an opening and closing tag (for example, you'd write `<p>This is a paragraph</p>` in XHTML, but `<p>This is all you need` in HTML, as no end tag is required).

- For empty elements—those that hold no content but refer to a resource of some kind, such as an `img`, `link`, or meta element—the tag must have a trailing closing slash, like so: ``. Evidently, this makes XHTML a little more verbose than HTML, but not to the extent that it has an adverse effect on the page weight.
- XHTML allows us to indicate any element as being empty—for example, an empty paragraph can be expressed as `<p/>`—but this isn't valid when the page is served as `text/html`. To that end, you should restrict your use of this syntax to elements that are defined to be empty in the HTML specifications.
- In XHTML, all tags must be written in lowercase. In HTML, you can use capital letters for elements, lowercase letters for attributes, or whatever convention you like!
- All attributes in XHTML must be contained in quotes (single or double, but usually double), hence `<input type=submit name=cmdGo/>` would be valid in HTML 4.01, but would be invalid in XHTML. To be valid, it would need to be `<input type="submit" name="cmdGo"/>`.
- In XHTML, all attributes must be expressed in attribute-name and attribute-value pairings with quote marks surrounding the attribute value part, like so: `class="fuzzy"`.
- In HTML, some elements have attributes that do not *appear* to require a value—for example, the `checked` attribute for checkbox input elements. I stressed the word “appear” because technically it's the attribute *name* that's omitted, not the value. These are known as Boolean attributes, and in HTML you could specify that a checkbox should be checked simply by typing `<input type="checkbox" name="chkNewsletter" checked>`. In XHTML, though, you must supply both an attribute and value, which results in seemingly needless repetition: `<input type="checkbox" name="chkNewsletter" checked="checked">`.
- In XHTML, the opening `<html>` tag requires an `xmlns` attribute (XML Namespace) as follows: `<html xmlns="http://www.w3.org/1999/xhtml">`. However, strangely, if you omit it, the W3C validator doesn't protest as it should.
- XHTML requires certain characters to appear as named entities. For example, you can't use the `&` character: it must be expressed using an HTML entity `"&";`.

- In XHTML, languages in the document must be expressed using the `xml:lang` attribute instead of `lang`.
- A MIME type must be declared appropriately in the HTTP headers as `application/xhtml+xml` (this is the best option), `application/xml` (acceptable), or `text/xml` (which isn't recommended). The MIME type is set as a configuration option on the server, and is usually Apache or IIS.
- DTDs don't support the validation of mixed namespace documents very well.
- If you use XHTML and set the proper MIME type (see the section below called Serving), you'll encounter a small snag: Internet Explorer. At the time of writing, this browser—which still holds the lion's share of the market—is the only one of the browsers tested for this reference that can't handle a document set with a MIME type of `application/xhtml+xml`. When IE encounters a page that contains this HTTP header, it doesn't render the page on screen, but instead prompts the user to download or save the document.
- When you're using XHTML, text encoding should be set within the XML declaration, not in the HTTP headers (although doing the latter is still allowed). A full tutorial on the thorny issues surrounding the ways of setting character encodings is available on the W3C site.

4.11 SUMMARY

In this unit, we have learnt the different types of Markup Language and basic text markup language. Here we have learnt how to give links and use images in the pages. Usage of Lists, tables, forms and frames are been discussed here how exactly these topics are being used. The syntax of the Lists, tables, forms and frames have been discussed in detail. Links from one page to another page have been discussed using the HyperText Links. The syntactic difference between HTML and XHTML have been discussed in detail.

4.12 KEYWORDS

HTML – HyperText Markup Language,

XHTML – Extensible HyperText Markup Language,

W3C markup, XML, URL.

4.13 UNIT END EXERCISE AND ANSWERS

- 1) What is Markup Language? Explain the different types of Markup Language in detail.
- 2) Explain the basic text markup in detail.
- 3) How are Links and images, lists, tables, forms, and frames used in markup language? Explain it with syntax.
- 4) What are Hypertext Links? How do you use link one page to another? Explain it with syntax and example.
- 5) Differentiate HTML versus XHTML?

4.14 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

Module 2

UNIT 5: STYLE SHEET

Structure:

- 5.0 Objectives
- 5.1 Introduction
- 5.2 Introduction to Style Sheet
- 5.3 Levels of Style sheet and Style specification formats
- 5.4 Selectors
- 5.5 Summary
- 5.6 Keywords
- 5.7 Unit-end exercises and answers
- 5.8 Suggested readings

5.0 OBJECTIVES

At the end of this unit you will be able to know:

- Style sheet working and usage
- Levels in Style sheet
- Style sheet specification format and syntax
- Selectors

5.1 INTRODUCTION

A web **style sheet** is a form of separation of presentation and content for web design in which the markup (i.e., HTML or XHTML) of a webpage contains the page's semantic content and structure, but does not define its visual layout (style). Instead, the style is defined in an external style sheet file using a style sheet language such as CSS or XSLT. This design approach is identified as a "separation" because it largely supersedes the antecedent methodology in which a page's markup defined both style and structure.

The philosophy underlying this methodology is a specific case of separation of concerns.

5.1.1 Benefits

Separation of style and content has many benefits, but has only become practical in recent years due to improvements in popular web browser's CSS implementations.

Speed: Overall, user's experience of a site utilizing style sheets will generally be quicker than sites that don't use the technology. 'Overall' as the first page will probably load more slowly because the style sheet AND the content will need to be transferred. Subsequent pages will load faster because no style information will need to be downloaded – the CSS file will already be in the browser's cache.

Maintainability: Holding all the presentation styles in one file significantly reduces maintenance time and reduces the chance of human errors, thereby improving presentation consistency. For example, the font color associated with a type of text element may be specified - and therefore easily modified - throughout an entire website simply by changing one short string of characters in a single file. The alternate approach, using styles embedded in each individual page, would require a cumbersome, time consuming, and error-prone edit of every file.

Accessibility: Sites that use CSS with either XHTML or HTML are easier to tweak so that they appear extremely similar in different browsers (Explorer, Mozilla, Opera, Safari, etc.).

Sites using CSS "degrade gracefully" in browsers unable to display graphical content, such as Lynx, or those so very old that they cannot use CSS. Browsers ignore CSS that they do not understand, such as CSS 3 statements. This enables a wide variety of user agents to be able to access the content of a site even if they cannot render the style sheet or are not designed with graphical capability in mind. For example, a browser using a refreshable for output could disregard layout information entirely, and the user would still have access to all page content.

Customization: If a page's layout information is all stored externally, a user can decide to disable the layout information entirely, leaving the site's bare content still in a readable form. Site authors may also offer multiple style sheets, which can be used to completely change the appearance of the site without altering any of its content.

Most modern web browsers also allow the user to define their own style sheet, which can include rules that override the author's layout rules. This allows users, for example, to bold every hyperlink on every page they visit.

Consistency: Because the semantic file contains only the meanings an author intends to convey, the styling of the various elements of the document's content is very consistent. For example, headings, emphasized text, lists and mathematical expressions all receive consistently applied style properties from the external style sheet. Authors need not concern themselves with the style properties at the time of composition. These presentational details can be deferred until the moment of presentation.

Portability: The deferment of presentational details until the time of presentation means that a document can be easily re-purposed for an entirely different presentation medium with merely the application of a new style sheet already prepared for the new medium and consistent with elemental or structural vocabulary of the semantic document. A carefully authored document for a web page can easily be printed to a hard-bound volume complete with headers and footers, page numbers and a generated table of contents simply by applying a new style sheet.

5.2 LEVELS OF STYLE SHEETS AND STYLE SPECIFICATION FORMATS

There are three levels of style sheets

1. Inline
2. Document-level style
3. Class & ID Styles
4. Cascading Style Sheets
5. External Style sheets

5.2.1 Inline Style Sheets: An Inline style sheet is a term that refers to style sheet information being applied to the *current* element. By this, I mean that instead of defining the style once, then applying the style against all instances of an element (say the `<p>` tag), you only apply the style to the instance you want the style to apply to. Actually, it's not really a style sheet as such, so a more accurate term would be *inline styles*.

Syntax: *In the code below, we've added CSS by using inline styles. You can tell because we've added the styles to the `<p>` tag using the style attribute.*

`<p style="font-size:large;line-height:1.2em;color: #ff9900">This text has been styled using inline style sheets.</p>`

Using inline styles is not as powerful as embedded style sheets, imported style sheets, or linking to an external style sheet. To specify style information for more than one element, you should embed a style sheet in the header of the HTML document. Better yet, you should define styles in an external style sheet, and then link to that style sheet. You then can apply the style across multiple HTML documents.

5.2.2 Document-level styles: Document-level styles are specified within a pair of <STYLE> ... </STYLE> DHTML tags that must be placed between the <HEAD> ... </HEAD> section tags of a document. The global document formatting for a desired tag is indicated by listing the HTML tag, (without the < > brackets), following by a set of braces, { }, containing the CSS style name: value pairs to be applied to all tagged text in the document, (the colon separator is required). The following example specifies global styles for heading one tags and document paragraphs:

```
<HEAD>
<STYLE>
<!-- comment for old browsers not supporting CSS
H1 {color: blue; font-family:"Helvetica"; font-style:italic; font-size:18pt;}
P  {color: black; font-family:"times";   font-style:normal; font-size:12pt;}
-->
</STYLE>
</H1>
```

All text in the document surrounded with H1 or P tags would be formatted according to these rules. The author needs only change the styles to change all of the corresponding document text.

5.2.3 Class & ID Styles: Quite often a writer needs to be able to define their own combination of formatting and layout to apply at particular locations in their document. For example: a quoted paragraph, special lists, alternating table rows/columns, etc. Most word processors allow a user to do this by providing the capability for a user to define their own "named styles". The CSS specification also provides this functionality. To define a class style for an indented paragraph with blue text, one would first specify in the STYLE section a list of standard style definitions and properties. The list must be preceded by a period immediately followed by a space and the

desired name of the class style, (.QuotePara) and surrounded by curly braces {}. See the example.

```
CSS Class Style
<HEAD>
<STYLE>
.QuotePara
{
margin-left: 100pt;
margin-right: 100pt;
color: blue
}
</STYLE>
```

In order to apply the class style to a particular paragraph, the HTML CLASS attribute is used with any HTML tag. For example to apply it to this particular paragraph, the P tag was specified:

```
<P CLASS="QuotePara">
```

at the beginning of the paragraph.

ID styles operate essentially in the same manner as class styles. They are specified using a pound sign character instead of a period. The previous .QuotePara class would be specified as an ID style as shown at the right. It is applied to a HTML tag using the ID attribute instead of the class attribute. In this paragraph it was specified as:

```
<P ID=QPara>
```

```
CSS ID Style
<HEAD>
<STYLE>
#QPara
{
margin-left: 100pt;
margin-right: 100pt;
color: blue
}
</STYLE>
```

5.2.4 Cascading style sheets: *Cascading* style sheet languages such as CSS allow style information from several sources to be blended together. However, not all style sheet languages support cascading. To define a cascade, authors specify a sequence of LINK and/or STYLE elements. The style information is cascaded in the order the elements appear in the HEAD.

In the following example, we specify two alternate style sheets named "compact". If the user selects the "compact" style, the user agent must apply both external style sheets, as well as the persistent "common.css" style sheet. If the user selects the "big print" style, only the alternate style sheet "bigprint.css" and the persistent "common.css" will be applied.

```
<LINK rel="alternate stylesheet" title="compact" href="small-base.css" type="text/css">
<LINK rel="alternate stylesheet" title="compact" href="small-extras.css" type="text/css">
<LINK rel="alternate stylesheet" title="big print" href="bigprint.css" type="text/css">
<LINK rel="stylesheet" href="common.css" type="text/css">
```

Here is a cascade example that involves both the LINK and STYLE elements.

```
<LINK rel="stylesheet" href="corporate.css" type="text/css">
<LINK rel="stylesheet" href="techreport.css" type="text/css">
<STYLE type="text/css">
  p.special { color: rgb(230, 100, 180) }
</STYLE>
```

Media-dependent cascades: A cascade may include style sheets applicable to different media. Both LINK and STYLE may be used with the media attribute. The user agent is then responsible for filtering out those style sheets that do not apply to the current medium.

In the following example, we define a cascade where the "corporate" style sheet is provided in several versions: one suited to printing, one for screen use and one for speech-based browsers (useful, say, when reading email in the car). The "techreport" stylesheet applies to all media. The color rule defined by the STYLE element is used for print and screen but not for aural rendering.

```
<LINK rel="stylesheet" media="aural" href="corporate-aural.css" type="text/css">
```

```
<LINK rel="stylesheet" media="screen" href="corporate-screen.css" type="text/css">
<LINK rel="stylesheet" media="print" href="corporate-print.css" type="text/css">
<LINK rel="stylesheet" href="techreport.css" type="text/css">
<STYLE media="screen, print" type="text/css">
  p.special { color: rgb(230, 100, 180) }
</STYLE>
```

Inheritance and cascading: When the user agent wants to render a document, it needs to find values for style properties, e.g. the font family, font style, size, line height, text color and so on. The exact mechanism depends on the style sheet language, but the following description is generally applicable:

The cascading mechanism is used when a number of style rules all apply directly to an element. The mechanism allows the user agent to sort the rules by specificity, to determine which rule to apply. If no rule can be found, the next step depends on whether the style property can be inherited or not. Not all properties can be inherited. For these properties the style sheet language provides default values for use when there are no explicit rules for a particular element.

If the property can be inherited, the user agent examines the immediately enclosing element to see if a rule applies to that. This process continues until an applicable rule is found. This mechanism allows style sheets to be specified compactly. For instance, authors may specify the font family for all elements within the BODY by a single rule that applies to the BODY element.

5.2.5 External style sheets: Authors may separate style sheets from HTML documents. This offers several benefits:

- Authors and Web site managers may share style sheets across a number of documents (and sites).
- Authors may change the style sheet without requiring modifications to the document.
- User agents may load style sheets selectively (based on media descriptions).

Preferred and alternate style sheets: HTML allows authors to associate any number of external style sheets with a document. The style sheet language defines how multiple external style sheets interact (for example, the CSS "cascade" rules).

Users may select their favorite among these depending on their preferences. For instance, an author may specify one style sheet designed for small screens and another for users with weak vision (e.g., large fonts). User agents should allow users to select from alternate style sheets.

The author may specify that one of the alternates is a preferred style sheet. User agents should apply the author's preferred style sheet unless the user has selected a different alternate.

When a user selects a named style, the user agent must apply all style sheets with that name. User agents must not apply alternate style sheets with a different style name. The section on [specifying external style sheets](#) explains how to name a group of style sheets.

User agents must respect [media descriptors](#) when applying any style sheet.

User agents should also allow users to disable the author's style sheets entirely, in which case the user agent must not apply any persistent or alternate style sheets.

Specifying external style sheets: Authors specify external style sheets with the following attributes of the [LINK](#) element:

- Set the value of [href](#) to the location of the style sheet file. The value of [href](#) is a [URI](#).
- Set the value of the [type](#) attribute to indicate the language of the linked (style sheet) resource. This allows the user agent to avoid downloading a style sheet for an unsupported style sheet language.
- Specify that the style sheet is persistent, preferred, or alternate:
 - To make a style sheet persistent, set the [rel](#) attribute to "stylesheet" and don't set the [title](#) attribute.
 - To make a style sheet preferred, set the [rel](#) attribute to "stylesheet" and name the style sheet with the [title](#) attribute.
 - To specify an alternate style sheet, set the [rel](#) attribute to "alternate stylesheet" and name the style sheet with the [title](#) attribute.

User agents should provide a means for users to view and pick from the list of alternate styles.

The value of the [title](#) attribute is recommended as the name of each choice.

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

Setting the [title](#) attribute makes this the author's preferred style sheet:

```
<LINK href="mystyle.css" title="compact" rel="stylesheet" type="text/css">
```

Adding the keyword "alternate" to the [rel](#) attribute makes it an alternate style sheet:

```
<LINK href="mystyle.css" title="Medium" rel="alternate stylesheet" type="text/css">
```

For more information on external style sheets, please consult the section on [links and external style sheets](#).

Authors may also use the [META](#) element to set the document's preferred style sheet. For example, to set the preferred style sheet to "compact" (see the preceding example), authors may include the following line in the [HEAD](#):

```
<META http-equiv="Default-Style" content="compact">
```

The preferred style sheet may also be specified with HTTP headers. The above [META](#) declaration is equivalent to the HTTP header:

```
Default-Style: "compact"
```

If two or more [META](#) declarations or HTTP headers specify the preferred style sheet, the last one takes precedence. HTTP headers are considered to occur earlier than the document [HEAD](#) for this purpose.

If two or more [LINK](#) elements specify a preferred style sheet, the first one takes precedence.

Preferred style sheets specified with [META](#) or HTTP headers have precedence over those specified with the [LINK](#) element.

5.3 SELECTOR

Selectors are used to declare which elements a style applies to, a kind of match expression. Selectors may apply to all elements of a specific type, or only those elements which match a certain attribute; elements may be matched depending on how they are placed relative to each other in the markup code, or on how they are nested within the document object model.

A *simple selector* is either a [type selector](#) or [universal selector](#) followed immediately by zero or more [attribute selectors](#), [ID selectors](#), or [pseudo-classes](#), in any order. The simple selector matches if all of its components match.

A *selector* is a chain of one or more simple selectors separated by combinators. *Combinators* are: whitespace, ">", and "+". Whitespace may appear between a combinators and the simple selectors around it.

The elements of the document tree that match a selector are called *subjects* of the selector. A selector consisting of a single simple selector matches any element satisfying its requirements. Prepending a simple selector and combinators to a chain imposes additional matching constraints, so the subjects of a selector are always a subset of the elements matching the rightmost simple selector.

One pseudo-element may be appended to the last simple selector in a chain, in which case the style information applies to a subpart of each subject.

Grouping

When several selectors share the same declarations, they may be grouped into a comma-separated list.

In this example, we condense three rules with identical declarations into one. Thus,

H1 { font-family: sans-serif }

H2 { font-family: sans-serif }

H3 {font-family: sans-serif}

is equivalent to:

H1, H2, H3 {font-family: sans-serif}

CSS offers other "shorthand" mechanisms as well, including multiple declarations and shorthand properties.

5.3.1 Universal selector

The universal selector, written "*", matches the name of any element type. It matches any single element in the document tree.

If the universal selector is not the only component of a simple selector, the "*" may be omitted.

For example:

- *[LANG=fr] and [LANG=fr] are equivalent.
- *.warning and .warning are equivalent.
- *#myid and #myid are equivalent.

5.3.2 Type selectors

A *type selector* matches the name of a document language element type. A type selector matches every instance of the element type in the document tree.

The following rule matches all H1 elements in the document tree:

H1 { font-family: sans-serif }

5.3.3 Descendant selectors

At times, authors may want selectors to match an element that is the descendant of another element in the document tree (e.g., "Match those EM elements that are contained by an H1 element"). Descendant selectors express such a relationship in a pattern. A descendant selector is made up of two or more selectors separated by whitespace. A descendant selector of

the form "A B" matches when an element B is an arbitrary descendant of some ancestor element A.

For example, consider the following rules:

```
H1 { color: red }
```

```
EM { color: red }
```

Although the intention of these rules is to add emphasis to text by changing its color, the effect will be lost in a case such as:

```
<H1>This headline is <EM>very</EM> important</H1>
```

We address this case by supplementing the previous rules with a rule that sets the text color to blue whenever an EM occurs anywhere within an H1:

```
H1 {color: red}
```

```
EM {color: red}
```

```
H1 EM {color: blue}
```

The third rule will match the EM in the following fragment:

```
<H1>This <SPAN class="myclass">headline is <EM>very</EM> important</SPAN></H1>
```

The following selector:

```
DIV * P
```

matches a P element that is a grandchild or later descendant of a DIV element. Note the whitespace on either side of the "*".

The selector in the following rule, which combines descendant and attribute selectors, matches any element that (1) has the "href" attribute set and (2) is inside a P that is itself inside a DIV:

```
DIV P *[href]
```

5.3.4 Child selectors

A *child selector* matches when an element is the child of some element. A child selector is made up of two or more selectors separated by ">".

The following rule sets the style of all P elements that are children of BODY:

```
BODY > P { line-height: 5.3 }
```

The following example combines descendant selectors and child selectors:

```
DIV OL>LI P
```

It matches a P element that is a descendant of an LI; the LI element must be the child of an OL element; the OL element must be a descendant of a DIV. Notice that the optional whitespace around the ">" combinators has been left out.

For information on selecting the first child of an element, please see the section on the :first-child pseudo-class below.

5.3.5 Adjacent sibling selectors

Adjacent sibling selectors have the following syntax: E1 + E2, where E2 is the subject of the selector. The selector matches if E1 and E2 share the same parent in the document tree and E1 immediately precedes E2.

In some contexts, adjacent elements generate formatting objects whose presentation is handled automatically (e.g., collapsing vertical margins between adjacent boxes). The "+" selector allows authors to specify additional style to adjacent elements.

Thus, the following rule states that when a P element immediately follows a MATH element, it should not be indented:

MATH + P {text-indent: 0}

The next example reduces the vertical space separating an H1 and an H2 that immediately follows it:

H1 + H2 {margin-top: -5mm}

The following rule is similar to the one in the previous example, except that it adds an attribute selector. Thus, special formatting only occurs when H1 has class="opener":

H1.opener + H2 {margin-top: -5mm}

5.3.6 Attribute selectors

CSS2 allows authors to specify rules that match attributes defined in the source document.

5.3.6.1 Matching attributes and attribute values

Attribute selectors may match in four ways:

[att]

Match when the element sets the "att" attribute, whatever the value of the attribute.

[att=val]

Match when the element's "att" attribute value is exactly "val".

[att~=val]

Match when the element's "att" attribute value is a space-separated list of "words", one of which is exactly "val". If this selector is used, the words in the value must not contain spaces (since they are separated by spaces).

[att=val]

Match when the element's "att" attribute value is a hyphen-separated list of "words", beginning with "val". The match always starts at the beginning of the attribute value. This is primarily intended to allow language subcode matches (e.g., the "lang" attribute in HTML) as described in RFC 1766 ([RFC1766]).

Attribute values must be identifiers or strings. The case-sensitivity of attribute names and values in selectors depends on the document language.

For example, the following attribute selector matches all H1 elements that specify the "title" attribute, whatever its value:

```
H1 [title] {color: blue;}
```

In the following example, the selector matches all SPAN elements whose "class" attribute has exactly the value "example":

```
SPAN [class=example] {color: blue;}
```

Multiple attribute selectors can be used to refer to several attributes of an element, or even several times the same attribute.

Here, the selector matches all SPAN elements whose "hello" attribute has exactly the value "Cleveland" and whose "goodbye" attribute has exactly the value "Columbus":

```
SPAN [hello="Cleveland"] [goodbye="Columbus"] {color: blue;}
```

The following selectors illustrate the differences between "=" and "~=". The first selector will match, for example, the value "copyright copyleft copyeditor" for the "rel" attribute. The second selector will only match when the "href" attribute has the value "http://www.w3.org/".

```
A[rel~="copyright"]
```

```
A[href="http://www.w3.org/"]
```

The following rule hides all elements for which the value of the "lang" attribute is "fr" (i.e., the language is French).

```
*[LANG=fr] {display: none}
```

The following rule will match for values of the "lang" attribute that begin with "en", including "en", "en-US", and "en-cockney":

`*[LANG="en"] {color: red}`

Similarly, the following aural style sheet rules allow a script to be read aloud in different voices for each role:

`DIALOGUE [character=romeo]`

`{voice-family: "Lawrence Olivier", charles, male }`

`DIALOGUE [character=juliet]`

`{voice-family: "Vivien Leigh", victoria, female }`

5.3.6.2 Default attribute values in DTDs

Matching takes place on attribute values in the document tree. For document languages other than HTML, default attribute values may be defined in a DTD or elsewhere. Style sheets should be designed so that they work even if the default values are not included in the document tree.

For example, consider an element `EXAMPLE` with an attribute `"notation"` that has a default value of `"decimal"`. The DTD fragment might be

`<! ATTLIST EXAMPLE notation (decimal,octal) "decimal">`

If the style sheet contains the rules

`EXAMPLE [notation=decimal] {/*... default property settings ...*/}`

`EXAMPLE [notation=octal] {/*... other settings...*/}`

then to catch the cases where this attribute is set by default, and not explicitly, the following rule might be added:

`EXAMPLE {/*... default property settings ...*/}`

Because this selector is less specific than an attribute selector, it will only be used for the default case. Care has to be taken that all other attribute values that don't get the same style as the default are explicitly covered.

5.3.6.3 Class selectors

For style sheets used with HTML, authors may use the dot (.) notation as an alternative to the `"~="` notation when matching on the `"class"` attribute. Thus, for HTML, `"DIV.value"` and `"DIV [class~=value]"` have the same meaning. The attribute value must immediately follow the `"."`.

For example, we can assign style information to all elements with `class~="pastoral"` as follows:

`*.pastoral {color: green} /* all elements with class~=pastoral */`

or just

```
.pastoral {color: green} /* all elements with class~=pastoral */
```

The following assigns style only to H1 elements with class~="pastoral":

```
H1.pastoral {color: green} /* H1 elements with class~=pastoral */
```

Given these rules, the first H1 instance below would not have green text, while the second would:

```
<H1>Not green</H1>
```

```
<H1 class="pastoral">Very green</H1>
```

To match a subset of "class" values, each value must be preceded by a ".", in any order.

For example, the following rule matches any P element whose "class" attribute has been assigned a list of space-separated values that includes "pastoral" and "marine":

```
P.pastoral.marine {color: green}
```

This rule matches when class="pastoral blue aqua marine" but does not match for class="pastoral blue".

5.3.7 ID selectors

Document languages may contain attributes that are declared to be of type ID. What makes attributes of type ID special is that no two such attributes can have the same value; whatever the document language, an ID attribute can be used to uniquely identify its element. In HTML all ID attributes are named "id"; XML applications may name ID attributes differently, but the same restriction applies.

The ID attribute of a document language allows authors to assign an identifier to one element instance in the document tree. CSS ID selectors match an element instance based on its identifier. A CSS ID selector contains a "#" immediately followed by the ID value.

The following ID selector matches the H1 element whose ID attribute has the value "chapter1":

```
H1#chapter1 {text-align: center}
```

In the following example, the style rule matches the element that has the ID value "z98y". The rule will thus match for the P element:

```
<HEAD>
```

```
<TITLE>Match P</TITLE>
```

```
<STYLE type="text/css">
```

```
*#z98y { letter-spacing: 0.3em }
```



```
</STYLE>
</HEAD>
<BODY>
  <P id=z98y>Wide text</P>
</BODY>
```

In the next example, however, the style rule will only match an H1 element that has an ID value of "z98y". The rule will not match the P element in this example:

```
<HEAD>
  <TITLE>Match H1 only</TITLE>
  <STYLE type="text/css">
    H1#z98y {letter-spacing: 0.5em}
  </STYLE>
</HEAD>
<BODY>
  <P id=z98y>Wide text</P>
</BODY>
```

ID selectors have a higher precedence than attribute selectors. For example, in HTML, the selector #p123 is more specific than [ID=p123] in terms of the cascade.

5.4 SUMMARY

A fundamental feature of CSS is that more than one style sheet can influence the presentation of a document. This feature is known as *cascading* because the different style sheets are thought of as coming in a series. Cascading is a fundamental feature of CSS, because we realized that any single document could very likely end up with style sheets from multiple sources: the browser, the designer, and possibly the user.

In the last set of examples you saw that the text color of the links turned blue without that being specified in the style sheet. Also, the browser knew how to format block quote and h1 elements without being told so explicitly. Everything that the browser knows about formatting is stored in the browser's *default style sheet* and is merged with author and user style sheets when the document is displayed.

5.5 KEYWORDS

CSS – Cascading Style Sheet, Inline, Inheritance, META elements, Combinators, DIV tag and SPAN tag

5.6 UNIT END EXERCISE AND ANSWER

- 1) What is Style Sheet? Explain.
 - 2) Explain the levels of style sheet in detail.
 - 3) Explain the style specification formats
 - 4) What are Selectors? Explain in detail.
-

5.7 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

UNIT 6:

Structure:

- 6.0 Objectives
- 6.1 Font Properties
- 6.2 List Properties
- 6.3 Color
- 6.4 Alignment of Text
- 6.5 The Box Model
- 6.6 Background Image
- 6.7 Summary
- 6.8 Keywords
- 6.9 Unit-end exercises and answers
- 6.10 Suggested readings

6.0 OBJECTIVES

At the end of this unit you will be able to know:

- Usage of Font and List properties
- Color scheme in Websites or Web pages
- How to align a text
- The Box Model
- How to insert, resize Background Image

6.1 FONT PROPERTIES

6.1.1 Font: The font property may be used as shorthand for the various font properties, as well as the line height. For example,

P {font: italic bold 12pt/14pt Times, serif}

specifies paragraphs with a bold and italic Times or serif font with a size of 12 points and a line height of 14 points.

Syntax	font: <value>
Possible Values	[<font-style> <font-variant> <font-weight>]? <font-size>

	[/ <line-height>]? <font-family>
Initial Value	Not defined
Applies to	All elements
Inherited	Yes

6.1.2 Font Family: Font family may be assigned by a specific font name or a generic font family. Obviously, defining a specific font will not be as likely to match as a generic font family. Multiple family assignments can be made, and if a specific font assignment is made it should be followed by a generic family name in case the first choice is not present.

A sample font-family declaration might look like this:

P {font-family: "New Century Schoolbook", Times, serif}

Notice that the first two assignments are specific type faces: New Century Schoolbook and Times. However, since both of them are serif fonts, the generic font family is listed as a backup in case the system does not have either of these but has another serif font which meets the qualifications.

Any font name containing whitespace must be quoted, with either single or double quotes.

The font family may also be given with the font property.

Syntax	font-family: [[<family-name> <generic-family>],]* [<family-name> <generic-family>]
Possible Values	<family-name> <ul style="list-style-type: none"> Any font family name may be used <generic-family> Serif(e.g., Times) Sans-serif(e.g., Arial or Helvetica) Cursive(e.g., Zapf-Chancery) Fantasy(e.g., Western) Monospace(e.g., Courier)
Initial Value	Determined by browser
Applies to	All elements
Inherited	Yes

6.1.3 Font Style: The font-style property defines that the font be displayed in one of three ways: normal, italic or oblique (slanted). A sample style sheet with font-style declarations might look like this:

H1 {font-style: oblique}

P {font-style: normal}

Syntax	font-style: <value>
Possible Values	normal italic oblique
Initial value	normal
Applies to	All elements
Inherited	Yes

6.1.4 Font Variant: The font-variant property determines if the font is to display in normal or SMALL CAPS. Small-caps are displayed when all the letters of the words are in capital with uppercase characters slightly larger than lowercase. Later versions of CSS may support additional variants such as condensed expanded, small-caps numerals or other custom variants. An example of a font-variant assignment would be:

SPAN {font-variant: small-caps}

Syntax	font-variant: <value>
Possible Values	normal SMALL CAPS
Initial Value	normal
Applies to	All elements
Inherited	Yes

6.1.5 Font Weight: The font-weight property is used to specify the weight of the font. The bolder and lighter values are relative to the inherited font weight, while the other values are absolute font weights.

- 500 may be switched with 400, and vice-versa
- 100-300 may be assigned to the next lighter weight, if any, or the next darker weight otherwise
- 600-900 may be assigned to the next darker weight, if any, or the next lighter weight otherwise

Some example font-weight assignments would be:

```
H1 {font-weight: 800}
P {font-weight: normal}
```

Syntax	font-weight: <value>
Possible Values	normal bold bolder lighter 100 200 300 400 500 600 700 800 900
Initial Value	Normal
Applies to	All elements
Inherited	Yes

6.1.6 Font Size: The font-size property is used to modify the size of the displayed font. Absolute lengths should be used sparingly due to their weakness in adapting to different browsing environments. Fonts with absolute lengths can very easily be too small too large for a user.

Some example size assignments would be:

```
H1 {font-size: large}
P {font-size: 12pt}
LI {font-size: 90% }
STRONG {font-size: larger}
```

Authors should be aware that Microsoft Internet Explorer 3.x incorrectly treats em and ex units as pixels, which can easily make text using these units unreadable. The browser also incorrectly applies percentage values relative to its default font size for the selector, rather than relative to the parent element's font size. This make rules like

```
H1 {font-size: 200% }
```

dangerous in that the size will be twice IE's default font size for level-one headings, rather than twice the parents element's font size. In this case, BODY would most likely be the parent element, and it would likely define a medium font size, whereas the default level-one heading font size imposed by IE would probably be considered xx-large.

Given these bugs, authors should take care in using percentage values for font-size, and should probably avoid em and ex units for this property.

Syntax	font-size: <absolute-size> <relative-size> <length> <percentage>
Possible Values	<absolute-size> <ul style="list-style-type: none">xx-small x-small small medium large x-large xx-large <relative-size> <ul style="list-style-type: none">larger smaller<length>

	• <percentage>
Initial Value	Medium
Applies to	All elements
Inherited	Yes

6.2 LIST PROPERTIES

6.2.1 Introduction

In HTML, there are two types of lists:

- unordered lists - the list items are marked with bullets
- ordered lists - the list items are marked with numbers or letters

With CSS, lists can be styled further, and images can be used as the list item marker.

Different List Item Markers: The type of list item marker is specified with the list-style-type property:

Example

```
ul.a {list-style-type: circle;}
ul.b {list-style-type: square;}
ol.c {list-style-type: upper-roman;}
ol.d {list-style-type: lower-alpha;}
```

An Image as the List Item Marker: To specify an image as the list item marker, use the list-style-image property:

Example

```
ul
{
    list-style-image: url('sqpurple.gif');
}
```

The example above does not display equally in all browsers. IE and Opera will display the image-marker a little bit higher than Firefox, Chrome, and Safari.

If you want the image-marker to be placed equally in all browsers, a crossbrowser solution is explained below.

Crossbrowser Solution: The following example displays the image-marker equally in all browsers:

Example:

```

ul
{
    list-style-type: none;
    padding: 0px;
    margin: 0px;
}
ul li
{
    background-image: url(sqpurple.gif);
    background-repeat: no-repeat;
    background-position: 0px 5px;
    padding-left: 14px;
}

```

Example explained:

- For ul:
 - Set the list-style-type to none to remove the list item marker
 - Set both padding and margin to 0px (for cross-browser compatibility)
- For all li in ul:
 - Set the URL of the image, and show it only once (no-repeat)
 - Position the image where you want it (left 0px and down 5px)
 - Position the text in the list with padding-left

6.2.2 List Properties

Property	Description
<u>list-style</u>	Sets all the properties for a list in one declaration
<u>list-style-image</u>	Specifies an image as the list-item marker
<u>list-style-position</u>	Specifies if the list-item markers should appear inside or outside the content flow
<u>list-style-type</u>	Specifies the type of list-item marker

6.2.2.1 list-style Property

The list-style shorthand property sets all the list properties in one declaration. The properties that can be set are (in order): list-style-type, list-style-position, list-style-image.

If one of the values above is missing, e.g. "list-style: circle inside;" the default value for the missing property will be inserted, if any.

Default value:	disc outside none
Inherited:	Yes
Version:	CSS1
JavaScript syntax:	<i>object.style.listStyle</i> ="decimal inside"

6.2.2.2 list-style-image Property

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

Default value:	None
Inherited:	Yes
Version:	CSS1
JavaScript syntax:	<i>object.style.listStyleImage</i> ="url('/images/blueball.gif')"

6.2.2.3 list-style-position Property

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

Default value:	Outside
Inherited:	Yes
Version:	CSS1
JavaScript syntax:	<i>object.style.listStylePosition</i> ="inside"

6.2.2.4 list-style-type Property

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

Default value:	"disc" for and "decimal" for
-----------------------	--

Inherited:	Yes
Version:	CSS1
JavaScript syntax:	<i>object.style.listStyleType="square"</i>

6.3 COLOR

6.3.1 Colors

Colors in Web technology can be specified by the following methods:

- Hexadecimal colors
- RGB colors
- RGBA colors
- HSL colors
- HSLA colors
- Predefined/Cross-browser color names

Hexadecimal Colors: Hexadecimal color values are supported in all major browsers. A hexadecimal color is specified with: #RRGGBB, where the RR (red), GG (green) and BB (blue) hexadecimal integers specify the components of the color. All values must be between 0 and FF.

For example, the #0000ff value is rendered as blue, because the blue component is set to its highest value (ff) and the others are set to 0.

Example:

Define different HEX colors:

```
#p1 {background-color:#ff0000;} /* red */
#p2 {background-color:#00ff00;} /* green */
#p3 {background-color:#0000ff;} /* blue */
```

RGB Colors: RGB color values are supported in all major browsers. An RGB color value is specified with: rgb(red, green, blue). Each parameter (red, green, and blue) defines the intensity of the color and can be an integer between 0 and 255 or a percentage value (from 0% to 100%).

For example, the rgb(0,0,255) value is rendered as blue, because the blue parameter is set to its highest value (255) and the others are set to 0.

Also, the following values define equal color: rgb(0,0,255) and rgb(0%,0%,100%).

Example:

Define different RGB colors:

```
#p1 {background-color:rgb(255,0,0);} / * red */  
#p2 {background-color:rgb(0,255,0);} / * green */  
#p3 {background-color:rgb(0,0,255);} / * blue */
```

RGBA Colors: RGBA color values are supported in IE9+, Firefox 3+, Chrome, Safari, and in Opera 10+. RGBA color values are an extension of RGB color values with an alpha channel - which specifies the opacity of the object. An RGBA color value is specified with: rgba(red, green, blue, alpha). The alpha parameter is a number between 0.0 (fully transparent) and 1.0 (fully opaque).

Example:

Define different RGB colors with opacity:

```
#p1 {background-color:rgba(255,0,0,0.3);} / * red with opacity */  
#p2 {background-color:rgba(0,255,0,0.3);} / * green with opacity */  
#p3 {background-color:rgba(0,0,255,0.3);} / * blue with opacity */
```

HSL Colors: HSL color values are supported in IE9+, Firefox, Chrome, Safari, and in Opera 10+. HSL stands for hue, saturation, and lightness - and represents a cylindrical-coordinate representation of colors.

An HSL color value is specified with: hsl(hue, saturation, lightness).

Hue is a degree on the color wheel (from 0 to 360) - 0 (or 360) is red, 120 is green, 240 is blue. Saturation is a percentage value; 0% means a shade of gray and 100% is the full color. Lightness is also a percentage; 0% is black, 100% is white.

Example:

Define different HSL colors:

```
#p1 {background-color:hsl(120,100%,50%);} / * green */  
#p2 {background-color:hsl(120,100%,75%);} / * light green */  
#p3 {background-color:hsl(120,100%,25%);} / * dark green */  
#p4 {background-color:hsl(120,60%,70%);} / * pastel green */
```

HSLA Colors: HSLA color values are supported in IE9+, Firefox 3+, Chrome, Safari, and in Opera 10+. HSLA color values are an extension of HSL color values with an alpha channel - which specifies the opacity of the object.

An HSLA color value is specified with: `hsla(hue, saturation, lightness, alpha)`, where the alpha parameter defines the opacity. The alpha parameter is a number between 0.0 (fully transparent) and 1.0 (fully opaque).

Example:

Define different HSL colors with opacity:

```
#p1 {background-color:hsla(120,100%,50%,0.3);} /* green with opacity */
#p2 {background-color:hsla(120,100%,75%,0.3);} /* light green with opacity */
#p3 {background-color:hsla(120,100%,25%,0.3);} /* dark green with opacity */
#p4 {background-color:hsla(120,60%,70%,0.3);} /* pastel green with opacity */
```

6.3.2 Color Property

The color property specifies the color of text.

Default value:	<i>not specified</i>
Inherited:	Yes
Version:	CSS1
JavaScript syntax:	<i>object.style.color="#FF0000"</i>

Example

Set the text-color for different elements:

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

6.4 ALIGNMENT OF TEXT

Text alignment refers to how the lines of text on the page line up. There are four basic ways to align text on the web:

1. Left

- Default setting when text alignment or the direction of text is not set on a web page.
- Default setting for languages read top to bottom, and left to right.
- Each line of text is even with the left margin.
- The right edges of left aligned text are jagged.

2. Right

- Default setting if the direction of the language used on the web page is right to left.
- Each line of text is aligned with the right margin.
- The left edge of right aligned text is jagged.

3. Center

- As the name implies, the text is centered on the web page.

4. Justified

- Both the left and right edges of text in a line are even against the left and right margins.
- The words are spaced out to fill the line of text.
- Most commonly used in newspapers and other printed media.

6.4.1 Setting Text Alignment: The most familiar way to align text is the old way of including aligns your setting in an HTML element tag. This old method has 2 problems:

1. If you try to validate your HTML coding using a Strict DOCTYPE declaration or Transitional DOCTYPE declaration when using XHTML, you will not pass. The align=your setting attribute was eliminated in HTML 4.01.

“align = left|center|right|justify [CI]

Deprecated. This attribute specifies the horizontal alignment of its element with respect to the surrounding context.”

Alignment, font styles, and horizontal rules in HTML documents – 15.1.2
Alignment of the HTML 4.01 Specification

2. Using this method will not let you format the text in other ways without adding a bunch of additional coding.

To set text alignment for an HTML element, you use a style in your style sheet.

```
.centered {  
  text-align: center;  
}
```

In your HTML element you specify the style:

```
<p class="centered">Text in paragraph</p>
```

6.4.2 Set body text alignment to minimize gaps and maximize scanning

Text alignment in Web pages is, by default, to the left, with ragged edges on the right. Justified text—sometimes called *newspaper columns*, where both edges of the text are aligned—is rare on the Web.

text-align: left;

text-align: justify;

In print, justified text is created using a variety of techniques including word spacing, letter spacing, hyphenation, and glyph reshaping. In addition, well-formed justification is calculated on a paragraph level to prevent “rivers” of white space flowing down the middle. On the Web, unfortunately, justification is simply created by adding small amounts of space between words. On the screen, where you can only add whole pixels, this often results in uncomfortably large amounts of space between some words, especially in narrower columns.

When choosing to use left or justified alignment, keep in mind these factors:

- Justified text is often seen as more formal and structured, while left alignment is more informal and approachable.
- Justified text reinforces the grid structure of a page but can be harder to scan, since it often creates rivers of white space throughout the text, which interrupts the eye path.

- Left-aligned text adds an element of white space to the right edge, softening the overall appearance of the page.

6.4.3 Center or right-justify text for effect and variety

More rarely used, centering or right-justifying text can create a specific feeling on the page.

text-align: center;
text-align: right;

Centering and right aligning text is integrally dependent on the design you are creating and how you want your readers to scan the page. While using a variety of justifications helps create rhythm and motion on your page, it can quickly seem cluttered or obnoxious. Always have a specific purpose for the variance of alignment, and use it sparingly. Here are a few ideas:

Bulleted or numbered lists should not be centered or right aligned, as this makes them harder to scan by moving the beginning of each line around.

- Center section or module titles/headers if you want to make your site look a little different. Generally, section titles are best when left aligned, but centering them gives your designs a unique feel and may also improve scan ability.
- Right-align text in the left column of a page or table if it helps show a closer relationship between the elements in adjacent columns.

6.4.4 Increase margins for longer quotations and style the citation

Short quotes of less than three lines are included in a paragraph surround by quotation marks, requiring no other special formatting. In HTML, the block quote tag is used to set off a block of text as a quotation, generally of two lines of text or longer. The quotation should be styled to distinguish it from other text by indenting its left and right margins and increasing the top and bottom margins. The amount of left/right indentation is based on the width of the column and then adjusted so that it does not conflict with any other indents. A good measure to offset block quotes is to double the font size (2em), although more or less space may be required for wider or narrower columns:

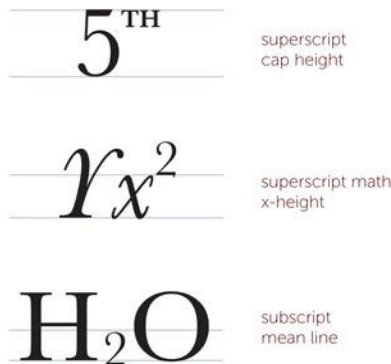
blockquote { margin: 2em; }

6.4.5 Set footnotes and scientific or mathematical annotations using positioning rather than vertical alignment

Vertical text alignment allows you to adjust the position of inline text in relation to its natural baseline, shifting it up or down. For footnotes, mathematics, and scientific notation, it will not be enough to simply raise or lower the characters; you will also need to reduce their size relative to the surrounding text. These styles can be applied to the superscript and subscript tags, setting the vertical position to the baseline and then setting a position relative to that:

```
sup, sub { font-size: .5em;
           vertical-align: baseline;
           position: relative; }
sup { top: -.65em; }
sup.math { top: -.8em }
sub { top: .2em; }
```

Although vertical-align provides several values to set the vertical position of the text, these have proved to be unreliable in multi-column layouts. The exact values will vary depending on the font, and you may also need to add some left/right margins to add breathing room.



6.5 THE BOX MODEL

6.5.1 Introduction

The box is a virtual diagram of the content (text or graphics), or any padding, border, or margin added to that content. In a document, each element is represented as a rectangular box. Determining the size, properties — like its color, background, borders aspect — and the position of these boxes is the goal of the rendering engine.

In CSS, each of these rectangular boxes is described using the standard *box model*. This model describes the content of the space taken by an element. Each box has four edges: the **margin edge**, **border edge**, **padding edge**, and **content edge**.

The CSS Box Models consists of:

- Content area: The element's background property
- Padding area: The element's background property
- Border area: The element's border properties
- Margin area: No background, margins are always transparent

The padding, border, and margins can each be specified with different values for their top, bottom, left, and right sides. Some of these properties are: padding-top, border-bottom-width, and margin-left.

6.5.2 Implementing the Box Model

The box model is best demonstrated with a short example. The calculation we'll use to ascertain the total space required to accommodate an element on the page (ignoring margin collapse for the time being—see below for more on this) will be as follows:

```
Total width = left margin + left border + left padding + width +  
               right padding + right border + right margin  
  
Total height = top margin + top border + top padding + height +  
               bottom padding + bottom border + bottom margin
```

Here's our example CSS—a rule set that contains declarations for all the box properties of an element that has the class "box":

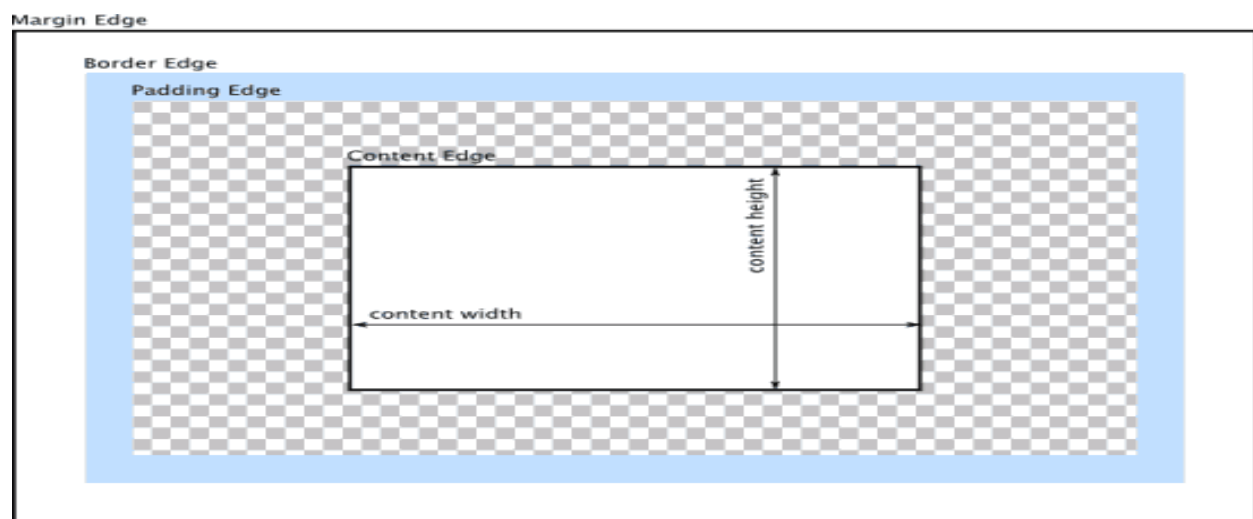
```
.box {  
  
  width: 300px;  
  
  height: 200px;
```

```
padding: 10px;  
  
border: 1px solid #000;  
  
margin: 15px;  
  
}
```

The total size of the element above will be calculated as follows:

Total width = $15 + 1 + 10 + 300 + 10 + 1 + 15 = 352\text{px}$

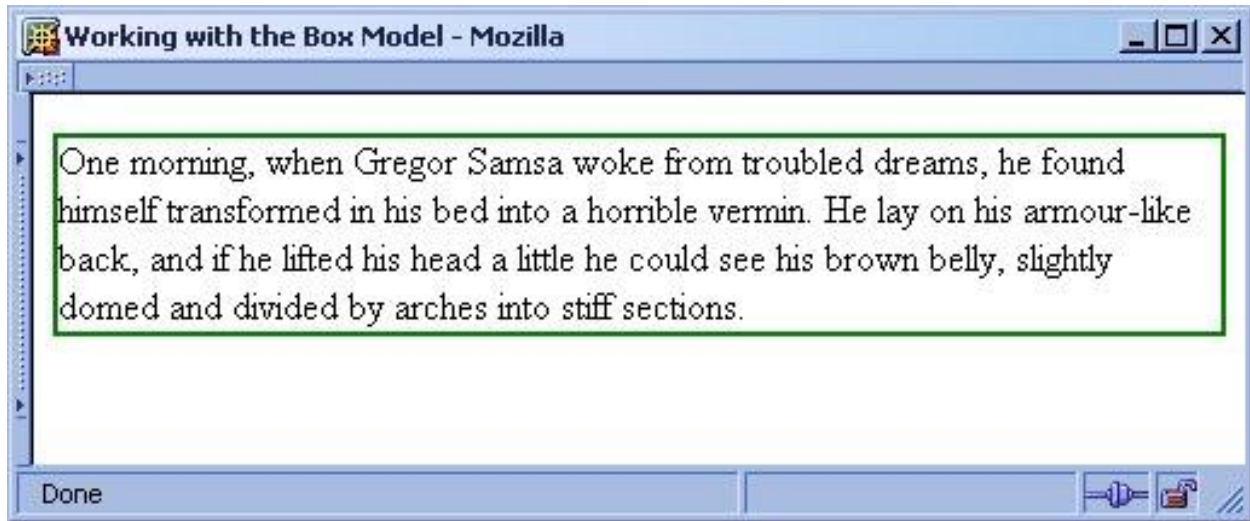
Total height = $15 + 1 + 10 + 200 + 10 + 1 + 15 = 252\text{px}$



For example 1, let's start with a simple paragraph:

```
<p> One morning, when Gregor Samsa woke from troubled dreams, he found himself  
transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his  
head a little he could see his brown belly, slightly domed and divided by arches into stiff section.  
</p>
```

Figure 1 shows the text plus the fact that there's a box generated as shown by the visual presence of the box's border.



[Figure 1](#) Visualizing a paragraph's box.

The **content area** is the area containing the real content of the element. It is located inside the *content edge*, and its dimensions are the *content width*, or *content-box width*, and the *content height*, or *content-box height*.

If the CSS box-sizing property is not set to another value than its default, the CSS properties width, min-width, max-width, height, min-height and max-height controls the content size.

The **padding area** extends the content area with the empty area between the content and the eventual borders surrounding it. It often has a background, a color or an image (in that order, an opaque image hiding the background color), and is located inside the *padding edge*. Its dimensions are the *padding-box width* and the *padding-box height*.

The space between the padding and the content edge can be controlled using the padding-top, padding-right, padding-bottom, padding-left and the shorthand padding CSS properties.

The **border area** extends the padding area with the area containing the borders. It is the area inside the *border edge*, and its dimensions are the *border-box width* and the *border-box height*. This area depends of the size of the border that is defined by the border-width property or the shorthand border.

The **margin area** extends the border area with an empty area used to separate the element from its neighbors. It is the area inside the *margin edge*, and its dimensions are the *margin-box width* and the *margin-box height*.

The size of the margin area is controlled using the margin-top, margin-right, margin-bottom, margin-left and the shorthand margin CSS properties.

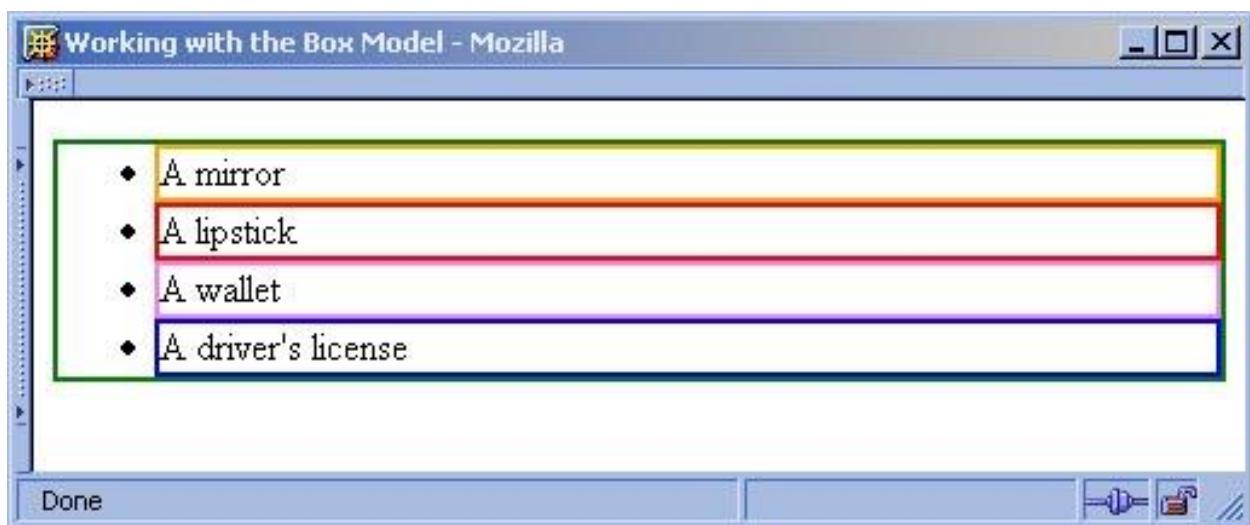
When margin collapsing happens, the margin area is not clearly defined since margins are shared between boxes.

Finally, note that, for non-replaced inline elements, the amount of space taken up (the contribution to the height of the line) is determined by the line-height property, even though the border and padding appear visually around the content.

How about something a bit more complicated, such as a list? Example 2 shows an unordered list:

```
<ul>
  <li>A mirror</li>
  <li>A lipstick</li>
  <li>A wallet</li>
  <li>A driver's license</li>
</ul>
```

Figure 2 shows how both the ul element and the li elements each generate a box. The ul element contains all the li elements including their generated bullets, and I've provided a different color border style to show the individual boxes for each li, which can then each be styled differently.

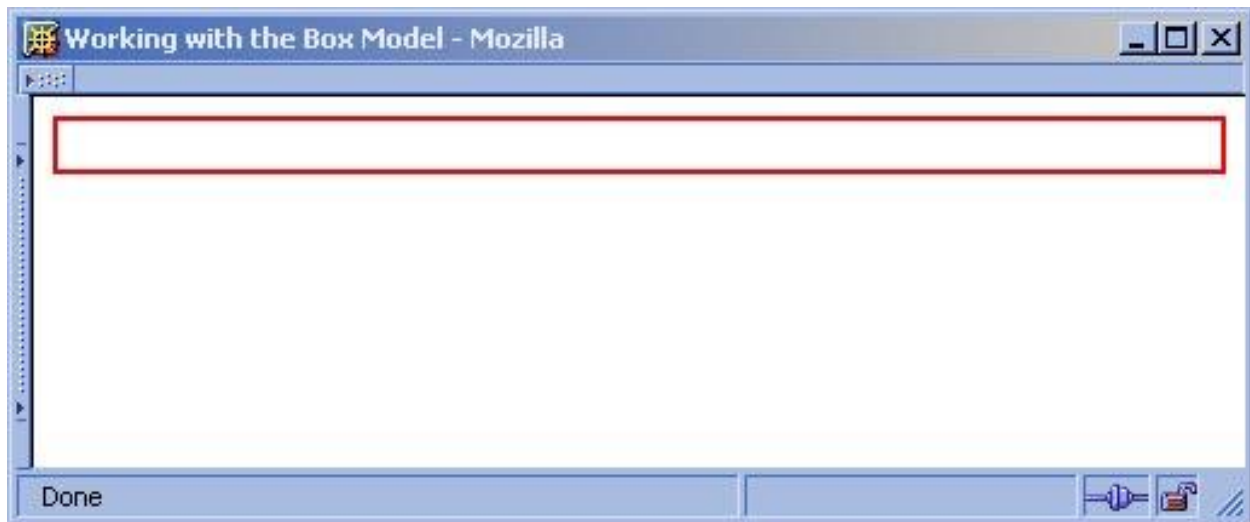


[Figure 2](#) Boxes on all elements.

For example 3, I'll take an empty (also referred to as replaced) element which has no text content. I've added a little padding so you can see the display box that it generates, too:

```
<hr />
```

Figure 3 shows the results, demonstrating even how unexpected elements have boxes as well. Note that in the case of replaced elements, there is no content box per se, but you can set a specific width and add padding, borders and margins, too.



[Figure 3](#) Boxes for empty elements, too.

6.5.3 Different Box Model

The Microsoft Box Model: Microsoft, being a proprietary commercial entity, has often chosen to implement aspects of browsers that are non-standard. Fortunately, this practice is being reduced now in favor of including W3C standards in their software, as well as their own features. Until such a time as existing discrepancies don't impact our day-to-day tasks, we still have to look at where the issues lie and how they affect the work we do.

The Microsoft Box Model can be seen in Figure 4.

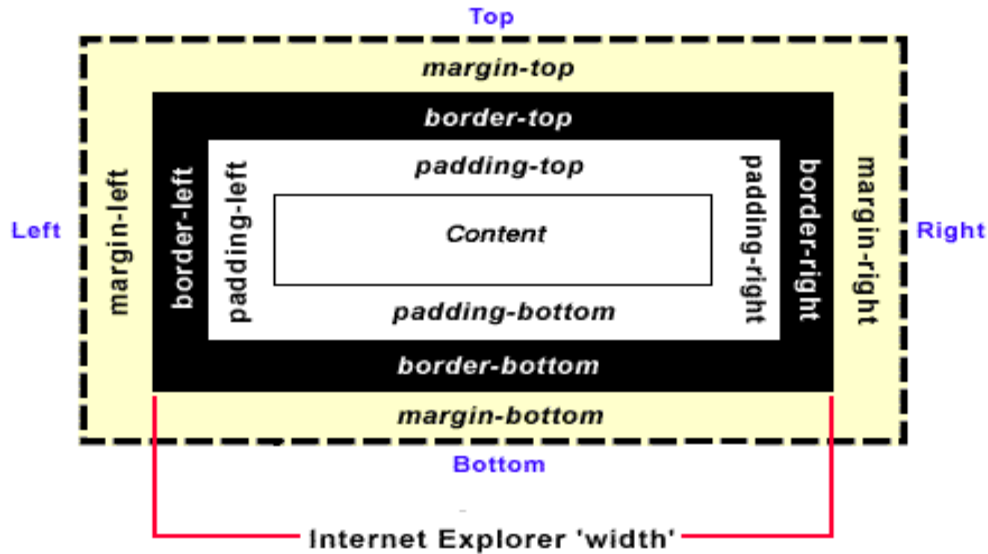


Figure 4 The Microsoft Box Model

Notice the following issues:

- The Microsoft Box Model places the padding inside the box
- In this model, margins are **not** calculated within the box
- Borders are also included in width calculations for the box

This translates to a big problem. If I want to set a width to the paragraph I showed in Example 1, I have to *include* the padding, margin and borders in the total width. Consider this CSS:

```
P {
    width: 100px;
    padding-left: 10px;
    padding-right: 10px;
    border-left: 1px solid blue;
    border-right: 1px solid red;
    margin-left: 5px;
    margin-right: 5px;
```

}

How wide is my box, really? If you guessed 122 pixels, you're correct, according to the Microsoft calculation. That's because I've had to add the content width of 100 plus the left padding and right padding, plus the 1 pixel borders to the left and to the right of the content box. The margins aren't calculated as part of the Microsoft Box Model.

The W3C (Standard) Box Model: The W3C's Box Model is decidedly unlike Microsoft's model (Figure 5).

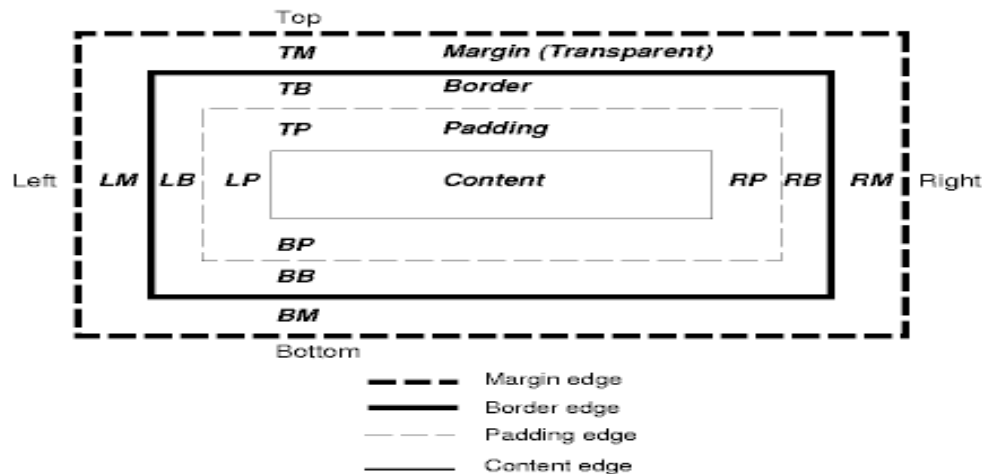


Figure 5 The W3C Box Model

Notice the following:

- The W3C Box Model places the padding outside the box
- In this model, margins are calculated outside the box
- Borders are also excluded from width calculations for the box
- The width of a calculated box using the W3C box model is the explicit width of the content section

If we consider the same CSS applied to this box, how wide is my box? If you guessed 100 pixels, you're correct! You can see how prior to IE 6.0, which has a fix for the box model, we had to use hacks to get around this problem, or there'd be no consistency in our CSS-based element designs from browser to browser.

6.6 BACKGROUND IMAGE

6.6.1 Introduction

The background-image property sets one or more background images for an element. The background of an element is the total size of the element, including padding and border (but not the margin).

By default, a background-image is placed at the top-left corner of an element, and repeated both vertically and horizontally.

Default value:	none
Inherited:	no
Version:	CSS1
JavaScript syntax:	<i>object</i> .style.backgroundImage="url(stars.gif)"

Property Values

Value	Description
url('URL')	The URL to the image. To specify more than one image, separate the URLs with a comma
None	No background image will be displayed. This is default
inherit	Specifies that the background image should be inherited from the parent element

The CSS background-image property sets one or several background images for an element. The images are drawn on successive stacking context layers, with the first specified being drawn as if it is the closest to the user. The borders of the element are then drawn on top of them, and the background-color is drawn beneath them.

How the images are drawn relative to the box and its borders is defined by the background-clip and background-origin CSS properties.

If a specified image cannot be drawn (for example when the file denoted by the specified URI cannot be loaded), browsers handle them as if it was the none value.

Syntax:

Formal syntax: <bg-image>#

background-image: none

background-image: url(http://www.example.com/images/bck.png)

background-image: inherit

6.6.2 Scaling Background Images

The background-size CSS property makes it possible to adjust the size of background images, instead of the default behavior of tiling the image at its full size. You can scale the image upward or downward as desired.

Tiling a large image: Let's consider a large image, a 1233x1233 Firefox logo image. We want (for some reason likely involving horrifyingly bad site design) to tile four copies of this image into a 300x300 pixel square, resulting in this look:



This can be achieved using the following CSS:

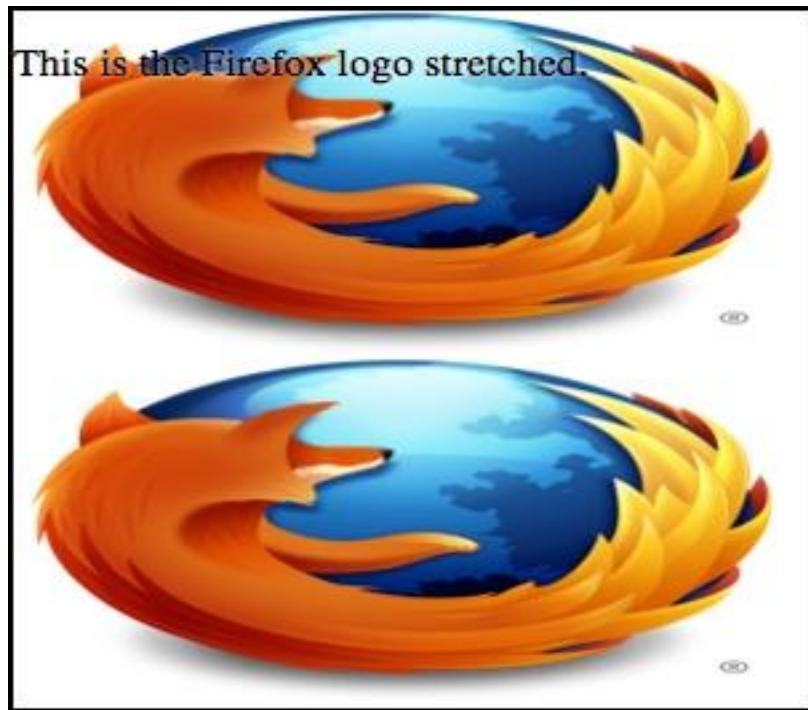
```
.square {  
  width: 300px;  
  height: 300px;  
  background-image: url(fxlogo.png);  
  border: solid 2px;
```

```
text-shadow: white 0px 0px 2px;  
font-size: 16px;  
background-size: 150px;  
}
```

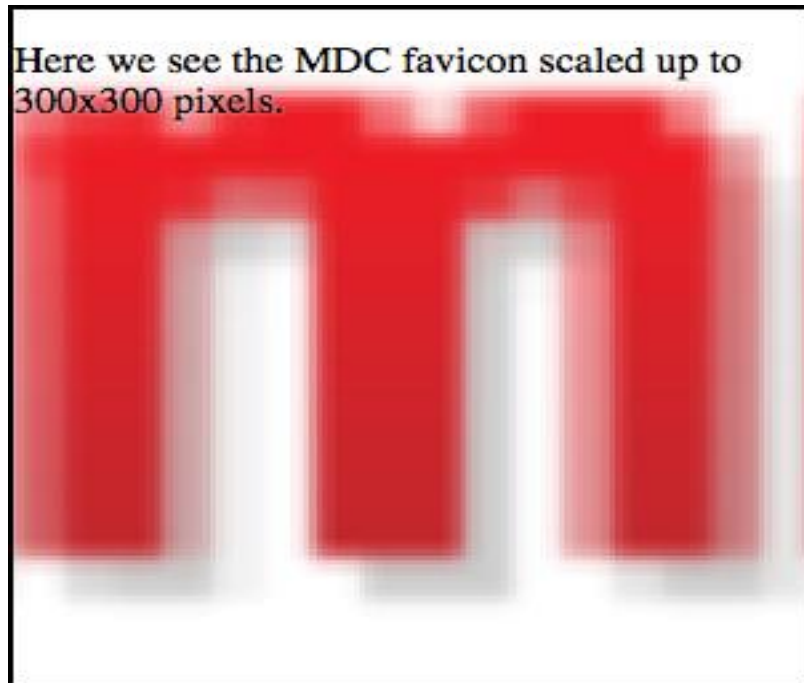
There is no need to prefix background-size anymore, though you may consider adding some prefixed version of it if you are targeting very old browsers.

Stretching an image: You can also specify both the horizontal and vertical sizes of the image, like this:

```
background-size: 300px 150px;
```



Scaling an image up: On the other end of the spectrum, you can scale an image up in the background. Here we scale a 16x16 pixel favicon to 300x300 pixels:



```
.square2 {  
    width: 300px;  
    height: 300px;  
    background-image: url(favicon.png);  
    background-size: 300px;  
    border: solid 2px;  
    text-shadow: white 0px 0px 2px;  
    font-size: 16px;  
}
```

Special values: "contain" and "cover": Beside <length> values, the background-size CSS property offers two special size values, contain and cover. Let's take a look at these.

Contain: The contain value specifies that regardless of the size of the containing box, the background image should be scaled so that each side is as large as possible while not exceeding the length of the corresponding side of the container. Try resizing this window using a browser that supports scaling background images (such as Firefox 3.6 or later) to see this in action in the live example below.

```
<div class="bgSizeContain">  
    <p> Try resizing this window and see what happens. </p>  
</div>
```

```
.bgSizeContain
{
    height: 200px;
    background-image: url(https://developer.mozilla.org/files/2917/fxlogo.png);
    background-size: contain;
    border: 2px solid darkgray;
    color: #000; text-shadow: 1px 1px 0 #fff;
}
```

Cover: The cover value specifies that the background image should be sized so that it is as small as possible while ensuring that both dimensions are greater than or equal to the corresponding size of the container.

This example uses following HTML & CSS:

```
<div class="bgSizeContain">
    <p> Try resizing this window and see what happens. </p>
</div>
```

```
.bgSizeCover
{
    height: 200px;
    background-image: url('/files/2917/fxlogo.png');
    background-size: cover;
    border: 2px solid darkgray;
    color: #000; text-shadow: 1px 1px 0 #fff;
}
```

6.7 SUMMARY

In this unit, you have learnt FONT and LIST properties and its usage. The color and alignment of text which are used in Webpages are discussed in detail. The Box Model have been explained in detail. The syntax and properties of Background image have been discussed in detail.

6.8 KEYWORDS

Generic Font Family, Font Variant, Ordered list, Unordered list, Crossbrowser, RGB color, RGBA color, HSL color, HSLA color.

6.9 UNIT END EXERCISE AND ANSWER

- 1) Explain the different Font and List properties.

- 2) Explain the color scheme involved in Web pages.
- 3) Explain Text Alignment and Box Model in detail.
- 4) How are Background Images inserted in Web Pages? Explain with the syntax

6.10 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

UNIT 7: THE `` and `<div>` TAG

Structure:

- 7.0 Objectives
- 7.1 Introduction
- 7.2 Practical Usage
- 7.3 Overuse
- 7.4 Span Tag
- 7.5 Div Tag
- 7.6 Differences and default behavior
- 7.7 Summary
- 7.8 Keywords
- 7.9 Unit-end exercises and answers
- 7.10 Suggested readings

7.0 OBJECTIVES

At the end of this unit you will be able to know the usage of `` and `<div>` tag and its difference and default behavior.

7.1 INTRODUCTION

The span and div elements are used for generic organizational or stylistic applications, typically when extant meaningful elements have exhausted their purpose.

Most elements signify the specific meaning of their content – i.e. the element describes, and can be made to function according to, the type of data contained within. For example, a `<p>` element should contain a paragraph of text, while an `<h1>` element should contain the highest-level heading of the page or section; agents should distinguish them accordingly. Span and div signify no specific meaning besides the generic grouping of content, and are therefore more appropriate for creating organization or stylistic additions without signifying superfluous meaning.

7.2 PRACTICAL USAGE

Span and div elements are used purely to imply a logical grouping of enclosed elements. There are three main reasons to use span and div tags with class or id attributes:

7.2.1 Styling with CSS

Perhaps the most common use of `` and `<div>` elements is to carry class or id attributes in conjunction with CSS to apply layout, typographic, color, and other presentation attributes to parts of the content. CSS does not just apply to visual styling: when spoken out loud by a voice browser, CSS styling can affect speech-rate, stress, richness and even position within a stereophonic image.

For these reasons, and for compatibility with the concepts of the semantic web, discussed below, attributes attached to elements within any HTML should describe their semantic purpose, rather than merely their intended display properties in one particular medium. For example, the HTML in `password too short` is semantically weak, whereas `<em class="warning">password too short` uses an `em` element to signify emphasis, and uses a more appropriate class name. By the correct use of CSS, 'warnings' may be rendered in a red, bold font on a screen, but when printed out they may be omitted, as by then it is too late to do anything about them. Perhaps when spoken they should be given extra stress, and a small reduction in speech-rate. The second example is semantically correct markup, rather than merely presentational.

7.2.2 Semantic clarity

This kind of grouping and labeling of parts of the page content might be introduced purely to make the page more semantically meaningful in general terms. It is impossible to say how and in what ways the World Wide Web will develop in years and decades to come. Web pages designed today may still be in use when information systems that we cannot yet imagine are trawling, processing, and classifying the web. Even today's search engines such as Google and others use proprietary information processing algorithms of considerable complexity.

For some years, the World Wide Web Consortium (W3C) has been running a major Semantic Web project designed to make the whole web increasingly useful and meaningful to today's and the future's information systems.

The microformats movement is an attempt to build an idea of semantic classes. For example, microformats-aware software might automatically find an element like `123-456-7890` and allow for automatic dialing of the telephone number.

7.2.3 Access from code

Once the HTML or XHTML markup is delivered to a page-visitor's client browser, there is a chance that client-side code will need to navigate the internal structure (or Document Object Model) of the web page. The most common reason for this is that the page is delivered with client-side JavaScript that will produce on-going dynamic behavior after the page is rendered. For example, if rolling the mouse over a 'Buy now' link is meant to make the price, elsewhere on the page, become emphasized, JavaScript code can do this, but JavaScript needs to identify the price element, wherever it is in the markup, in order to affect it. The following markup would suffice: `<div id="price">$45.99</div>`. Another example is the Ajax programming technique, where, for example, clicking a hypertext link may cause JavaScript code to retrieve the text for a new price quotation to display in place of the current one within the page, without re-loading the whole page. When the new text arrives back from the server, the JavaScript must identify the exact region on the page to replace with the new information.

Less common, but just as important examples of code gaining access to final web pages, and having to use span and div elements' class or id attributes to navigate within the page include the use of automatic testing tools. On dynamically generated HTML, this may include the use of automatic page testing tools such as Http Unit, a member of the xUnit family, and load or stress testing tools such as Apache JMeter when applied to form-driven web sites.

7.3 OVERUSE

The judicious use of div and span is a vital part of HTML and XHTML markup. However, they are sometimes overused.

For example, when structurally and semantically a series of items need an outer, containing element and then further containers for each item, then there are various list structures available in HTML, one of which may be preferable to a homemade mixture of div and span elements.

For example, this...

```
<ul class="menu">
  <li>Main page</li>
  <li>Contents</li>
  <li>Help</li>
</ul>
```

...is usually preferable to this:

```
<div class="menu">
  <span>Main page</span>
  <span>Contents</span>
  <span>Help</span>
</div>
```

Other examples of the semantic use of HTML rather than div and span elements include the use of fieldset elements to divide up a web form, the use of legend elements to identify such divisions and the use of label to identify form input elements rather than div, span or table elements used for such purposes.

7.4 Span tag

The span tags are used to define the span element. The span element begins with the tag and ends with the tag. The span element is used to provide additional structure to an HTML document. The span element is an inline element and can be used to give different characteristics to specific parts of other elements.

The `` tag is an HTML tag and is used to target **inline** elements in a document. The `` tag provides no visual change by itself and it works with CSS codes to style a part of a larger group. Basically CSS targets tags to style elements, but **if you happen to have an element without a tag, you can wrap `` tag around the element and then apply the id or class CSS to that element.**

For instance to change the color of one word in a sentence, you can put `` tag around it, then assign a class for the `` and style it in the CSS section.

7.4.1 Span Element Attributes

"span" element attributes include:

- **class** - The class attribute is used in conjunction with style sheets to associate an element with a class. The class attribute can set a class for specific element types or it can be independent of element types and work for all elements. The class attribute will provide the settings for specific style formatting.
- **ID** - The ID attribute is used to apply style settings to specific individual HTML elements.
- **style** - The style attribute is used to apply style settings for the specific element the style attribute is included with. An example is " `<p style="font: 16pt courier">` - This sets the style or color of the text. This statement starts a paragraph with color, green: `<p style="color: green">`. The STYLE attribute is common to most HTML elements.
- **title** - Used to give specific elements a title which may appear as a tooltip in some browsers when the mouse is held at or near the element.

7.4.2 Span Element Contents

The span element may contain other inline elements as listed at the HTML Inline Elements page.

7.4.3 Span Element Inside

The span element can be contained inside any block, inline, or combination block/inline elements.

7.4.4 Span Element Use Example

This is the HTML code for a paragraph demonstrating use of the "span" element while using the "style" attribute to change characteristics of part of the content of a paragraph element.

```
<p style="text-align: 'center'; font: '16pt courier'; color: 'blue'">
```

The color of the sky is blue, but if you look at the span style="color: 'green'">trees, they are green. This effect is produced using the `` element with the style="color: 'green'" attribute set.

```
</p>
```

Here is how it looks:

The color of the sky is blue, but if you look at the trees, they are green. This effect is produced using the `` element with the style="color: 'green'" attribute set.

In the code below we are applying red color to the word "example":

```
<html>
    <head><style type="text/css">
        .red {color: #ff0000;}
    </style>
</head>
<body>
    This is an <span class="red">example</span> of the selector
</body>
</html>
```

Result:

This is an **example** of class selector.

7.5 DIV TAG

Div is short for division and the `<div>` element is a block-level element that is used to divide the page into logical sections. `<div>` tags can hold any element inside them including text and image. Div tags are well-suited to take over from tables as a layout tool for laying out web pages. This element indicates a generic block of content that should be treated as a logical unit for scripting or styling purposes.

Defining ID: div tags don't have a default identity, so anytime you create a `<div>`, give it an id. Each unique id will be referred to in the `<style>` section starting with # sign, and can be styled with CSS rules.

The `<div>` tag is a container for a group of elements. It is not seen on a webpage, but works behind the scenes to organize the layout of a webpage a certain way. Div is a *block-*

level element which means that it will automatically begin on a new line in the browser. This is opposed to *inline elements* which continue on the same line.

Standard Syntax

```
<div
    align="center | justify | left | right" (transitional only)
    class="class name(s)"
    dir="ltr | rtl"
    id="unique alphanumeric identifier"
    lang="language code"
    style="style information"
    title="advisory text">

</div>
```

7.5.1 Attributes of the <div> tag

- id - Denotes a unique name for the container.
- class - Denotes what class this container belongs to (This attribute as well as id is used for styling purposes. For more on styling, read our [HTML stylesheets](#) page or our [CSS tutorials](#))
- style - Sets a set of styles using for this container such as background color, font size, and border color.
- Deprecated. align - Sets how content will be aligned inside the container. Possible values include "left", "right", "center", or "justify"

7.5.2 Element-Specific Attribute

Nowrap: This Internet Explorer–specific attribute is used to control the wrapping of text within a <div> tag. If set to yes, text should not wrap. The default is no. CSS rules should be used instead of this attribute.

Examples

```
<div align="justify">
    <!-- IE syntax -->
    All text within this division will be justified
```

```

</div>
<div class="special" id="div1" style="background-color: yellow;">
    Divs are useful for setting arbitrary style
</div>
<div class="container">
    <div class="wrapper">
        <div class="content"><p>I have divitis</p>
    </div>
</div>
</div>

```

Note:

- A **<div>** tag is a generic block tag and is very useful for binding scripts or styles to an arbitrary section of a document. It complements ****, which is used inline.
- Excessive use of **<div>** tags is almost as bad as excessive use of tables, particularly when structuring page content.
- The HTML 4 specification specifies that the **datafld**, **dataformatas**, and **datasrc** attributes are reserved for **<div>** and might be supported in the future. They were removed from XHTML, but Internet Explorer supports them for data binding.
- Under the HTML 4.01 strict specification, the **align** attribute is not supported.
- HTML 7.2 supports only the **align** attribute.

7.5.3 <div> Default Size and Position

Anytime you create a **<div>** tag, it will have the following specs as default:

Width: 100% of the available screen width

Height: 0 pixels

stack underneath one another (unless you position them)

So the first thing you need to do is to assign the following for each **<div>**:

1. **Width**
2. **Height**
3. **Background color**

4. **Positioning** (see next section)

NOTE: if you don't assign **height** for a Div, it'll **stretch** to accommodate its content.

7.5.4 Normal Flow of <div> Tags

<div> tags stack on the bottom of each other as default. So basically if you insert two <div> tags one after another, the second one will automatically go to the next line under the first <div> tag.

Example 1: Stacking <div> tags

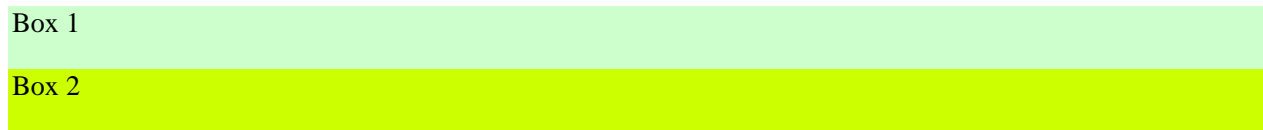
```
<html>
  <head>
    <style type="text/css">
      #box1
      {
        width: 200px;
        height: 50px;
        background-color: #ccffcc;
      }
      #box2
      {
        width: 200px;
        height: 50px;
        background-color: #ccff00;
      }
    </style>
  </head>
  <body>
    <div id="box1">
      Box1.
    </div>
    <div id="box2">
```

```

        Box2.
    </div>
</body>
</html>

```

Result:



7.5.5 Floating <div> Tags

In CSS you can float an element meaning push it to the left or right, allowing other elements to wrap around it. Float is very often used for images, but it is also useful when working with <div> tags layout web pages.

When you float an element to the **left** or **right**, it will be pushed to the side of whatever element it is contained within. **Floated <div> tags remove themselves from normal document flow** and position themselves to touch the edge of the containing block, therefore any **element that comes after them, moves up to occupy the space.**

Example 2: Floating to the left

```

<html>
  <head>
    <style type="text/css">
      #box1
      {
        width: 200px;
        height: 50px;
        background-color: #ccffcc;
        float: left;
      }
      #box2
      {

```

```

        width: 200px;
        height: 50px;
        background-color: #ccff00;
        float: left;
    }
</style>
</head>

<body>
    <div id="box1">
        Box1.
    </div>
    <div id="box2">
        Box2.
    </div>
</body>
</html>

```

Result:

Box 1

Box 2

7.5.6 Clearing Floats

Once you float a <div>, all other elements underneath the <div> will be affected by the float and move up and under the <div>. In order to stop this affect, you can **clear the float** for the second element.

Example 1: Only Box 1 is floated to the left, so Box 2 will move up and under Box 1

```

<html>
  <head>
    <style type="text/css">
      #box1
      {

```



```

        width: 200px;
        height: 50px;
        background-color: #ccffcc;
        float: left;
    }
    #box2
    {
        width: 200px;
        height: 70px;
        background-color: #ccff00;
    }
</style>
</head>
<body>
    <div id="box1">
        Box1.
    </div>
    <div id="box2">
        Box2.
    </div>
</body>
</html>

```

Result:



Example 2: Box 2 is cleared

```

<html>
  <head>
    <style type="text/css">
      #box1
      {
        width: 200px;
        height: 50px;
        background-color:#ccffcc;

```

```

        float: left;
    }
    #box2
    {
        width: 200px;
        height: 70px;
        background-color: #ccff00;
        clear: left;
    }
</style>
</head>

<body>
    <div id="box1">
        Box1.
    </div>
    <div id="box2">
        Box2.
    </div>
</body>
</html>

```

Result:



7.5.7 <div> Webpage Layout

<div> tags are most commonly used to layout web pages. With the usage of <div> tags you can divide the web page in smaller sections, create rows and columns and place these sections exactly where you want.

Here is an example of simple web page layouts:

```
<html>
```

```

<head>
  <style type="text/css">
    #box1 {width: 800px; height: 50px; background-color: #9F6;}
    #box2 {width: 200px; height: 300px; background-color: #96C; float:left;}
    #box3 {width: 600px; height: 300px; background-color: #0CF; float:right;}
    #box4 {width: 800px; height: 25px; background-color: #F30; clear:both;}
    #container {width: 800px; height: 675px; background-color: #FF9;
      margin-right:auto; margin-left:auto;}
  </style>
</head>

<body>
  <div id="container">
    <div id="box1"> Header </div>
    <div id="box2"> Nav. Bar </div>
    <div id="box3"> Body Content </div>
    <div id="box4"> Copyright </div></div>
</body>
</html>

```

7.6 DIFFERENCES AND DEFAULT BEHAVIOUR

There are multiple differences between div and span.

- The primary difference between the and <div> tags is that doesn't do any formatting of its own. The <div> tag acts includes a paragraph break, because it is defining a logical division in the document. The tag simply tells the browser to apply the style rules to whatever is within the .
- A span element is in-line and usually used for a small chunk of in-line HTML whereas a div element is block-line and used to group larger chunks of code
- A div is a block-level element whereas a span is an inline element. The div block visually isolates a section of a document on the page, and may contain other block-level components. The span element contains a piece of information inline with the

surrounding content, and may only contain other inline-level components. In practice, the default display of the elements can be changed by the use of Cascading Style Sheets (CSS); however the permitted contents of each element may not be changed. For example, regardless of CSS, a span element may not contain block-level children.

- I think of span as being used to divide up text. The `` and `<i>` tags are inline as well. You can apply them to text, and text stays there with its new boldness or italiciness or whatever you told it to look like.
- Divs, Tables, Headings and Paragraphs are blocks. I use divs to divide up parts of a page (menu, header, content, footer). Divs and other blocks like to separate themselves. If you put a div around words in a sentence, you would notice they go down a line into their own little box (you can add an outline around it in CSS if you want to see the borders), and the text after the div is another line down. With CSS you can do fancy stuff to divs and other blocks like float them or use absolute positioning.
- Inlines for breaking up text (ie - give some words a background color, different size, a border) without disturbing the flow of the text, and Blocks for breaking up parts of a page.
- The `<div>` tag should be used to separate the contents of a webpage into several parts based on the relationships between the elements on the webpage. The `<div>` tag should be used for grouping content. Use the `<div>` tag for things such as columns in a layout, and the location of a menu on a page. The `` tag should be used to change the style of something on a webpage without affecting the context it is within.
- Try to think of them as semantic elements. Use `` when you want to tag content used inside of blocks of text, for example and use `<div>`'s when you need to add extra structure to the page itself.
- A `` tag will only affect one line of text whereas a `<div>` tag is for a block of text.
- If you have a line wrapped in a span tag and it goes to a second line, it will not carry the attributes applied to it to the second line. Because a div tag is block level, it can be use similarly to a paragraph or header tag.

7.7 SUMMARY

An HTML element is an individual component of an HTML document or web page, once this has been parsed into the Document Object Model. HTML is composed of a tree of HTML elements and other nodes, such as text nodes. Each element can have HTML attributes specified. Elements can also have content, including other elements and text. HTML elements represent semantics, or meaning. For example, the `title` element represents the title of the document.

7.8 UNIT END EXERCISE AND ANSWERS

1. Explain the practical usage and overuse of `` and `<div>` tag in detail.
2. What is `` tag? Why is it used? Explain
3. What is `<div>` tag? Why is it used? Explain
4. Differentiate `` and `<div>` tag and its behaviour

7.9 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

UNIT 8: Conflict Resolution

Structure:

- 8.0 Objectives
- 8.1 Introduction
- 8.2 What happens when conflicts occurs?
- 8.3 Browser conflicts
- 8.4 Resolving conflicts
- 8.5 Conflicts in CSS
- 8.6 Resolving conflicts in CSS style rules
- 8.7 Summary
- 8.8 Keywords
- 8.9 Unit-end exercises and answers
- 8.20 Suggested readings

8.0 OBJECTIVES

In this unit, the participants will:

1. Demonstrate an understanding of conflict resolution strategies.
2. Practice assertive communication.

8.1 INTRODUCTION

Conflict resolution is the process of working through differences and disagreements, often with the help of a mediator who has the skills to help others work through conflicts.

CSS cascading inevitably leads to conflicts in the way styles are applied to elements. In this case, CSS conflict resolution can be seen under two main aspects: the first aspect is given by the way a browser handles such conflicts using the rules for cascading and specificity, while the second is under the direct control of the author of the style sheet who, in turn, can change his/her style rules to resolve these conflicts.

A typical example can be the situation of multiple CSS classes applied to the same element:

```
<body class="home event promo">  
</body>
```

Styles can be cumulative or conflicting. They're cumulative when all rules are different, while they're conflicting when one or more rules specify the same thing. Here's an example of cumulative styles:

```
body.home
{
    background: #333;
}

body.event
{
    font: 100% Arial, sans-serif;
    color: #fff;
}

body.promo
{
    border-top: 1em solid #c60;
}
```

Here the computed styles for the body element will be the sum of all the styles specified in the CSS classes because rules are not conflicting. Instead, here's an example of conflicting rules:

```
body.home
{
    background: #333;
}

body.event
{
    font: 100% Arial, sans-serif;
    color: #fff;
}

body.promo
{
    border-top: 1em solid #c60;
    font: 100% Verdana, sans-serif;
}
```

Conflict resolution in this case is handled by the browser which will apply the rule that tells that when there are conflicting rules and the selectors have the same specificity, then the

rule that comes later in the source wins. So in this particular case the font statement of the last rule will win over the preceding one.

Authors, however, can change their style rules to modify this default behavior, for example by incrementing the importance and weight of a statement:

```
body.event
{
    font: 100% Arial, sans-serif !important;
    color: #fff;
}
```

```
body.promo
{
    border-top: 1em solid #c60;
    font: 100% Verdana, sans-serif;
}
```

Or using more specific selectors:

```
html body.event
{
    font: 100% Arial, sans-serif;
    color: #fff;
}
```

```
body.promo
{
    border-top: 1em solid #c60;
    font: 100% Verdana, sans-serif;
}
```

Finally, they can combine both techniques:

```
html body.event
{
    font: 100% Arial, sans-serif !important;
    color: #fff;
}
```

```
body.promo
{
    border-top: 1em solid #c60;
    font: 100% Verdana, sans-serif;
```


}

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

8.2 WHAT HAPPENS WHEN CONFLICTS OCCUR?

There may be times when two or more declarations are applied to the same element. It is also possible that there may be a conflict between them. When conflicts like this occur, the declaration with the most weight is used. So, how is weight determined?

When documents load in a browser, all declarations are sorted and given a weight. They are then applied to a document based on this weight. There are four steps to sorting. As the process is quite complex, we will use one HTML element (<h2>) as an example.

1. Find all declarations whose selectors match a particular element.

After looking at all style sheets, all the declarations that match and <h2> element are found. The browser style sheet may include the following relevant rule:

```
h2
{
    font-size: 1.5em;
    margin: .83em 0;
    color: black;
}
```

The user’s style sheet could include the following relevant rules:

```
h2
{
    color: brown !important;
}
```

The author style sheet could include the following relevant rules:

```
h2
{
```

```

        color: green;
        font-size: 1.2em;
        padding: 10px;
    }

h2#main
{
    color: red;
}

h2.navigation
{
    color: blue;
}

```

2. Sort these declarations by weight and origin

The above rules are then sorted by weight and origin. For normal declarations, the author style sheets will override both the user style sheets and the browser style sheet unless there is an !important declaration in the user's style sheet. For “!important” declarations, user style sheets override author and browser style sheets.

For general users, <h2> elements that are not specifically styled with “#main” or “.navigation” will be styled with:

```

font-size: 1.2em; /* author wins over browser */
margin: .83em 0; /* specified only by browser */
padding: 10px;    /* specified only by author */
color: green;     /* author wins over browser */

```

For the user that has set an !important rule, <h2> elements that are not specifically styled with “#main” or “.navigation” will be styled with:

```

font-size: 1.2em; /* author wins over browser */
margin: .83em 0; /* specified only by browser */
padding: 10px;    /* specified only by author */
color: brown;     /* user wins due to !important */

```

3. Sort the selectors by specificity

Even though we have sorted out the general <h2> style, there still may be conflict in other areas of the document. What happens with an <h2> that has been styled with “#main”?

Every selector is given a specificity which is calculated on the entire selector. More specific selectors will override more general ones. The calculation is based on the following (simplified) rules:

- a. Count the number of id's in the overall selector
- b. Count the number of other selectors in the overall selector
- c. Count the number of elements within the overall selector

These three calculations are then written out as a-b-c. Using the examples above, the following declarations can be scored as:

h2

```
{  
    font-size: 1.5em;  
    margin: .83em 0  
    color: black;  
}
```

0-0-1 > specificity = 1

h2

```
{  
    color: brown !important  
}
```

0-0-1 > specificity = 1

h2

```
{  
    color: green;  
    font-size: 1.2em;
```

```

padding: 10px;
}
0-0-1 > specificity = 1
h2#main
{
    color: red;
}
0-0-1 > specificity = 101
h2.navigation
{
    color: blue;
}
0-0-1 > specificity = 11

```

This means that “h2#main” has the greatest weight, followed by “h2.navigation”.

For general users any <h2> styled with id = “main” will be:

```
color: red;          /* id more specific than type selector */
```

Any <h2> styled with class = “navigation” will be:

```
color: blue;        /* class more specific than type selector */
```

If there was a clash between “#main” and “.navigation” the ID would win and the <h2> will be:

```
color: red;          /* id more specific than class or type selector */
```

4. Sort by order specified

If two rules have the same weight, origin and specificity, the one written lower down in the style sheet wins. Rules within imported style sheets are considered to be before any rules in the style sheet itself.

For example, the style sheet could contain two identical rules in various sections of the document:

```
h2
{
    color: blue;
}
```

```
h2
{
    color: green;
}
```

The second rule, that appears later in the style sheet would win.

8.3 BROWSER CONFLICTS

Different operating systems have different default browsers. Mac computers have Safari and PC has Internet Explorer. Yet there are so many other browsers out there that are competing with the popularity of these two.

Firefox has captured a large number of the Internet users' attention by offering an open-source platform, stronger security, and add-ons. Statistics show that people still tend to try other browsers like Opera and Chrome too.

Web Pages Look Differently in Various Browsers: Each web browser looks at the web page code and tries to interpret how the page should be displayed. Some browsers will display the page exactly as the designer intended and others won't.

Web browsers do not render the design of a page pixel by pixel. What they do is to look at the page and render it based on the rules that have been set by the designer and the defaults that have been set by the browser programmers. Some older browsers have difficulty validating the version of the XHTML and CSS coding.

Minimizing Browser Conflicts

STEP 1: Insert a DOCTYPE

DOCTYPE (document type declaration) informs the browser which version of XHTML you're using. Insert the following tag at the very top, **before the <html> tag**.

```
<!DOCTYPE html>
```

STEP 2: Apply Zero Margins and Padding

Most Web browsers have different default settings for the base margins and padding. This means you could get inconsistent results on the page depending upon which browser you're using to view the page. The best way to solve this is to set all the margins and padding on the html and body tags to zero:

```
*{  
    margin:0;  
    padding:0;  
}
```

STEP 3: Use a Container <div>

Always use a container <div> tag and put all other elements inside this container.

8.4 RESOLVING CONFLICTS

When we use multiple sources for scripts to create a website, the possibility of conflicts between codes increase significantly. It should not deter you using multiple vendor/programmers for your website because the advantages that come with them are too large to be ignored.

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

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

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

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

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

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

Because the h1, h2, and h3 selectors are in the source order after the body rule, the headings would display in Arial, not in Georgia.

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

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

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

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

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

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

}

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

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

1. Browser style rules
2. External style rules
3. Internal style (in the document head) rules
4. Inline style rules
5. Individual user style rules

In the list, the browser style rules are the most “distant” from the HTML page, the individual user styles are the “closest.” Within this cascade of style declarations, the closest rule wins. An inline style overrules an external style, which overrules a browser style.

8.5 CONFLICTS IN CSS

It is possible to have CSS declarations in both an external style sheet and an embedded style sheet. The order of the cascade specifies that the embedded styles will override conflicts with the external styles. It is important to understand that only those styles that are in direct conflict will be overridden. The browser will attempt to use both the external and embedded styles together.

For example, you have the following declaration in your external style sheet:

```
p
{
    font-family: Arial, Helvetical, sans-serif;
    color: #00F;
    font-size: 90%;
    line-height: 1.4em;
}
```

In an embedded style sheet on a page linked to that external sheet, you have:


```
p
{
    font-weight: bold;
    line-height: 1.2em;
}
```

The resulting paragraph text on the page would be displayed in bold, blue Arial at 90% with a line height of 1.2em. As the embedded style sheet is silent on the font-family, color and size, those properties are not overridden.

Cascade: The cascade in style sheets is a hierarchy of application.

Example: If you have an inline style for a paragraph, an embedded style for a paragraph, and a linked style for a paragraph, which one will take precedence over another?

The inline styles take precedence over the embedded styles, which take precedence over the linked styles. The cascade appears is in the ordering of multiple style sheets within a document and there are three style sheets linked to the document.

Example

```
<link rel="stylesheet" type="text/css" href="molly1.css" />
<link rel="stylesheet" type="text/css" href="molly2.css" />
<link rel="stylesheet" type="text/css" href="molly3.css" />
```

The rules of the cascade say that whichever is last in the list is the style that takes precedence. So, styles in molly3.css will take precedence over molly2.css and so on.

If you're struggling to figure out why your h2 element isn't turning blue, the reason might be because that style is in conflict with another h2 style that appears in a style sheet that takes precedence within the hierarchy.

Inheritance: Inheritance is the relationship of parent elements to child elements. So, if you're finding conflicts in ground-up CSS design, be sure to look at how the styles for a parent element might be influencing the style of its children or grandchildren. So, have a well-structured document.

Specificity: Specificity is the weight or importance of a given element and is calculated in a fairly complex way. If an element is higher in specificity (due to these calculations), the style associated with that element is what will be applied.

An example of this is that elements with an id will have the highest specificity because by their very nature, ids are extremely specific—they apply to one element within the document only.

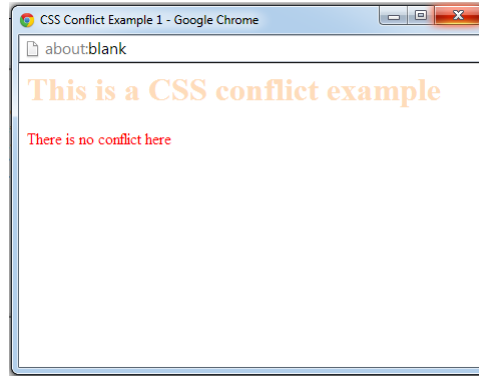
Therefore, styles for an id will take precedence over other styles you attempt to apply to that element.

8.6 RESOLVING CONFLICT IN CSS STYLE RULES

We can resolve conflicts in several different ways depending on what is causing the problem. CSS specific conflict resolution methods. It is widely used language for styling of web based text and media. It is so popular, one cannot think of a single major website that is completely CSS free. The CSS conflicts are easy to resolve because CSS web standards are very flexible. For example, if you would like to change a condition in a styles sheet, you may not have to edit the original file. Instead, you may be able to inject the change from outside.

Consider the CSS rules shown below. What will be the color of the text in the header and the paragraph? Click the button to see if you were right.

```
<html>
  <head>
    <title> CSS Conflict Example 1</title>
    <style type="text/css"><!--
      body {color: red ;}
      h1 {color: #fedcba; }
      →</style>
  </head>
  <body>
    <h1> This is a CSS conflict example</h1>
    <p> There is no conflict here</p>
  </body>
</html>
```



A conflict arises in CSS when an element has a property set by two or more rules. In the above example, above, both the body and the h1 elements set the text color.

The all-important question is: How are conflicts resolved in CSS?

There are three mechanisms involved that apply in these cases.

- Inheritance – an element inherits properties from its containing element
- Specificity – the importance of a rule according to certain criteria
- The Cascade – the order of declaration

In CSS 2.1, the cascade rules for deciding how to display an element in a document are as follows;

1. Find all declarations that contain a selector matching the element
2. Sort the declarations by explicit weight – rules marked

!important

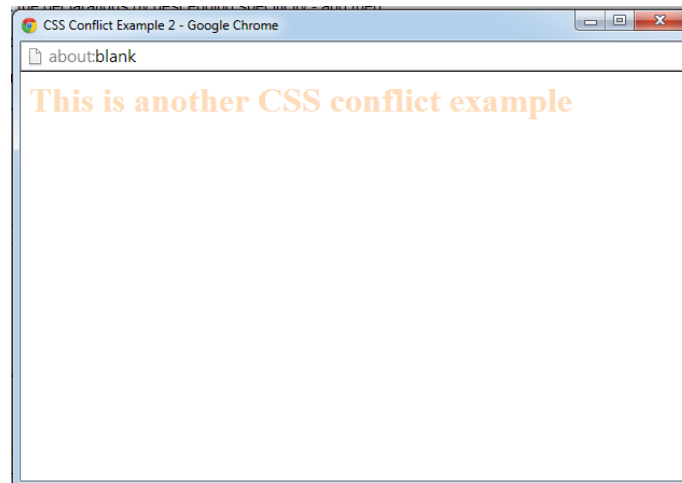
are given a higher weight than those that are not – and then...

3. Sort the declarations by descending specificity – and then...
4. Sort the declarations by order of appearance in the document – later is better

As an example of the cascade in operation, consider the example below:

```
<html>
  <head>
    <title> CSS Conflict Example 2</title>
    <style type="text/css"><!--
      h1 {color: red;}
      h1 {color: red;} →
    </style>
```

```
</head>
<body>
  <h1> This is another CSS conflict example</h1>
</body>
</html>
```



In the example, above, both the h1 element in the document has two declarations that apply to it. They are of equal importance and equal specificity. Therefore, the “order of declaration” rule applies, which means that the later declaration wins and is applied to the element.

8.7 SUMMARY

Conflict in the workplace can be incredibly destructive to good teamwork. Managed in the wrong way, real and legitimate differences between people can quickly spiral out of control, resulting in situations where co-operation breaks down and the team's mission is threatened. This is particularly the case where the wrong approaches to conflict resolution are used. To calm these situations down, it helps to take a positive approach to conflict resolution, where discussion is courteous and non-confrontational, and the focus is on issues rather than on individuals. If this is done, then, as long as people listen carefully and explore facts, issues and possible solutions properly, conflict can often be resolved effectively.

8.8 KEYWORDS

CSS – CASCADING STYLE SHEET

8.9 UNIT END EXERCISE AND ANSWER

- 1) What is Conflict resolution? What happens when conflicts occurs? Explain
- 2) Explain in detail about Browser conflicts and resolving the conflicts.
- 3) How does a conflict occur in CSS? Explain in detail and how resolving conflicts takes place in CSS?

8.10 SUGGESTED READING

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: WebProgrammingBuilding Internet Applications, 3rd Edition, Wiley India, 2006.
4. XueBai et al: The Web Warrior Guide to Web Programming, Thomson, 2003.

MODULE - 3

UNIT 9-: The Basics of JavaScript

Structure

- 9.0 Learning Objectives
- 9.1 Introduction
- 9.2 Overview of JavaScript
- 9.3 Object orientation and JavaScript
- 9.4 General syntactic characteristics
- 9.5 Primitives, operations, and expressions
- 9.6 Summary
- 9.7 Keywords
- 9.8 Exercises
- 9.9 References

9.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Understand about Basics of JavaScript
- Advantage and limitation of JavaScript
- The concept of **General syntactic characteristics**
- String properties and method
- Expression, different types of data object

9.1. Introduction

JavaScript is the premier client-side interpreted scripting language. It's widely used in tasks ranging from the validation of form data to the creation of complex user interfaces. Dynamic

HTML is a combination of the content formatted using HTML, CSS, Scripting language and DOM. By combining all of these technologies, we can create interesting and interactive websites.

9.2 Overview of JavaScript

This section discusses the origins of JavaScript, a few of its characteristics, and some of its uses. Included are a comparison of JavaScript and Java and a brief introduction to event-driven programming.

JavaScript, which was developed by Netscape, was originally named Mocha but soon was, renamed LiveScript. In late 1995 LiveScript became a joint venture of Netscape and Sun Microsystems, and its name again was changed, this time to JavaScript. Netscape's JavaScript has gone through extensive evolution, moving from version 1.0 to version 1.5, primarily by adding many new features. A language standard for JavaScript was developed in the late 1990s by the European Computer Manufacturers Association (ECMA) as ECMA-262. This standard has also been approved by the International Standards Organization (ISO) as ISO-16262. The ECMA-262 standard is now in version 3, which corresponds to Netscape's version 1.5 of JavaScript. Microsoft's JavaScript is named JScript. The FireFox 3 (FX3) and Internet Explorer 8 (IE8) browsers both implement languages that conform to ECMA-262 version 3.

What a JavaScript can do?

JavaScript gives web developers a programming language for use in web pages & allows them to do the following:

- JavaScript gives HTML designers a programming tool
- JavaScript can be used to validate data
- JavaScript can read and write HTML elements
- Create pop-up windows
- Perform mathematical calculations on data
- React to events, such as a user rolling over an image or clicking a button
- Retrieve the current date and time from a user's computer or the last time a document was modified
- Determine the user's screen size, browser version, or screen resolution

- JavaScript can put dynamic text into an HTML page
- JavaScript can be used to create cookies

Advantages of JavaScript:

- Less server interaction
- Immediate feedback to the visitors
- Increased interactivity
- Richer interfaces
- Web surfers don't need a special plug-in to use your scripts
- Java Script is relatively secure.

Limitations of JavaScript:

We cannot treat JavaScript as a full-fledged programming language. It lacks some of the important features like:

- Client-side JavaScript does not allow the reading or writing of files. This has been kept for security reason.
- JavaScript cannot be used for networking applications because there is no such support available.
- JavaScript doesn't have any multithreading or multiprocess capabilities.
- If your script doesn't work then your page is useless.

Applications of JavaScript

The original goal of JavaScript was to provide programming capability at both the server and the client ends of a Web connection. Since then, JavaScript has grown into a full-fledged programming language that can be used in a variety of application areas. As stated earlier, this book focuses on client-side JavaScript. Client-side JavaScript can serve as an alternative for some of what is done with server-side programming, in which computational capability resides on the server and is requested by the client. Because client-side JavaScript is embedded in XHTML documents (either physically or logically) and is interpreted by the browser, this transfer of load from the often overloaded server to the often under loaded client can obviously benefit other clients. Client-side JavaScript cannot replace all server-side computing,

however. In particular, although server-side software supports file operations, database access, and networking, client-side JavaScript supports none of these.

JavaScript and HTML Page

Having written some JavaScript, we need to include it in an HTML page. We can't execute these scripts from a command line, as the interpreter is part of the browser. The script is included in the web page and run by the browser, usually as soon as the page has been loaded. The browser is able to debug the script and can display errors.

Embedding JavaScript in HTML file

If we are writing small scripts, or only use our scripts in few pages, then the easiest way is to include the script in the HTML code. The syntax is shown below:

```
<html>

<head>

<script language="javascript">

<!-- -

Javascript code here

//-- - >

</head>

<body>

.....

</body>

</html>
```

Browsers and XHTML/JavaScript Documents

If an XHTML document does not include embedded scripts, the browser reads the lines of the document and renders its window according to the tags, attributes, and content it finds. When a JavaScript script is encountered in the document, the browser uses its JavaScript interpreter to “execute” the script. Output

from the script becomes the next markup to be rendered. When the end of the script is reached, the browser goes back to reading the XHTML document and displaying its content. There are two different ways to embed JavaScript in an XHTML document: implicitly and explicitly. In explicit embedding, the JavaScript code physically resides in the XHTML document. This approach has several disadvantages. First, mixing two completely different kinds of notation in the same document makes the document difficult to read. Second, in some cases, the person who creates and maintains the XHTML is distinct from the person who creates and maintains the JavaScript. Having two different people doing two different jobs working on the same document can lead to many problems. To avoid these problems, the JavaScript can be placed in its own file, separate from the XHTML document. This approach, called implicit embedding, has the advantage of hiding the script from the browser user. It also avoids the problem of hiding scripts from older browsers, a problem that is discussed later in this section. Except for the chapter's first simple example, which illustrates explicit embedding of JavaScript in an XHTML document, all of the JavaScript example scripts in this chapter are implicitly embedded.

9.3 Object orientation and JavaScript

JavaScript is not an object-oriented programming language. Rather, it is an object-based language. JavaScript does not have classes. Its objects serve both as objects and as models of objects. Without classes, JavaScript cannot have class-based inheritance, which is supported in object-oriented languages such as C++ and Java. It does, however, support a technique that can be used to simulate some of the aspects of inheritance. This is done with the prototype object; thus, this form of inheritance is called prototype-based inheritance (not discussed in this book). Without class-based inheritance, JavaScript cannot support polymorphism. A polymorphic variable can reference related methods of objects of different classes within the same class hierarchy. A method call through such a polymorphic variable can be dynamically bound to the method in the object's class.¹ Despite the fact that JavaScript is not an object-oriented language, much of its design is rooted in the concepts and approaches used in object-oriented programming. Specifically, client-side JavaScript deals in large part with documents and document elements, which are modeled with objects.

JavaScript Objects

In JavaScript, objects are collections of properties, which correspond to the members of classes in Java and C++. Each property is either a data property or a function or method property. Data properties appear in two categories: primitive values and references to other objects. (In JavaScript, variables that refer to objects are often called objects rather than references.) Sometimes we will refer to the data properties

simply as properties; we often refer to the method properties simply as methods or functions. We prefer to call subprograms that are called through objects methods and subprograms that are not called through objects functions. JavaScript uses no object types for some of its simplest types; these non-object types are called primitives. Primitives are used because they often can be implemented directly in hardware, resulting in faster operations on their values (faster than if they were treated as objects). Primitives are like the simple scalar variables of non-object-oriented languages such as C, C++, Java, and JavaScript all have both primitives and objects; All objects in a JavaScript program are indirectly accessed through variables. Such a variable is like a reference in Java. All primitive values in JavaScript are accessed directly—these are like the scalar types in Java and C++. These are often called value types. The properties of an object are referenced by attaching the name of the property to the variable that references the object. For example, if myCar is a variable referencing an object that has the property engine, the engine property can be referenced with myCar.engine.

The root object in JavaScript is Object. It is the ancestor, through prototype inheritance, of all objects. Object is the most generic of all objects, having some methods but no data properties. All other objects are specializations of Object, and all inherit its methods (although they are often overridden). JavaScript object appears, both internally and externally, as a list of property–value pairs. The properties are names; the values are data values or functions. All functions are objects and are referenced through variables. The collection of properties of a JavaScript object is dynamic: Properties can be added or deleted at any time. Every object is characterized by its collection of properties, although objects do not have types in any formal sense. Recall that Object is characterized by having no properties

9.4 General syntactic characteristics

JavaScript scripts are embedded, either directly or indirectly, in XHTML documents. Scripts can appear directly as the content of a <script> tag. The type attribute of <script> must be set to “text/javascript”. The JavaScript script can be indirectly embedded in an XHTML document with the src attribute of a <script> tag, whose value is the name of a file that contains the script for example,

```
<script type = “text/javascript” src = “tst_number.js” >  
  
</script>
```

Notice that the script element requires the closing tag, even though it has no content when the src attribute is included. There are some situations when a small amount of JavaScript code is embedded in an XHTML document. Furthermore, some documents have a large number of places where JavaScript code

is embedded. Therefore, it is sometimes inconvenient and cumbersome to place all JavaScript code in a separate file. In JavaScript, identifiers, or names, are similar to those of other common programming languages. They must begin with a letter, an underscore (_), or a dollar sign (\$).³ Subsequent characters may be letters, underscores, dollar signs, or digits. There is no length limitation for identifiers. As with most C-based languages, the letters in a variable name in JavaScript are case sensitive, meaning that FRIZZY, Frizzy, FrIzZy, frizzy, and friZZy are all distinct names. However, by convention, programmer-defined variable names do not include uppercase letters. JavaScript has 25 reserved words, which are listed in below table 9.1

break	delete	function	return	typeof
case	do	if	switch	var
catch	else	in	this	void
continue	finally	instanceof	throw	while
default	for	new	try	with

Table 9.1: JavaScript reserved words

Besides its reserved words, another collection of words is reserved for future use in JavaScript—these can be found at the ECMA Web site. In addition, JavaScript has a large collection of predefined words, including alert, open, java, and self. JavaScript has two forms of comments, both of which are used in other languages. First, whenever two adjacent slashes (//) appear on a line, the rest of the line is considered a comment. Second, /* may be used to introduce a comment, and */ to terminate it, in both single- and multiple-line comments. Two issues arise regarding embedding JavaScript in XHTML documents. First, some browsers that are still in use recognize the <script> tag but do not have JavaScript interpreters. Fortunately, these browsers simply ignore the contents of the script element and cause no problems. Second, a few browsers that are still in use are so old that they do not recognize the <script> tag. Such browsers would display the contents of the script element as if it were just text. It has been customary to enclose the contents of all script elements in XHTML comments to avoid this problem. Because there are so few browsers that do not recognize the <script> tag, we believe that the issue no longer exists. However, the XHTML validator also has a problem with embedded JavaScript. When the embedded JavaScript happens to include recognizable tags—for example
 tags in the output of the JavaScript—these tags often cause validation errors. Therefore, we still enclose embedded JavaScript in XHTML comments when we explicitly embed JavaScript.

The XHTML comment used to hide JavaScript uses the normal beginning syntax, `<!--`. However, the syntax for closing such a comment is special. It is the usual XHTML comment closer, but it must be on its own line and must be preceded by two slashes (which makes it a JavaScript comment). The following XHTML comment form hides the enclosed script from browsers that do not have JavaScript interpreters, but makes it visible to browsers that do support JavaScript:

```
<!--  
-- JavaScript script --  
// -->
```

There are other problems with putting embedded JavaScript in comments in XHTML documents. “Dynamic Documents with JavaScript.” The best solution to all of these problems is to put all JavaScript scripts that are of significant size in separate files and embed them implicitly. The use of semicolons in JavaScript is unusual. The JavaScript interpreter tries to make semicolons unnecessary, but it does not always work. When the end of a line coincides with what could be the end of a statement, the interpreter effectively inserts a semicolon there. But this can lead to problems. For example, consider the following lines of code:

```
return x;
```

The interpreter will insert a semicolon after `return`, making `x` an invalid orphan. The safest way to organize JavaScript statements is to put each on its own line whenever possible and terminate each statement with a semicolon. If a statement does not fit on a line, be careful to break the statement at a place that will ensure that the first line does not have the form of a complete statement.

In the following complete, but trivial, XHTML document that simply greets the client who requests it, there is but one line of JavaScript—the call to write through the document object to display the message

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- hello.html
    A trivial hello world example of XHTML/JavaScript
-->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Hello world </title>
  </head>
  <body>
    <script type = "text/javascript">
      <!--
        document.write("Hello, fellow Web programmers!");
      // -->
    </script>
  </body>
</html>
```

9.5 Primitives, operations, and expressions

The primitive data types, operations, and expressions of JavaScript are similar to those of other common programming languages. Therefore, our discussion of them is brief.

Primitive Types

JavaScript has five primitive types: Number, String, Boolean, Undefined, and Null.⁵ Each primitive value has one of these types. JavaScript includes predefined objects that are closely related to the Number, String, and Boolean types, named Number, String, and Boolean, respectively. These objects are called wrapper objects. Each contains a property that stores a value of the corresponding primitive type. The purpose of the wrapper objects is to provide properties and methods that are convenient for use with values of the primitive types. In the case of Number, the properties are more useful; in the case of String, the methods are more useful. Because JavaScript coerces values between the Number type primitive values and Number objects and between the String type primitive values and String objects, the methods of Number and String can be used on variables of the corresponding primitive types. In fact, in most cases you can simply treat Number and String type values as if they were objects.

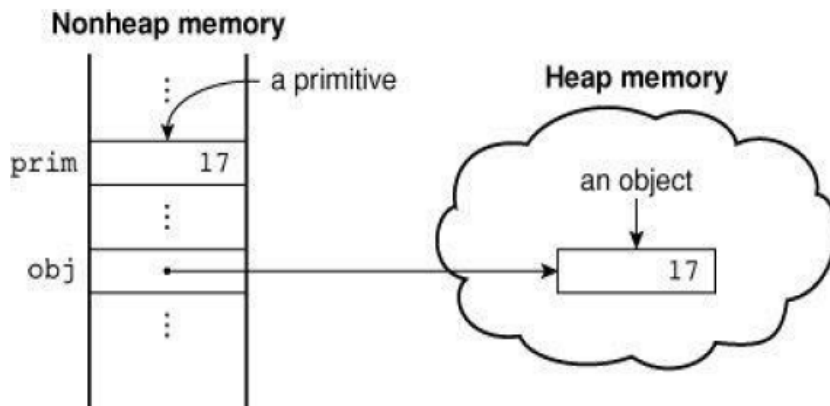


Figure 4.1 Primitives and objects

Numeric and String Literals

All numeric literals are values of type Number. The Number type values are represented internally in double-precision floating-point form. Because of this single numeric data type, numeric values in JavaScript are often called numbers. Literal numbers in a script can have the forms of either integer or floating-point values. Integer literals are strings of digits. Floating-point literals can have decimal points, exponents, or both. Exponents are specified with an uppercase or lowercase e and a possibly signed integer literal. The following are valid numeric literals:

72 7.2 .72 72. 7E2 7e2 .7e2 7.e2 7.2E-2

Integer literals can be written in hexadecimal form by preceding their first digit with either 0x or 0X. (The first character is zero, not “oh”.) A string literal is a sequence of zero or more characters delimited by either single quotes (‘) or double quotes (“). String literals can include characters specified with escape sequences, such as \n and \t. If you want an actual single-quote character in a string literal that is delimited by single quotes, the embedded single quote must be preceded by a backslash: ‘You\’re the most freckly I\’ve ever met’ A double quote can be embedded in a double-quoted string literal by preceding it with a backslash. An actual backslash character in any string literal must be itself backslashed, as in the following example:

“D:\\bookfiles”

There is no difference between single-quoted and double-quoted literal strings. The null string (a string with no characters) can be denoted with either “or”. All string literals are primitive values.

Other Primitive Types

The only value of type Null is the reserved word `null`, which indicates no value. A variable is null if it has not been explicitly declared or assigned a value. If an attempt is made to use the value of a variable whose value is null, it will cause a runtime error. The only value of type Undefined is undefined. Unlike null, there is no reserved word undefined. If a variable has been explicitly declared, but not assigned a value, it has the value undefined. If the value of an undefined variable is displayed, the word “undefined” is displayed. The only values of type Boolean are true and false. These values are usually computed as the result of evaluating a relational or Boolean expression. The existence of both the Boolean primitive type and the Boolean object can lead to some confusion

Declaring Variables

One of the characteristics of JavaScript that sets it apart from most common non-scripting programming languages is that it is dynamically typed. This means that a variable can be used for anything. Variables are not typed; values are. A variable can have the value of any primitive type, or it can be a reference to any object. The type of the value of a particular appearance of a variable in a program can be determined by the interpreter. In many cases, the interpreter converts the type of a value to whatever is needed for the context in which it appears. A variable can be declared either by assigning it a value, in which case the interpreter implicitly declares it to be a variable, or by listing it in a declaration statement that begins with the reserved word `var`. Initial values can be included in a `var` declaration, as with some of the variables in the following declaration:

```
var counter,
    index,
    pi = 3.14159265,
    quarterback = "Elway",
    stop_flag = true;
```

Numeric Operators

JavaScript has the typical collection of numeric operators: the binary operators `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for modulus. The unary operators are plus (`+`), negate (`-`), decrement (`--`), and increment (`++`). The increment and decrement operators can be either

prefix or postfix. As with other languages that have the increment and decrement unary operators, the prefix and postfix uses are not always equivalent. Consider an expression consisting of a single variable and one of these operators. If the operator precedes the variable, the value of the variable is changed and the expression evaluates to the new value. If the operator follows the variable, the expression evaluates to the current value of the variable and then the value of the variable is changed. For example, if the variable **a** has the value **7**, the value of the following expression is **24**:

(++a) * 3

But the value of the following expression is **21**:

(a++) * 3

In both cases, **a** is set to **8**. All numeric operations are done in double-precision floating point. The precedence rules of a language specify which operator is evaluated first when two operators with different precedence are adjacent in an expression.

Adjacent operators are separated by a single operand. For example, in the following code, ***** and **+** are adjacent:

a * b + 1

The associativity rules of a language specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression. The precedence and associativity of the numeric operators of JavaScript are given in Table 9.2.

Operator	Associativity
++, --, unary -, unary +	Right (though it is irrelevant)
*, /, %	Left
Binary +, binary -	Left

Table 9.2: Precedence and associativity of Numeric operators

The first operators listed have the highest precedence. As examples of operator precedence and associativity, consider the following code:

```
var a = 2,  
    b = 4,  
    c,  
    d;  
c = 3 + a * b;  
// * is first, so c is now 11 (not 24)  
d = b / a / 2;  
// / associates left, so d is now 1 (not 4)
```

Parentheses can be used to force any desired precedence. For example, the addition will be done before the multiplication in the following expression:

$(a + b) * c$

MATH OBJECT

For performing the mathematical computations there are some useful methods available from math object.

`sqrt(num)` `abs(num)` `ceil(num)` `floor(num)` `log(num)` `pow(a,b)` `min(a,b)` `max(a,b)`

`sin(num)` `cos(num)` `tan(num)` `exp(num)` `asin(value)` `acos(value)` `atan(value)`

`random()` ----> returns a psuedorandom number between 1 to 1

`round (value)`

In addition to the above methods, it has several properties (Numeric constants) like:

`Math.E`----->Euler constant

`Math.PI`-----> 3.14159

`Math.SQRT_2`----->The square root of 2

`Math.SQRT12`----->The square root of ½

`Math.LN2`-----> Log of 2

Math.LN10 -----> Log of 10

Example:

```
<html>

<body>

<script language="javascript">

var n = prompt("enter any number");

alert("square root is "+Math.sqrt(n));

</script>

</body>

</html>
```

Exercise1:

1. Write a JavaScript program that generates the following table for the given value of n

Number Square

1 1

2 4

3 9

4 16

```
<html>
```

```
<body>
```

```
<table border=1>
```

```

<tr> <th>Number <th>Square </tr>

<script language="javascript">

var n = prompt("enter n");

for(i=1;i<=n;i++)

{

document.write("<tr><td>" + i + "<td>" + (i*i) + "</tr>");

}

</script>

</table>

</body>

</html>

```

String Properties and Methods

Because JavaScript coerces primitive string values to and from String objects when necessary, the differences between the String object and the String type have little effect on scripts. String methods can always be used through String primitive values, as if the values were objects. The String object includes one

property, length, and a large collection of methods. The number of characters in a string is stored in the length property as follows:

```

var str = "George";

var len = str.length;

```

Assignment Statements

The assignment statement in JavaScript is exactly like the assignment statement in other common C-based programming languages. There is a simple assignment operator, denoted by `=`, and a host of compound assignment operators, such as `+=` and `/=`. For example, the statement

```
a += 7;
```

means the same as

```
a = a + 7;
```

In considering assignment statements, it is important to remember that JavaScript has two kinds of values: primitives and objects. A variable can refer to a primitive value, such as the number 17, or an object, as shown in Figure 9.1. Objects are allocated on the heap, and variables that refer to them are essentially reference variables. When used to refer to an object, a variable stores an address only. Therefore, assigning the address of an object to a variable is fundamentally different from assigning a primitive value to a variable.

The Date Object

There are occasions when information about the current date and time is useful in a program. Likewise, sometimes it is convenient to be able to create objects that represent a specific date and time and then manipulate them. These capabilities are available in JavaScript through the `Date` object and its rich collection of methods. In what follows, we describe this object and some of its methods. A `Date` object is created with the `new` operator and the `Date` constructor, which has several forms. Because we focus on uses of the current date and time, we use only the simplest `Date` constructor, which takes no parameters and builds an object with the current date and time for its properties. For example, we might have `var today = new Date();`

The date and time properties of a `Date` object are in two forms: local and Coordinated Universal Time (UTC, which was formerly named Greenwich Mean Time). We deal only with local time in this section. Table 9.5 shows the methods, along with the descriptions, that retrieve information from a `Date` object.

Method	Returns
<code>toLocaleString</code>	A string of the Date information
<code>getDate</code>	The day of the month
<code>getMonth</code>	The month of the year, as a number in the range from 0 to 11
<code>getDay</code>	The day of the week, as a number in the range from 0 to 6
<code>getFullYear</code>	The year
<code>getTime</code>	The number of milliseconds since January 1, 1970
<code>getHours</code>	The number of the hour, as a number in the range from 0 to 23
<code>getMinutes</code>	The number of the minute, as a number in the range from 0 to 59
<code>getSeconds</code>	The number of the second, as a number in the range from 0 to 59
<code>getMilliseconds</code>	The number of the millisecond, as a number in the range from 0 to 999

Table 9.3: Methods for the Date object

9.6 SUMMARY

JavaScript is the premier client-side interpreted scripting language. It's widely used in tasks ranging from the validation of form data to the creation of complex user interfaces. Dynamic HTML is a combination of the content formatted using HTML, CSS, Scripting language and DOM. By combining all of these technologies, we can create interesting and interactive websites.

9.7 KEY WORDS

Corporate Planning, Growth Strategy, Market Strategy, Corporate Strategy, Strategic analysis of business, Drivers of strategic business

9.8 EXERCISE

9.1 Describe briefly three major differences between Java and JavaScript.

9.2 Describe briefly three major uses of JavaScript on the client side.

9.3 Describe briefly the basic process of event-driven computation.

9.4 What are the two categories of properties in JavaScript?

9.5 Why does JavaScript have two categories of data variables, namely, primitives and objects?

9.6 Describe the two ways to embed a JavaScript script in an XHTML document.

9.7 What are the two forms of JavaScript comments?

9.8 Why are JavaScript scripts sometimes hidden in XHTML documents by putting them into XHTML comments?

9.9 What are the five primitive data types in JavaScript?

9.9 References:

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.

2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.

3. Chris Bates: Web Programming Building Internet Applications, 3rd Edition, Wiley India, 2006.

UNIT -10:

Structure

- 10.0 Learning Objectives
- 10.1 Screen output and keyboard input
- 10.2 Control statements
- 10.3 Arrays
- 10.4 Object creation and modification
- 10.5 Summary
- 10.6 Keywords
- 10.7 Exercises
- 10.8 References

10.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Understand differences between Java and JavaScript
- Identify different types of control statements
- Understand the basic concept of Array
- Understand Object creation and modification
- Identify object-based array function

10.1. Screen output and keyboard input

A JavaScript script is interpreted when the browser finds the script or a reference to a separate script file in the body of the XHTML document. Thus, the normal output screen for JavaScript is the same as the screen in which the content of the host XHTML document is displayed. JavaScript models the XHTML document with the Document object. The window in which the browser displays an XHTML document is modelled with the Window object. The Window object includes two properties, document and window. The document property refers to the Document object. The window property is self-referential; it refers to the Window object. The Document object has several properties script output, which is dynamically created XHTML document content. This content is specified in the parameter of write. For example, if the value

of the variable `script` output, which is dynamically created XHTML document content. This content is specified in the parameter of `write`. For example, if the value of the variable `result` is 42, the following statement produces the screen shown in Figure 4.2: `document`.

```
write ("The result is: ", result, "<br />");
```



Figure 4.2 An example of the output of `document.write`

Because `write` is used to create XHTML code, the only useful punctuation in its parameter is in the form of XHTML tags. Therefore, the parameter of `write` often includes `
`. The `writeln` method implicitly adds `"\n"` to its parameter, but since browsers ignore line breaks when displaying XHTML, it has no effect on the output. The parameter of `write` can include any XHTML tags and content. The parameter is simply given to the browser, which treats it exactly like any other part of the XHTML document. The `write` method actually can take any number of parameters. Multiple parameters are concatenated and placed in the output. As stated previously, the `Window` object is the JavaScript model for the browser window. `Window` includes three methods that create dialog boxes for three specific kinds of user interactions. The default object for JavaScript is the `Window` object currently being displayed, so calls to these methods need not include an object reference. The three methods—`alert`, `confirm`, and `prompt`—which are described in the following paragraphs, are used primarily for debugging rather than as part of a servable document.

The `alert` method opens a dialog window and displays its parameter in that window. It also displays an OK button. The string parameter of `alert` is not XHTML code; it is plain text. Therefore, the string parameter of `alert` may include `\n` but never should include `
`. As an example of an alert, consider the following code, in which we assume that the value of `sum` is 42.

```
alert ("The sum is:" + sum + "\n");
```

This call to `alert` produces the dialog window shown in Figure 4.3.



Figure 4.3 An example of the output of `alert`

The `confirm` method opens a dialog window in which the method displays its string parameter, along with two buttons: OK and Cancel. `confirm` returns a Boolean value that indicates the user's button input: `true` for OK and `false` for Cancel. This method is often used to offer the user the choice of continuing some process. For example, the following statement produces the screen shown in Figure 4.4:

```
var question = confirm("Do you want to continue this download?");
```

After the user presses one of the buttons in the `confirm` dialog window, the script can test the variable, `question`, and react accordingly.



Figure 4.4 An example of the output of `confirm`

The `prompt` method creates a dialog window that contains a text box used to collect a string of input from the user, which `prompt` returns as its value. As with `confirm`, this window also includes two buttons: OK and Cancel. `prompt` takes two parameters: the string that prompts the user for input and a default string in case the user does not type a string before pressing one of the two buttons. In many cases, an empty string is used for the default input. Consider the following

example:

```
name = prompt("What is your name?", "");
```

Figure 4.5 shows the screen created by this call to `prompt`. `alert`, `prompt`, and `confirm` cause the browser to wait for a user response. In the case of `alert`, the OK button must be pressed for the JavaScript interpreter to continue. The `prompt` and `confirm` methods wait for either OK or Cancel to be pressed. The next two example XHTML and JavaScript files `roots.html` and `roots.js` illustrate some of the JavaScript features described so far. The JavaScript script gets the coefficients of a quadratic equation from the user with `prompt` and computes and displays the real roots of the given equation. If the roots of the equation are not real, the value NaN is displayed. This value comes from the `sqrt` function, which returns NaN when the function is given a negative parameter. This result corresponds mathematically to the equation not having real roots.

```

<?xml version = "1.0" encoding = "utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- roots.html
    A document for roots.js
-->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> roots.html </title>
  </head>
  <body>
    <script type = "text/javascript" src = "roots.js" >
    </script>
  </body>
</html>

// roots.js
// Compute the real roots of a given quadratic
// equation. If the roots are imaginary, this script
// displays NaN, because that is what results from
// taking the square root of a negative number

// Get the coefficients of the equation from the user
var a = prompt("What is the value of 'a'? \n", "");
var b = prompt("What is the value of 'b'? \n", "");
var c = prompt("What is the value of 'c'? \n", "");

// Compute the square root and denominator of the result
var root_part = Math.sqrt(b * b - 4.0 * a * c);
var denom = 2.0 * a;

// Compute and display the two roots
var root1 = (-b + root_part) / denom;
var root2 = (-b - root_part) / denom;
document.write("The first root is: ", root1, "<br />");
document.write("The second root is: ", root2, "<br />");

```

In the examples in the remainder of this chapter, the XHTML document that uses the associated JavaScript file is not shown.

10.2. Control Statements

This section introduces the flow-control statements of JavaScript. Before discussing the control statements, we must describe control expressions, which provide the basis for controlling the order of execution of statements. Once again, the similarity of these JavaScript constructs to their counterparts in Java and C++ makes them easy to learn for those who are familiar with one of those languages. Control statements often require some syntactic container for sequences of statements whose execution they are meant to control. In JavaScript, that container is the compound statement. A compound statement in JavaScript is a sequence of statements delimited

by braces. A control construct is a control statement together with the statement or compound statement whose execution it controls.

Unlike several related languages, JavaScript does not allow compound statements to create local variables. If a variable is declared in a compound statement, access to it is not confined to that compound statement..

Operator	Description	Example
==	is equal to	5==8 returns false
===	is equal to (checks for both value and type)	x=5 y="5" x==y returns true x===y returns false
!=	is not equal	5!=8 returns true
>	is greater than	5>8 returns false
<	is less than	5<8 returns true
>=	is greater than or equal to	5>=8 returns false
<=	is less than or equal to	5<=8 returns true

Table 10.1: Operator description

The properties of the object Boolean must not be confused with the primitive values true and false. If a Boolean object is used as a conditional expression, it evaluates to true if it has any value other than null or undefined. The Boolean object has a method, toString, which it inherits from Object, that converts the value of the object through which it is called to one of the strings “true” and “false”. The precedence and associativity of all operators discussed so far in this chapter are shown in Table 10.2.

Operator	Description	Example
&&	and	x=6 y=3 (x < 10 && y > 1) returns true
	or	x=6 y=3 (x==5 y==5) returns false
!	not	x=6 y=3 !(x==y) returns true

10.2.1 Selection Statements

The selection statements (if-then and if-then-else) of JavaScript are similar to those of the common programming languages. Either single statements or compound statements can be selected—for example,

```
if (a > b)
    document.write("a is greater than b <br />");
else {
    a = b;
    document.write("a was not greater than b <br />",
        "Now they are equal <br />");
}
```

10.2.2 The switch Statement

JavaScript has a switch statement that is similar to that of Java. The form of this construct is as follows:

```
switch (expression) {
    case value_1:
        // statement(s)
    case value_2:
        // statement(s)
    ...
    [default:
        // statement(s)]
}
```

In any case segment, the statement(s) can be either a sequence of statements or a compound statement. The semantics of a switch construct is as follows: The expression is evaluated when the switch statement is reached in execution. The value is compared with the values in the cases in the construct (those values that immediately follow the case reserved words). If one matches, control is transferred to the statement immediately following that case value. Execution then continues through the remainder of the construct. In the great majority of situations, it is intended that only the statements in one case be executed in each execution of the construct. To implement this approach, a break statement appears as the last statement in each sequence of statements following a case. The break statement is exactly like the break statement in Java and C++: It transfers control out of the compound statement in which it appears. The control expression of a switch statement could evaluate to a number, a string, or a Boolean value. Case labels also can be numbers, strings, or Booleans, and different case values can be of different types. Consider the following script, which includes a switch construct.


```

// borders2.js
//   An example of a switch statement for table border
//   size selection

var bordersize;
bordersize = prompt("Select a table border size \n" +
    "0 (no border) \n" +
    "1 (1 pixel border) \n" +
    "4 (4 pixel border) \n" +
    "8 (8 pixel border) \n");

switch (bordersize) {
    case "0": document.write("<table>");
        break;
    case "1": document.write("<table border = '1'>");
        break;
    case "4": document.write("<table border = '4'>");
        break;
    case "8": document.write("<table border = '8'>");
        break;
    default: document.write("Error - invalid choice: ",
        bordersize, "<br />");
}

document.write("<caption> 2008 NFL Divisional",
    " Winners </caption>");
document.write("<tr>",
    "<th />",

```

```

"<th> American Conference </th>",
"<th> National Conference </th>",
"</tr>",
"<tr>",
"<th> East </th>",
"<td> Miami Dolphins </td>",
"<td> New York Giants </td>",
"</tr>",
"<tr>",
"<th> North </th>",
"<td> Pittsburgh Steelers </td>",
"<td> Minnesota Vikings </td>",
"</tr>",
"<tr>",
"<th> West </th>",
"<td> San Diego Chargers </td>",
"<td> Arizona Cardinals </td>",
"</tr>",
"<tr>",
"<th> South </th>",
"<td> Tennessee Titans </td>",
"<td> Carolina Panthers </td>",
"</tr>",
"</table>";

```

The entire table element is produced with calls to write. Alternatively, we could have given all of the elements of the table, except the <table> and </table> tags, directly as XHTML in the XHTML document. Because <table> is in the content of the script element, the validator would not see it. Therefore, the </table> tag would also need to be hidden. Browser displays of the prompt dialog box and the output of borders2.js are shown in Figures 10.1 and 10.2, respectively.

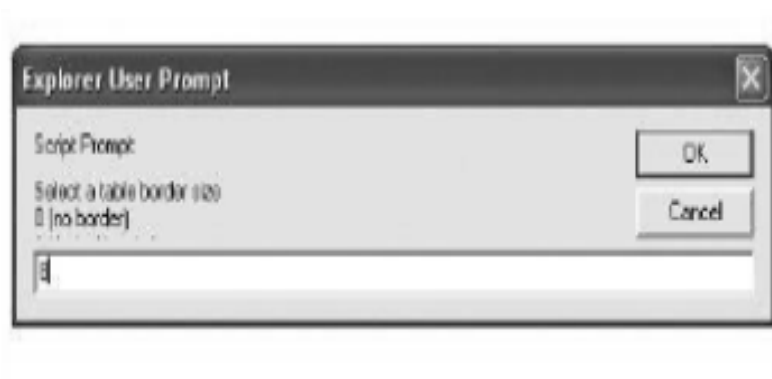


Figure 10.1: Dialogue box from borders2.js

2008 NFL Divisional Winners

	American Conference	National Conference
East	Miami Dolphins	New York Giants
North	Pittsburgh Steelers	Minnesota Vikings
West	San Diego Chargers	Arizona Cardinals
South	Tennessee Titans	Carolina Panthers

Figure 10.2: Display produced by borders2.js

Looping statements: Loops are used to execute certain statements repeatedly. The various loops in JavaScript are:

- **for** loop
- **while** loop
- **do-while** loop

for loop syntax:

for(initialization; condition; loop variable update)

{

Set of statements

}

Example (for loop):

```
<html>
```

```
<body>
```

```
<script language="javascript">
```

```
for (var i = 1; i <= 6; i++)
```

```

{

document.write("<h" + i + ">This is header " + i + "</h" + i + ">");

}

</script>

</body>

</html>

```

for- in loop:

The **for-in** loop has a special use to enumerate all the properties contained within an object. This loop is rarely used in regular JavaScript.

Example: Display **window** object properties

```

var i, a = "";

for( i in window)

a += i + "...";

alert(a);

```

while loop:

Executes the statements as long as the condition is true.

Syntax: while(condition)

```

{

Set of statements

}

```

Example:

```
<html>

<body>

<script language="javascript">

var i=0;

while (i<=10)

{

document.write("The number is " + i);

document.write("<br />");

i=i+1;

}

</script>

</body>

</html>
```

do-while syntax:

```
do

{

Set of statements

}while(condition);
```

break statement:

Break statement is used to terminate a loop, switch, or label statement. When we use break without a label, it terminates the innermost enclosing while, do-while, for, or switch immediately and transfers

control to the following statement. When we use **break** with a label, it terminates the specified labeled statement

Syntax:

- **break;**
- **break *label*;**

Continue Statement:

When we use **continue** without a label, it terminates the current iteration of the innermost enclosing while, do-while or for statement and continues execution of the loop with the next iteration. When we use **continue** with a label, it applies to the looping statement identified with that label.

Syntax:

- **continue;**
- **continue *label*;**

10.3 Arrays

An array is an ordered set of data elements which can be accessed through a single variable name. In many programming languages arrays are contiguous areas of memory which means that the first element is physically located next to the second and so on. In JavaScript, an array is slightly different because it is a special type of object and has functionality which is not normally available in other languages.

Basic Array Functions: The basic operations that are performed on arrays are creation, addition of elements (inserting elements), accessing individual elements, removing elements.

Creating Arrays: We can create arrays in several ways:

- `var arrayObjectName = [element0, element1, ..., element N];`
- `var arrayObjectName = new Array(element0, element1, ..., element N);`
- `var arrayObjectName = new Array(arrayLength);`

Ex:

- `var colors = ["Red", "Green", "Blue"];`
- `var colors = new Array("Red", "Green", "Blue");`
- `var colors = Array("Red", "Green", "Blue");`
- `var thirdArray = [,,,];`
- `var fourthArray = [,,35,,,16,,23,];`

Note: JavaScript arrays can hold mixed data types as the following example shows:

```
var a = ["Monday", 34, 45.7, "Tuesday"];
```

Accessing Array Elements:

Array elements are accessed through their *index*. The **length** property can be used to know the length of the array. The index value runs from **0** to **length-1**.

Example:

```
<script language="javascript">
```

```
var a = [1,2,3];
```

```
var s = "";
```

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

```
{
```

```
s += a[i] + " ";
```

```
}
```

```
alert(s);
```

</script>

Adding elements to an array:

What happens if we want to add an item to an array which is already full? Many languages struggle with this problem. But JavaScript has a really good solution: *the interpreter simply extends the array and inserts the new item.*

Ex: var a = ["vit","svecw","sbsp"];

a[3] = "bvrit";

a[10] = "bvrice"; //this extends the array and the values of elements a[4] to a[9] will be

undefined.

Modifying array elements:

Array element values can be modified very easily.

Ex: To change a[1] value to "vdc" simply write:

a[1] = "vdc";

Searching an Array:

To search an array, simply read each element one by one & compare it with the value that we are looking for.

Removing Array Members:

JavaScript doesn't provide a builtin function to remove array element. To do this, we can use the following approach:

- read each element in the array
- if the element is not the one you want to delete, copy it into a temporary array
- if you want to delete the element then do nothing
- increment the loop counter
- repeat the process

- finally store the temporary array reference in the main array variable

Note: The statement **delete a[0]** makes the value of **a[0]** **undefined**.

10.4 Object creation and modification

In JavaScript, an array is an object. So, we can use various member functions of the object to manipulate arrays.

concat()

The **concat()** method returns the array resulting from concatenating argument arrays to the array on which the method was invoked. The original arrays are unaltered by this process.

Syntax: array1.concat (array2 [, array3,...arrayN]);

Example:

```
<script language="javascript">
```

```
var a = [1,2,3];
```

```
var b = ["a","b"];
```

```
alert(a.concat(b));
```

```
</script>
```

join() method allows to join the array elements as strings separated by given specifier. The original array is unaltered by this process.

Syntax: arrayname.join(separator);

Example:

```
<script language="javascript">
```

```
var a = [1,2,3];
```

```
alert(a.join("#"));
```

```
</script>
```

push() function adds one or more elements to the end of an array and returns the last element added.

Syntax: arrayname.push(element1 [, element2, ..elementN]);

Example:

```
<script language="javascript">
```

```
var a = [1,2,3];
```

```
alert(a.push(4,5)); //displays 5
```

```
alert(a); //displays 1,2,3,4,5
```

```
</script>
```

pop() removes the last element from the array and returns that element

Syntax: arrayname.pop();

reverse() method transposes the elements of an array: the first array element becomes the last and the last becomes the first. The original array is altered by this process.

Syntax: arrayname.reverse();

Example:

```
<script language="javascript">
```

```
var a = [1,2,3];
```

```
a.reverse();
```

```
alert(a);
```

```
</script>
```

shift() removes the first element of the array and in doing so shortens its length by one. It returns the first element that is removed.

Ex: var a = [1, 2, 3];

```
var first = a.shift();
```

```
alert(a); // 2,3
```

```
alert(first); //1
```

unshift() adds one or more elements to the front of an array.

Syntax: arrayname.unshift(element1 [, element2, ..elementN]);

Example:

```
var a = ["x","y","z"];
```

```
a.unshift("p","q");
```

```
alert(a); //p,q,x,y,z
```

slice() returns a “slice” (subarray) of the array on which it is invoked. The method takes two arguments, the *start* and *end* index, and returns an **array** containing the elements from index *vstart* up to but not including index *end*. If we specify only first parameter, then array containing the elements from *start* to the end of the array are returned. The original array is unaltered by this process.

Syntax: arrayname.slice(startindex , endindex);

Example:

```
var a = [1, 2, 3, 4, 5];
```

```
a.slice(2); // returns [3, 4, 5]
```

```
a.slice(1, 3); // returns [2, 3]
```

The **splice()** method can be used to add, replace, or remove elements of an array in place. Any elements that are removed are returned. It takes a variable number of arguments, the first two arguments are mandatory. The original array is altered by this process.

Syntax : `arrayname.splice(start, deleteCount, replacevalues);`

- The first argument *start* is the index at which to perform the operation.
- The second argument is *deleteCount*, the number of elements to delete beginning with index *start*. If we don't want to delete any elements specify this value as 0.
- Any further arguments represented by *replacevalues* (that are comma-separated, if more than one) are inserted in place of the deleted elements.

Example:

```
var myArray = [1, 2, 3, 4, 5];
```

```
myArray.splice(3,2,"a","b"); // returns 4,5
```

```
alert(a); //1,2,3,a,b
```

sort() method sorts the array into lexicographic order. Elements which are not text are converted into strings before the sort operation is performed. This means, for example, 732 will be placed before 80 in the sorted array. Original array is altered by this process.

Example1:

```
var myArray = ["vit","svecw","bvrice"];
```

```
myArray.sort();
```

```
alert(myArray); //bvrice,svecw,vit
```

Example2:

```
var a = [80,732,450];
```

```
a.sort();
```

```
alert(a); //450,732,80
```

10.5 SUMMARY

Normally, statements in a program execute one after the other, in the order in which they are written. This process is called sequential execution.

- Various JavaScript statements enable the programmer to specify that the next statement to be executed may be other than the next one in sequence. This process is called transfer of control.
- All programs can be written in terms of only three control structures, namely, the sequence structure, the selection structure and the repetition structure.
- A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called flowlines, which indicate the order in which the actions of the algorithm execute.
- JavaScript provides three selection structures. The if statement either performs (selects) an action if a condition is true or skips the action if the condition is false. The if...else statement performs an action if a condition is true and performs a different action if the condition is false.

10.5. KEY WORDS

Javascript, data types, syntactic characteristics, characteristics of JavaScript

10.6. EXERCISE

10.1 Describe briefly three major differences between Java and JavaScript.

10.2 Describe briefly three major uses of JavaScript on the client side.

10.3 Describe briefly the basic process of event-driven computation.

10.4 What are the two categories of properties in JavaScript?

10.5 Why does JavaScript have two categories of data variables, namely, primitives and objects?

10.6 Describe the two ways to embed a JavaScript script in an XHTML document.

10.7 What are the two forms of JavaScript comments?

10.8 Why are JavaScript scripts sometimes hidden in XHTML documents by putting them into XHTML comments?

10.9 What are the five primitive data types in JavaScript?

10.7. References:

1. Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.
2. M. Deitel, P.J. Deitel, A. B. Goldberg: Internet & World Wide Web How to H program, 3rd Edition, Pearson Education / PHI, 2004.
3. Chris Bates: Web Programming Building Internet Applications, 3rd Edition, Wiley India, 2006.

UNIT -11: INFORMATION CONCEPTS

Structure

- 11.0 Learning Objectives
- 11.1 Functions
- 11.2 Constructor
- 11.3 Pattern matching using regular expressions
- 11.4 Errors in scripts
- 11.5 Summary
- 11.6 Key Words
- 11.7 Exercise
- 11.8 References

11.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Analyze
- Understand the Quality of information
- Identify
- Understand the
- Identify the

11.1 Functions

A function is a piece of code that performs a specific task. The function will be executed by an event or by a call to that function. We can call a function from anywhere within the page (or even from other pages if the function is embedded in an external **.js** file). JavaScript has a lot of builtin functions.

11.1.1 Defining functions

JavaScript function definition consists of the **function** keyword, followed by the name of the function. A list of arguments to the function are enclosed in parentheses and separated by commas. The statements within the function are enclosed in curly braces { }.

Syntax:

```
function functionname(var1,var2,...,varX)
```

```
{
```

```
some code
```

```
}
```

11.1.2 Parameter Passing:

Not every function accepts parameters. When a function receives a value as a parameter, that value is given a name and can be accessed using that name in the function. The names of parameters are taken from the function definition and are applied in the order in which parameters are passed in.

- Primitive data types are passed by value in JavaScript. This means that a copy is made of a variable when it is passed to a function, so any manipulation of a parameter holding primitive data in the body of the function leaves the value of the original variable untouched.
- Unlike primitive data types, composite types such as arrays and objects are passed by reference rather than value.

11.1.3 Examining the function call:

In JavaScript parameters are passed as arrays. Every function has two properties that can be used to find information about the parameters:

functionname.arguments

This is an array of parameters that have been passed

functionname.arguments.length

This is the number of parameters that have been passed into the function

Example:


```
<html>

<body>

<script language="javascript">

function fun(a,b){

var msg = fun.arguments[0]+".." +fun.arguments[1]; //referring a,b values

alert(msg);

}

fun(10,20); //function call

fun("abc","vit"); //function call

</script>

</body>

</html>
```

11.1.4 Returning values

The **return** statement is used to specify the value that is returned from the function. So functions that are going to return a value must use the **return** statement.

Example:

```
function prod(a,b)

{

x=a*b; return x;

}
```

Scoping Rules:

Programming languages usually impose rules, called scoping, which determine how a variable can be accessed. JavaScript is no exception. In JavaScript variables can be either *local* or *global*.

global

Global scoping means that a variable is available to all parts of the program. Such variables are declared outside of any function.

local

Local variables are declared inside a function. They can only be used by that function.

EXCEPTION

HANDLING

>parseInt()	>Converts the string argument to an integer and returns the value. If the string cannot be converted, it returns NaN . Like parseFloat() , this method should handle strings starting with numbers and peel off what it needs, but other mixed strings will not be converted.	<pre>>var x; x = parseInt("-53"); // x is -53 x = parseInt("33.01568"); // x is 33 x = parseInt("47.6k-red-dog"); // x is 47 x = parseInt("a567.34"); // x is NaN x = parseInt("won't work"); // x is NaN</pre>
>unescape()	>Takes a hexadecimal string value containing some characters of the form %xx and returns the ISO-Latin-1 ASCII equivalent of the passed values.	<pre>>Var aString="O%27Neill%20%26%20Sons"; aString = unescape(aString); // aString = "O'Neill & Sons" aString = unescape("%54%56%26%23"); // aString = "dV&#"</pre>

Runtime error handling is vitally important in all programs. Many OOP languages provide a mechanism for handling with general classes of errors. The mechanism is called Exception Handling.

An exception in object-based programming language is an object, created dynamically at run-time, which encapsulates an error and some information about it. In Java Script it returns an error object when an error is generated at browser while the code is executing.

try-catch block

The block of code that might cause the exception is placed inside **try** block. **catch** block contains statements that are to be executed when the exception arises.

```
try

{

statement one

statement two

statement three

}

catch(Error)

{

//Handle errors here

}

finally

{

// execute the code even regardless of above catches are matched

}
```

Example:

```
try{

alert(„This is code inside the try clause“);

ablert („Exception will be thrown by this code“);

}

catch(exception)

{

alert(“Internet Explorer says the error is “ + exception.description);

}
```

11.1.5 Throw statement

The **throw** statement allows us to create an exception. If we use this statement together with the **try...catch** statement, we can control program flow and generate accurate error messages.

Syntax

throw(exception)

The exception can be a string, integer, Boolean or an object

Example:

```
<html>

<body>

<script type="text/javascript">

var x=prompt("Enter a number between 0 and 10:", "");

try
```

```
{  
  
if(x>10)  
  
{  
  
throw "Error! The value is too high";  
  
}  
  
else if(x<0)  
  
{  
  
throw "Error! The value is too low";  
  
}  
  
else if(isNaN(x))  
  
{  
  
throw "Error! The value is not a number";  
  
}  
  
}  
  
catch(er)  
  
{  
  
alert(er);  
  
}  
  
</script>  
  
</body>  
  
</html>
```

11.2 Constructors

JavaScript constructors are special methods that create and initialize the properties of newly created objects. Every new expression must include a call to a constructor whose name is the same as that of the object being created. For example, the constructor for arrays is named `Array`. Constructors are actually called by the `new` operator, which immediately precedes them in the new expression. Obviously, a constructor must be able to reference the object on which it is to operate. JavaScript has a predefined reference variable for this purpose, named `this`. When the constructor is called, `this` is a reference to the newly created object. The `this` variable is used to construct and initialize the properties of the object. For example, the constructor could be used as in the following statement:

```
function car(new_make, new_model, new_year) {  
    this.make = new_make;  
    this.model = new_model;  
    this.year = new_year;  
}
```

```
my_car = new car("Ford", "Contour SVT", "2000");
```

So far, we have considered only data properties. If a method is to be included in the object, it is initialized the same way as if it were a data property. For example, suppose you wanted a method for car objects that listed the property values. A function that could serve as such a method could be written as follows:

```
function display_car() {  
    document.write("Car make: ", this.make, "<br/>");  
    document.write("Car model: ", this.model, "<br/>");  
    document.write("Car year: ", this.year, "<br/>");  
}
```

The following line must then be added to the car constructor:

```
this.display = display_car;
```

Now the call `my_car.display()` will produce the following output:

```
Car make: Ford
Car model: Contour SVT
Car year: 2000
```

The collection of objects created by using the same constructor is related to the concept of class in an object-oriented programming language. All such objects have the same set of properties and methods, at least initially. These objects can diverge from each other through user code changes. Furthermore, there is no convenient way to determine in the script

whether two objects have the same set of properties and methods.

11.3 Pattern matching using regular expressions

A Regular expression is a way of describing a pattern in a piece of text. It's an easy way of matching a string to a pattern. We could write a simple regular expression and use it to check, quickly, whether or not any given string is a properly formatted user input. This saves us from difficulties and allows us to write clean and tight code. For instance, a script might take “*name*” data from a user and have to search through it checking that no digits have been entered. This type of problem can be solved by reading through the string one

character at a time looking for the target pattern. Although it seems like a straightforward approach, it is not (Efficiency & speed matters, so any code that does has to be written carefully). The usual approach in scripting languages is to create a pattern called a *regular expression*, which describes a set of characters that may be present in a string.

11.3.1 Creating Regular Expressions:

A regular expression is a JavaScript object. We can create regular expressions in one of two ways.

- Static regular expressions

```
Ex: var regex = /fish/fowl/ ;
```

- Dynamic regular expressions

```
Ex: var regex = new RegExp(“fish|fowl”);
```

Note: If performance is an issue for our script, then we should try to use static expressions whenever possible. If we don't know what we are going to be searching until runtime (for instance, the pattern may depend on user input) then we create dynamic patterns. A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`.

11.3.2 Using Simple Patterns:

Simple patterns are constructed of characters for which we want to find a direct match.

For example, the pattern `/abc/` matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings

"Hi, do you know your abc's?"

"The latest airplane designs evolved from slabcraft."

In both cases the match is with the substring 'abc'.

There is no match in the string "Grab crab" because it does not contain the substring 'abc'.

11.3.3 Using special characters:

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, the pattern includes special characters.

For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is

followed by zero or more 'b's (* means 0 or more occurrences of the preceding item) and then immediately followed by 'c'. In the string "**cbbabbbbcdebc**," the pattern matches the substring '**abbbbc**'

Either of the following:

- For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character 'b'. By placing a backslash in front of b, that is by using `/\b/`, the character becomes special to mean match a word boundary.

- For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, * is a special character that means 0 or more occurrences of the preceding item should be matched; for example, /a*/ means match 0 or more a's. To match * literally, precede it with a backslash; for example, /a*/ matches 'a*'.

Remembering the Result

Sometimes we may be looking for a pattern that we use elsewhere. This means that, we need to remember the result of our search. The **RegExp** object holds the result of its operations in an array which it returns to the calling script.

Example: Write JavaScript that takes 2 strings as input. In the 1st input string search for the pattern “a followed by zero or more b’s followed by c”. If the pattern appears in 1st string, then the matched string should appear in 2nd string

```
<html>

<body>

<script language="javascript">

var s1 = prompt("enter string1");

var s2 = prompt("enter string2");

var reg = /ab*c/;

var r = s1.match(reg);

if(r){

if(s2.match(r[0])) alert("the matched string is " + r[0]);

else alert("the string is not available in string2");

}
```

else

alert("the pattern is not available in string1");

</script>

</body>

</html>.

Using Parenthesized Substring Matches

- Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered.
- For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'.
- To recall these parenthesized substring matches, use the Array elements `[1]`, ..., `[n]`.
- The number of possible parenthesized substrings is unlimited. The returned array holds all that were found.

Example

```
<script language="javascript">
```

```
re = /(\w+)\s(\w+)/;
```

```
str = "John Smith";
```

```
newstr = str.replace(re, "$2, $1");
```

```
document.write(newstr);
```

```
</script>
```

11.4 Errors in Scripts

The JavaScript interpreter is capable of detecting various errors in scripts. These are primarily syntax errors, although uses of undefined variables are also detected. Debugging a script is a bit

different from debugging a program in a more typical programming language, mostly because errors that are detected by the JavaScript interpreter are found while the browser is attempting to display a document. In some cases, a script error causes the browser not to display the document and does not produce an error message. Without a diagnostic message, you must simply examine the code to find the problem. This is, of course, unacceptable for all but the smallest and simplest scripts. Fortunately, there are ways to get some debugging assistance. Although not a feature of IE7, the default settings for IE8 provide JavaScript syntax error detection and the display of error messages, along with the offending line and character position in the line with the error. These messages are shown in a small window. For example, consider the following sample script:

```
// debugdemo.js
//  An example to illustrate debugging help

var row;
row = 0;
while(row != 4 {
    document.write("row is ", row, "<br />");
    row++;
}
```

Notice the syntax error in the while statement (a missing right parenthesis). Figure 11.1 shows the browser display of what happens when an attempt is made to run

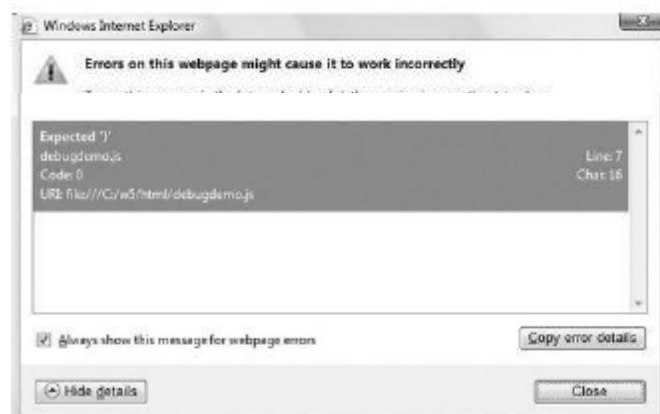


Figure 11.1: Browser display

The FX3 browser has a special console window that displays script errors. Select Tools/Error Console to open the window. When you use this browser to

display documents that include JavaScript, the window should be opened. After an error message has appeared and has been used to fix a script, press the Clear button

on the console. Otherwise, the old error message will remain there and possibly cause confusion about subsequent problems. An example of the FX3 JavaScript

Console window is shown in Figure 11.2.



Figure 11.2: Console window

11.5. SUMMARY

This unit discusses about the use of functions

- JavaScript can execute actions in response to the user's interaction with a GUI component in an XHTML form. This is referred to as GUI event handling
- An XHTML element's onclick attribute indicates the action to take when the user of the XHTML document clicks on the element.
- In event-driven programming, the user interacts with a GUI component, the script is notified of the event and the script processes the event. The user's interaction with the GUI "drives" the program.

The function that is called when an event occurs is known as an event-handling function or event handler.

- The `getElementById` method, given an id as an argument, finds the XHTML element with a matching id attribute and returns a JavaScript object representing the element.
- The `value` property of a JavaScript object representing an XHTML text input element specifies the text to display in the text field.
- Using an XHTML container (e.g. `div`, `span`, `p`) object's `innerHTML` property, we can use a script to set the contents of the element.

11.6. KEY TERMS

argument in a function call, base case, binary format, block, called function, caller, calling function, computer-assisted instruction.

11.7. EXERCISE

4.40 Is it possible to reference global variables in a JavaScript function?

4.41 What is the advantage of using local variables in functions?

4.42 What parameter-passing method does JavaScript use?

4.43 Does JavaScript check the types of actual parameters against the types of their corresponding formal parameters?

4.44 How can a function access actual parameter values for those actual parameters that do not correspond to any formal parameter?

4.45 What is one way in which primitive variables can be passed by reference to a function?

4.46 In JavaScript, what exactly does a constructor do?

4.47 What is a character class in a pattern?

4.48 What are the predefined character classes, and what do they mean?

4.49 What are the symbolic quantifiers, and what do they mean?

4.50 Describe the two end-of-line anchors.

4.51 What does the i pattern modifier do?

4.52 What exactly does the String method replace do?

4.53 What exactly does the String method match do?

11.8. REFERENCES:

1. WamanS.Jawadekar ,”Management Information Systems”,IIIrd Edition,
2. Gordon B.Davis&MargretheH.Olson“ Management Information systems, 2nd edition Tata MC-Graw HILL.

UNIT - 12

Structure

- 12.0 Learning Objectives
- 12.1 Introduction Syntax
- 12.2 Document Structure
- 12.3 Document type delimitation
- 12.4 Namespaces
- 12.5 XML Schemas
- 12.6 Displaying raw XML documents
- 12.7 Displaying xml documents with CSS
- 12.8 Summary
- 12.9 Key Words
- 12.10 Exercise
- 12.11 References

12.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Understand
- Identify the
- Identify the
- Understand the

12.1 Introduction Syntax

The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages. It is classified as an extensible language because it allows its users to define their own elements. Its primary purpose is to facilitate the sharing of structured data across different information systems, particularly via the Internet, and it is used both to encode documents and to serialize data. In the latter context, it is comparable with other text-based serialization languages such as JSON and YAM .

It started as a simplified subset of the Standard Generalized Markup Language (SGML), and is designed to be relatively human-legible. By adding semantic constraints, application languages can be

implemented in XML. These include XHTML, RSS, MathML, GraphML, Scalable Vector Graphics, MusicXML, and thousands of others. Moreover, XML is sometimes used as the specification language for such application languages

Differences between HTML & XML:

HTML	XML
HTML is used to mark up text so it can be displayed to users	XML is used to mark up data so it can be processed by computers
HTML describes both structure (e.g. <p>, <h2>,) and appearance (e.g. , , <i>)	XML describes only content, or “meaning”
HTML uses a fixed, unchangeable set of tags	In XML, you make up your own tags

HTML and XML look similar, because they are both SGML languages (SGML = Standard Generalized Markup Language)

- Both HTML and XML use elements enclosed in tags (e.g. <body>This is an element </body>)
- Both use tag attributes (e.g.,)
- Both use entities (<, >, &, ", ')

More precisely,

- HTML is defined in SGML
- XML is a (very small) subset of SGML

HTML is for humans

- HTML describes web pages
- You don't want to see error messages about the web pages you visit Browsers ignore and/or correct as many HTML errors as they can, so HTML is often sloppy

XML is for computers

- XML describes data
- The rules are strict and errors are not allowed
- In this way, XML is like a programming language
- Current versions of most browsers can display XML
- However, browser support of XML is spotty at best

XML-related technologies:

- DTD (Document Type Definition) and XML Schemas are used to define legal XML tags and their attributes for particular purposes
- CSS (Cascading Style Sheets) describe how to display HTML
- XSLT (eXtensible Stylesheet Language Transformations) and XPath are used to translate from one form of XML to another
- □ □ DOM (Document Object Model), SAX (Simple API for XML), and JAXP (Java API for

XML Processing) are all APIs for XML parsing

XML-based markup languages include

- XHTML
- MathML (for mathematics)
- VoiceXML (for speech)
- SMIL (the Synchronous Multimedia Integration Language—for multimedia presentations)
- CML (Chemical Markup Language—for chemistry)
- XBRL (Extensible Business Reporting Language—for financial data exchange)

Example:

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

```
<weatherReport>
```

```
<date>7/14/97</date>
```

```
<city>North Place</city>, <state>NX</state>
```

```
<country>USA</country>
```

```
High Temp: <high scale="F">103</high>
```

```
Low Temp: <low scale="F">70</low>
```

```
Morning: <morning>Partly cloudy, Hazy</morning>
```

```
Afternoon: <afternoon>Sunny & hot</afternoon>
```

```
Evening: <evening>Clear and Cooler</evening>
```

```
</weatherReport>
```

Another Example:

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

```
<article>
```

```
<title>Simple XML</title>

<date>September 19, 2001</date>

<author>

<firstName>Tem</firstName>

<lastName>Nieto</lastName>

</author>

<summary>XML is pretty easy.</summary>

<content>Once you have mastered XHTML, XML is easily
learned. You must remember that XML is not for
displaying information but for managing information.

</content>

</article>
```

Overall structure:

- An XML document may start with one or more processing instructions (PIs) or directives:

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

```
<?xml-stylesheet type="text/css" href="ss.css"?>
```

- Following the directives, there must be exactly *one* root element containing all the rest of the XML:

```
<weatherReport>

</weatherReport>
```

XML building blocks:

Aside from the directives, an XML document is built from:

- elements: high in `<high scale="F">103</high>`
- tags, in pairs: `<high scale="F">103</high>`
- attributes: `<high scale="F">103</high>`
- entities: `<afternoon>Sunny & hot</afternoon>`
- character data, which may be:
- parsed (processed as XML)--this is the default
- unparsed (all characters stand for themselves)

Elements and attributes:

- Attributes and elements are somewhat interchangeable
- Example using just elements:

```
<name>
```

```
<first>David</first>
```

```
<last>Matuszek</last>
```

```
</name>
```

- Example using attributes:
- `<name first="David" last="Matuszek"></name>`
- You will find that elements are easier to use in your programs--this is a good reason to prefer them
- Attributes often contain metadata, such as unique IDs

- Generally speaking, browsers display only elements (values enclosed by tags), not tags and attributes Well-formed and valid XML documents. There are two levels of correctness of an XML document:

Well-formed XML Document:

- Every element must have *both* a start tag and an end tag, e.g. <name> ... </name>
- But empty elements can be abbreviated: <break />.
- XML tags are case sensitive
- XML tags may not begin with the letters xml, in any combination of cases
- Elements must be properly nested, e.g. *not* <i>bold and italic</i>
- Every XML document must have one and only one root element
- The values of attributes must be enclosed in single or double quotes, e.g. <time unit="days">
- ☐ Character data cannot contain < or &

Valid XML Document:

- You can make up your own XML tags and attributes, *but*
 -any program that *uses* the XML must know what to expect!
- A DTD (Document Type Definition) defines what tags are legal and where they can occur in the XML
- An XML document *does not require* a DTD
- XML is well-structured if it follows the rules and Syntax of XML
- In addition, XML is valid if it declares a DTD and conforms to that DTD
- A DTD can be included in the XML, but is typically a separate document
- Errors in XML documents will *stop* XML programs

Some alternatives to DTDs are XML Schemas and RELAX NG

Entities:

Five special characters must be written as entities:

& for & (almost always necessary)

< for < (almost always necessary)

> for > (not usually necessary)

" for " (necessary inside double quotes)

' for ' (necessary inside single quotes)

- These entities can be used even in places where they are not absolutely required
- These are the *only* predefined entities in XML

XML declaration:

The XML declaration looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

- The XML declaration is not required by browsers, but *is* required by most XML processors (so include it!)
- If present, the XML declaration must be first--*not even whitespace* should precede it
- Note that the brackets are <? and ?>
- version="1.0" is required (this is the *only* version so far)
- encoding can be "UTF-8" (ASCII) or "UTF-16" (Unicode), or something else, or it can be omitted
- standalone tells whether there is a separate DTD

Processing instructions:

- PIs (Processing Instructions) may occur anywhere in the XML document (but usually first)

- A PI is a command to the program processing the XML document to handle it in a certain way
- XML documents are typically processed by more than one program
- Programs that do not recognize a given PI should just ignore it
- General format of a PI: `<?target instructions?>`
- Example: `<?xml-stylesheet type="text/css" href="mySheet.css"?>`

Comments:

- `<!-- This is a comment in both HTML and XML -->`
- Comments can be put anywhere in an XML document
- Comments are useful for:
 - Explaining the structure of an XML document
 - Commenting out parts of the XML during development and testing
- Comments are not elements and do not have an end tag
- The blanks after `<!--` and before `-->` are optional
- The character sequence `--` cannot occur in the comment
- The closing bracket *must* be `-->`
- Comments are not displayed by browsers, but can be seen by anyone who looks at the source code

Names in XML:

- Names (as used for tags and attributes) must begin with a letter or underscore, and can consist of:
 - Letters, both Roman (English) and foreign
 - Digits, both Roman and foreign

- . (dot)
- - (hyphen)
- _ (underscore)
- : (colon) should be used only for namespaces
- Combining characters and extenders (not used in English)

Another well-structured example:

```

<novel>

<foreword>

<paragraph> This is the great American novel.

</paragraph>

</foreword>

<chapter number="1">

<paragraph>It was a dark and stormy night.

</paragraph>

<paragraph>Suddenly, a shot rang out!

</paragraph>

</chapter>

</novel>

```

XML as a tree:

An XML document represents a hierarchy; a hierarchy is a tree

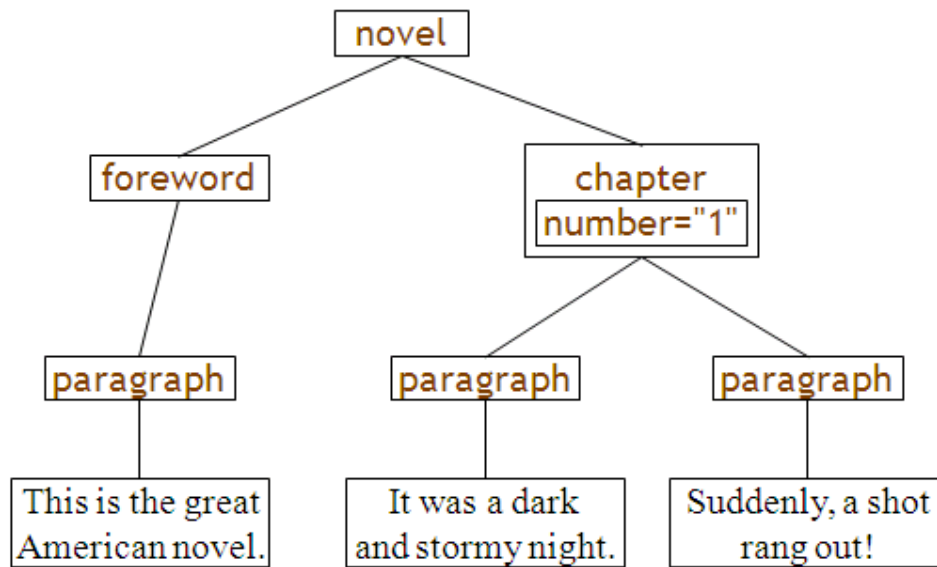


Figure 12.1: Representing hierarchy tree

Document Type Definition (DTD)

- The purpose of a DTD is to define the building blocks of an XML document.
- It defines the Document structure with a list of legal elements.
- A DTD can be declared in two ways
 - Inline DTD Declaration in the XML document itself
 - External DTD reference to XML Document.

Internal DTD Declaration:

- If the DTD is included in the XML source file, DTD definition should be wrapped in a DOCTYPE definition along with XML Definition.
- Syntax

<?xml version="1.0">

<!DOCTYPE root-element [element-declarations]>

XML definition

Simple Example

```
<?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>
```

External DTD Declaration:

- If the DTD is external to the XML source file, the external DTD file should be specified in a DOCTYPE definition without writing DTD definition again.
- Syntax

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

```
<!DOCTYPE root-element SYSTEM "filename">
```

XML definition

Simple Example:

This is a copy of the file “note.dtd” containing the DTD

```
<!ELEMENT note (to ,from ,heading ,body)>
```

```
<!ELEMENT to (#PCDATA)>
```

```
<!ELEMENT from (#PCDATA)>
```

```
<!ELEMENT heading (#PCDATA)>
```

```
<!ELEMENT body (#PCDATA)>
```

This is the XML document 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>
```

Basic Building Blocks of XML Document:

- According to DTD point of view , all XML documents are made up by the building blocks like
- Elements

- Tags
- Attributes
- Entities
- CDATA
- PCDATA

CDATA and PCDATA:

CDATA

- CDATA means character data which is a text that will NOT be parsed by XML parsers.
- Tags inside the text will NOT be treated as markup and entities will not be expanded.

PCDATA

- PCDATA means Parsable character data which is a text that will be parsed by XML Parser
- Tags inside the text will NOT be treated as markup and entities will not be expanded.

Element Declaration in DTD:

- In the DTD, XML elements are declared with element declaration.
- Syntax of an element declaration

<!ELEMENT element-name category>

Or

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

Different types of Element Declarations:

➤ Empty Element Declaration

- Empty elements are declared with the category keyword EMPTY
- Syntax
 - `<!ELEMENT element-name EMPTY>`

➤ DTD Example

- `<!ELEMENT br EMPTY>`

➤ XML Example

- `
`

➤ Elements with only character data

- Elements with only character data are declared with #PCDATA inside parenthesis
- Syntax
 - `<!ELEMENT element-name (#PCDATA)>`

➤ Example

- `<!ELEMENT from (#PCDATA)>`
-

➤ Elements with a sequence of children

- Elements with one or more children are defined with the name of the children elements inside parenthesis

➤ Syntax

- <!ELEMENT element-name (child element1, child element2 child element3,...) >
- Example
 - <!ELEMENT note (to , from ,heading ,body)>
- When children are declared in a sequence separated by comma, the children must be appear in the same sequence in the document.
- Declaring only one occurrence of the element
 - Syntax
 - <!ELEMENT element -name (child-name)>
 - Example
 - <!ELEMENT note(to,from.heading,body)>
- Declaring minimum one occurrence of the same element
 - Syntax
 - <!ELEMENT element-name (child-name+)>
 - Example
 - <!ELEMENT students(students+)>
- Declaring zero or more occurrence of the same content
 - Syntax
 - <!ELEMENT element-name (child-name*)>
 - Example
 - <!ELEMENT students(students*)>
- Declaring zero or one occurrence of the same element

- Syntax
 - `<!ELEMENT element-name (child-name?)>`
- Example
 - `<!ELEMENT note (message?)>`
- Declaring either/or content
 - Syntax
 - `<!ELEMENT element-name(child-name1|child-name2)>`
 - Example
 - `<!ELEMENT note(to,from,header,(message|body))>`
- Declaring Mixed Content
 - Syntax
 - `<!ELEMENT element-name(child-name1 | child-name2,..)>`
 - Example
 - `<!ELEMENT note(#PCDATA | to | from | header | message)>`
- **Declaring Attributes:**
- In the DTD, XML element attributes are declared with an **ATTLIST** declaration. An attribute declaration has the syntax like:
 - `<!ATTLIST element-name attribute-name attribute-type default-value>`
- For the attributes it have the properties like
 - Name of the attribute
 - Type of the attributes

- Default value if any.

Attribute Types:

| 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 |

Attribute Default Value:

- The last entry in the attribute specification determines the attributes default value, if any, and tells whether or not the attribute is required.

| <i>Specification</i> | <i>Specifies...</i> |
|------------------------|--|
| #REQUIRED | The attribute value must be specified in the document. |
| #IMPLIED | The value need not be specified in the document. If it isn't, the application will have a default value it uses. |
| "defaultValue" | The default value to use, if a value is not specified in the document. |
| #FIXED
"fixedValue" | The value to use. If the document specifies any value at all, it must be the same. |

Attribute declaration examples:

- Default attribute value
 - Syntax:
 - `<!ATTLIST element-name attribute-name CDATA "default-value">`
 - DTD example:
 - `<!ATTLIST payment type CDATA "check">`
 - XML example:
 - `<payment type="check">`
- Specifying #IMPLIED
 - Syntax:
 - `<!ATTLIST element-name attribute-name attribute-type IMPLIED>`
 - DTD example: `<!ATTLIST contact fax CDATA #IMPLIED>`
 - XML example: `<contact fax="555-667788">`
- Specifying #REQUIRED
 - Syntax:
 - `<!ATTLIST element-name attribute_name attribute-type REQUIRED>`

- DTD example:
 - `<!ATTLIST person number CDATA #REQUIRED>`
- XML example:
 - `<person number="5677">`
- Specifying #FIXED
 - Syntax:
 - `<!ATTLIST element-name attribute-name attribute-type #FIXED`
`"value">`
 - DTD example:
 - `<!ATTLIST sender company CDATA #FIXED "Microsoft">`
 - XML example:
 - `<sender company="Microsoft">`
- Enumerated attribute values
 - Syntax:
 - `<!ATTLIST element-name attribute-name (val1|val2|..) default-value>`
 - DTD example:
 - `<!ATTLIST payment type (check|cash) "cash">`
 - XML example:
 - `<payment type="check">` or `<payment type="cash">`

Declaring Entities:

- Entities are the variables used to define shortcuts to common text.

- `<!ENTITY>` PI is used to declare user-defined entities.

- Syntax

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

- DTD Example

- `<!ENTITY writer "Donald Duck.">`
- `<!ENTITY copyright "Copyright W3Schools">`

- XML Example

`<author> &writer; ©right; </author>`

Namespaces:

- Recall that DTDs are used to define the tags that can be used in an XML document
- An XML document may reference more than one DTD
- Namespaces are a way to specify which DTD defines a given tag
- XML, like Java, uses qualified names
 - This helps to avoid collisions between names
 - Java: `myObject.myVariable`
 - XML: `myDTD:myTag`
 - Note that XML uses a colon (:) rather than a dot (.)

Namespaces and URIs:

- A namespace is defined as a unique string
 - To guarantee uniqueness, typically a URI (Uniform Resource Indicator) is used, because the author “owns” the domain

- It doesn't have to be a “real” URI; it just has to be a unique string
- Example: `http://www.matuszek.org/ns`
- There are two ways to use namespaces:
 - Declare a default namespace
 - Associate a prefix with a namespace, then use the prefix in the XML to refer to the namespace

Namespace syntax:

- In *any* start tag you can use the reserved attribute name `xmlns`:

```
<book xmlns="http://www.matuszek.org/ns">
```

This namespace will be used as the default for all elements up to the corresponding end tag

You can override it with a specific prefix

- You can use almost this same form to declare a prefix:

```
<book xmlns:dave="http://www.matuszek.org/ns">
```

Use this prefix on *every tag and attribute* you want to use from this namespace, including end tags--it is *not* a default prefix

```
<dave:chapter dave:number="1">To Begin</dave:chapter>
```

- You can use the prefix in the start tag in which it is defined:

```
<dave:book xmlns:dave="http://www.matuszek.org/ns">
```

XML Schema:

- What is XML Schema?
 - XML Schema is vocabulary for expressing constraints for the validity of an XML document.

- A piece of XML is valid if it satisfies the constraints expressed in another XML file, the schema file.
- The idea is to check if the XML file is fit for a certain purpose.
- When we say “XML Schemas,” we usually mean the W3C XML Schema Language
 - This is also known as “XML Schema Definition” language, or xs

Need of XML Schema:

- DTDs provide a very weak specification language
 - We can’t put any restrictions on text content
 - We have very little control over mixed content (text plus elements)
 - We have little control over ordering of elements
- DTDs are written in a strange (non-XML) format
 - We need separate parsers for DTDs and XML
- The XML Schema Definition language solves these problems
 - xs gives us much more control over structure and content
 - xs is written in XML

Disadvantages of XML Schema:

- DTDs have been around longer than xs
 - Therefore they are more widely used
 - Also, more tools support them
- xs is very verbose, even by XML standards

- More advanced XML Schema instructions can be non- intuitive and confusing
- Nevertheless, xs is not likely to go away quickly

Referring to a schema:

- To refer to a DTD in an XML document, the reference goes *before* the root element:

- `<?xml version="1.0"?>`

`<!DOCTYPE rootElement SYSTEM "url">`

`<rootElement> ... </rootElement>`

- To refer to an XML Schema in an XML document, the reference goes *in* the root element:

- `<?xml version="1.0"?>`

- `<rootElement`

- `xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance"`

- (The XML Schema Instance reference is required)

- `xsi:schemaLocation="url xs">`

- (This is where *your* XML Schema definition can be found)

- ...

- `</rootElement>`

The xs Document:

- To prepare the XML document structure for the XML file, first we need to prepare the xs document.
- XML Schema is itself be XML file
- The file extension is .xs

- The root element is <schema>
- The xs starts like this:
 - <?xml version="1.0"?>
 - <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<schema>:

- The <schema> element may have attributes:
 - xmlns:xs="http://www.w3.org/2001/XMLSchema"
 - This is necessary to specify where all our xs tags are defined
 - elementFormDefault="qualified"
 - This means that all XML elements must be qualified with namespace.
 - targetNamespace=http://www.w3schools.com
 - indicates that all elements defined by this schema come from the
http://www.w3schools.com namespace.
- xmlns=http://www.w3schools.com
 - Indicates that the default namespace is “http://www.w3schols.com”.

“Simple” and “complex” elements:

- A “simple” element is one that contains text and nothing else
 - A simple element cannot have attributes
 - A simple element cannot contain other elements
 - A simple element cannot be empty

- However, the text can be of many different types, and may have various restrictions applied to it
- If an element isn't simple, it's "complex"
 - A complex element may have attributes
- A complex element may be empty, or it may contain text, other elements, or both text and other elements

Defining a simple element:

- A simple element is defined as

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

where:

- *name* is the name of the element
- the most common values for *type* are

xs:Boolean	xs:integer
xs:date	xs:string
xs:decimal	xs:time

- Other attributes a simple element may have:
 - `default="default value"` if no other value is specified
 - `fixed="value"` no other value may be specified

➤ Simple Example:

xs Example

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

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


<xs:element name="dateborn" type="xs:date" />

➤ XML Example

<lastname> Vishnu </lastname>

<age>34 </age>

<dateborn> 1968-03-27 </dateborn>

➤ Providing Default value Example

- <xs:element name="color" type="xs:string" default="red" />
- Providing Fixed Value Example
- <xs:element name="color" type="xs:string" fixed="red" />

Defining an attribute:

- Attributes themselves are always declared as simple types
- An attribute is defined as

<xs:attribute name="*name*" type="*type*" />

where:

- *name* and *type* are the same as for xs:element
- Other attributes a simple element may have:
 - default="*default value*" if no other value is specified
 - fixed="*value*" no other value may be specified
 - use="optional" the attribute is not required (default)
 - use="required" the attribute must be present

Simple Example:

➤ xs Example

- `<xs:attribute name="lang" type="xs:string">`

➤ XML Example

- `<lastname lang="EN">Smith</lastname>`

➤ Providing Default Value to the attribute

- `<xs:attribute name="lang" type="xs:string" default="EN" />`

➤ Providing Fixed Value to the attribute

- `<xs:attribute name="lang" type="xs:string" fixed="EN" />`

Restrictions on Content:

- With XML Schema , we can also add our own restrictions tyo our XML elements and attributes. These restrictions are called as “Facets”.

➤ The general form for putting a restriction on an element:

- `<xs:element name="name"> (or xs:attribute)`

`<xs:restriction base="type">`

... the restrictions ...

`</xs:restriction>`

`</xs:element>`

➤ For example:

- `<xs:element name="age">`

`<xs:restriction base="xs:integer">`

```
<xs:minInclusive value="0">  
  
<xs:maxInclusive value="60">  
  
</xs:restriction>  
  
</xs:element>
```

Restrictions on numbers:

- minInclusive -- number must be \geq the given *value*
- minExclusive -- number must be $>$ the given *value*
- maxInclusive -- number must be \leq the given *value*
- maxExclusive -- number must be $<$ the given *value*
- totalDigits -- number must have exactly *value* digits
- fractionDigits -- number must have no more than *value* digits after the decimal point

Restrictions on strings:

- length -- the string must contain exactly *value* characters
- minLength -- the string must contain at least *value* characters
- maxLength -- the string must contain no more than *value* characters
- pattern -- the *value* is a regular expression that the string must match
- whiteSpace -- not really a “restriction”--tells what to do with whitespace
 - value="preserve" Keep all whitespace
 - value="replace" Change all whitespace characters to spaces
 - value="collapse" Remove leading and trailing whitespace, and replace all sequences of whitespace with a single space

Enumeration – Restrictions on set of Values:

- An enumeration restricts the value to be one of a fixed set of values
- Example:

```
<xs:element name="season">
```

```
  <xs:simpleType>
```

```
    <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="Spring"/>
```

```
      <xs:enumeration value="Summer"/>
```

```
      <xs:enumeration value="Autumn"/>
```

```
      <xs:enumeration value="Fall"/>
```

```
      <xs:enumeration value="Winter"/>
```

```
    </xs:restriction>
```

```
  </xs:simpleType>
```

```
</xs:element>
```

Complex elements:

A complex element is defined as

```
<xs:element name="name">
```

```
  <xs:complexType>
```

```
    ... information about the complex type...
```

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

</xs:element>

➤ Example:

<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>

- <xs:sequence> says that elements must occur in this order
- However attributes are always simple types

Global and local definitions:

- Elements declared at the “top level” of a <schema> are available for use throughout the schema
- Elements declared within a xs:complexType are local to that type
- Thus, in

<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>

the elements firstName and lastName are only locally declared

- The order of declarations at the “top level” of a <schema> *do not* specify the order in the XML data document

➤ **xs:all:**

xs:all allows elements to appear in any order

<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>

- Despite the name, the members of an xs:all group can occur once or not at all
- You can use minOccurs="*n*" and maxOccurs="*n*" to specify how many times an element may occur (default value is 1) In this context, *n* may only be 0 or 1

Mixed elements:

- Mixed elements may contain both text and elements

- We add `mixed="true"` to the `xs:complexType` element
- The text itself is not mentioned in the element, and may go anywhere (it is basically ignored)

- `<xs:complexType name="paragraph" mixed="true">`

`<xs:sequence>`

`<xs:element name="someName" type="xs:anyType"/>`

`</xs:sequence>`

`</xs:complexType>`

Predefined string types:

- A simple element is defined as:

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

- Here are a few of the possible string types:

- `xs:string` -- a string
- `xs:normalizedString` -- a string that doesn't contain tabs, newlines, or carriage returns
- `xs:token` -- a string that doesn't contain any whitespace other than single spaces

- Allowable restrictions on strings:

- enumeration, length, maxLength, minLength, pattern, whiteSpace

Predefined date and time types:

- `xs:date` -- A date in the format `YYYY-MM-DD`, for example, 2002-11-05
- `xs:time` -- A date in the format `hh:mm:ss` (hours, minutes, seconds)
- `xs:dateTime` -- Format is `YYYY-MM-DDThh:mm:ss`

- Allowable restrictions on dates and times:
- enumeration, minInclusive, maxExclusive, maxInclusive, maxExclusive, pattern, whiteSpace

Predefined numeric types:

- Here are some of the predefined numeric types:
- `xs:decimal` `xs:positiveInteger`
- `xs:byte` `xs:negativeInteger`
- `xs:short` `xs:nonPositiveInteger`
- `xs:int` `xs:nonNegativeInteger`
- `xs:long`
- Allowable restrictions on numeric types:
 - enumeration, minInclusive, maxExclusive, minExclusive, maxInclusive, fractionDigits, totalDigits, pattern, whiteSpace

<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace=http://www.books.org

xmlns="http://www.books.org"

```
elementFormDefault="qualified">
```

<xs:element name="BookStore">

<xs:complexType>

<xs:sequence>

<xs:element ref="Book" minOccurs="1" maxOccurs="unbounded"/>


```

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="Book">

<xs:complexType>

<xs:sequence>

    <xs:element ref="Title" minOccurs="1" maxOccurs="1"/>

    <xs:element ref="Author" minOccurs="1" maxOccurs="1"/>

    <xs:element ref="Date" minOccurs="1" maxOccurs="1"/>

    <xs:element ref="ISBN" minOccurs="1" maxOccurs="1"/>

    <xs:element ref="Publisher" minOccurs="1" maxOccurs="1"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="Title" type="xs:string"/>

<xs:element name="Author" type="xs:string"/>

<xs:element name="Date" type="xs:string"/>

<xs:element name="ISBN" type="xs:string"/>

<xs:element name="Publisher" type="xs:string"/>

</xs:schema>

```

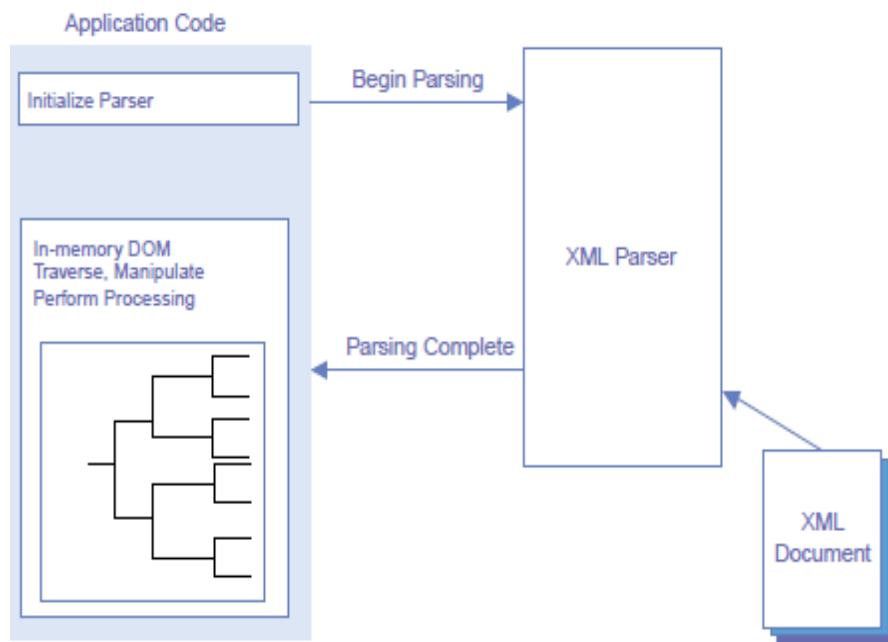
XML Parser:

- An *XML parser* is a software component that can read and validate any XML document.
- A parser makes data contained in an XML data structure available to the application that needs to use it.
- Numerous XML parsers have been developed like
 - DOM
 - SAX
 - Crimson
 - Xerces
 - JAXP

Document Object Model (DOM):

- In the Sun's implementation of DOM model, the parser will read in an entire XML data source and construct a treelike representation of it in memory.
- Under DOM, a pointer to the entire document is returned to the calling application.
- The application can then manipulate the document, rearranging nodes, adding and deleting content as needed by using DOM API.
- While DOM is generally easier to implement, it is far slower and more resource intensive
- DOM can be used effectively with smaller XML data structures in situations when speed is not of paramount importance to the application.

Using the DOM API



DOM API Packages:

The Document Object Model implementation is defined in the following packages:

org.w3c.dom	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.
com.sun.xml.tree	Sun's Java XML implementation of the DOM libraries, including the <u>XmlDocument</u> , <u>XmlDocumentBuilder</u> , and <u>TreeWalker</u> classes.
Javax.xml.parsers	Defines the <u>DocumentBuilderFactory</u> class and <u>DocumentBuilder</u> which returns an object that implements the W3C Document interface. This package also defines the <u>ParserConfigurationException</u> class for reporting errors.

All components in xml document are represented by different kind of nodes

➤ Package org.w3c.dom

- Document
- Element
- Attribute
- Text
- ProcessingInstruction
- Comment
- CDATASection
- PCDATASection
- DocumentType
- Entity
- Namespace
- Children

Some DOM Methods:

➤ Node information

- short getNodeType();
- String getNodeName();
- String getNodeValue();

➤ Namespace information

- String getPrefix();
- String getLocalName();

- String getNamespaceURI();
- Editing node values
 - void setNodeValue(String value);
 - void setPrefix(String prefix);
- Editing tree structure
 - Node appendChild(Node child);
 - Node insertBefore(Node newChild, Node refChild);
 - Node removeChild(Node child);
 - Node replaceChild(Node newChild, Node oldChild);
- Attributes
 - boolean hasAttribute(String name);
 - NamedNodeMap getAttributes();
- Children
 - Node getFirstChild();
 - Node getLastChild();
 - boolean hasChildNodes();
 - NodeList getChildNodes();
- Links to other nodes
 - Document getOwnerDocument();
 - Node getParentNode();
 - Node getPreviousSibling();

- Node getNextSibling();
- Query methods
 - int getLength();
 - Node item(int index);
 - Node getNamedItem(String name);
 - Node getNamedItemNS(String namespaceURI, String localName);
- Editing methods
 - Node setNamedItem(Node node);
 - Node setNamedItemNS(Node node);
 - Node removeNamedItem(String name);
 - Node removeNamedItemNS(String namespaceURI, String localName);

12.6 Displaying Raw XML Documents

An XML-enabled browser— or any other system that can deal with XML documents—cannot know how to format the tags defined in any given document. (After all, someone just made them up.) Therefore, if an XML document is displayed without a style sheet that defines presentation styles for the document's tags, the displayed document will not have formatted content. Contemporary browsers include default style sheets that are used when no style sheet is specified in the XML document. The display of such an XML document is only a somewhat stylized listing of the XML markup. The FX3 browser display of the planes.xml document is shown in Figure 7.2.



Figure 7.2 A display of the XML document `planes.xml` with the FX3 default style sheet

Some of the elements in the display shown in Figure 7.2 are preceded by dashes. These elements can be elided (temporarily suppressed) by placing the mouse cursor over the dash and clicking the left mouse button. For example, if the mouse cursor is placed over the dash to the left of the first `<ad>` tag and the left mouse button is clicked, the result is as shown in Figure 7.3. It is unusual to display a raw XML document. This is usually done only to review and check the structure and content of the document during its development. It is unusual to display a raw XML document. This is usually done only to review and check the structure and content of the document during its development.



Figure 7.3 The document of [Figure 7.2](#) with the first `ad` element elided

12.7 Displaying XML Documents with CSS

Style-sheet information can be provided to the browser for an XML document in two ways. First, a Cascading Style Sheet (CSS) file that has style information for the elements in the XML document can be developed. Second, the XSLT style-sheet technology, which was developed by the W3C, can be used. Although using CSS is effective, XSLT provides far more power over the appearance of the document's display. XSLT is discussed in Section 7.9. The form of a CSS style sheet for an XML document is simple: It is just a list of element names, each followed by a brace-delimited set of the element's CSS attributes. This is the form of the rules in a CSS document style sheet. The following shows a CSS style sheet for the planes XML document:

```

<!-- planes.css - a style sheet for the planes.xml document -->
ad { display: block; margin-top: 15px; color: blue;}
year, make, model { color: red; font-size: 16pt;}
color {display: block; margin-left: 20px; font-size: 12pt;}
description {display: block; margin-left: 20px; font-size: 12pt;}
seller { display: block; margin-left: 15px; font-size: 14pt;}
location {display: block; margin-left: 40px; }
city {font-size: 12pt;}
state {font-size: 12pt;}

```

The only style property in this style sheet that has not been discussed earlier is `display`, which is used to specify whether an element is to be displayed inline or in a separate block. These two options are

specified with the values inline and block. The inline value is the default. When display is set to block, the content of the element is usually separated from its sibling elements by line breaks.

The connection of an XML document to a CSS style sheet is established with the processing instruction `xml-stylesheet`, which specifies the particular type of the style sheet via its `type` attribute and the name of the file that stores the style sheet via its `href` attribute. For the planes example, this processing instruction is as follows:

```
<?xml-stylesheet type = "text/css" href = "planes.css" ?>
```

Figure 7.4 shows the display of `planes.xml`, in which the `planes.css` style sheet is used to format the document.



Figure 7.4

12.8. SUMMARY

XML documents are readable by both humans and machines.

- XML permits document authors to create custom markup for any type of information. This enables document authors to create entirely new markup languages that describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes.

- An XML parser is responsible for identifying components of XML documents (typically files with the .xml extension) and then storing those components in a data structure for manipulation.
- An XML document can optionally reference a Document Type Definition (DTD) or schema that defines the XML document's structure.
- An XML document that conforms to a DTD/schema (i.e., has the appropriate structure) is valid.
- If an XML parser (validating or nonvalidating) can process an XML document successfully, that XML document is well-formed.

DTDs and schemas specify documents' element types and attributes, and their relationships to one another.

- DTDs and schemas enable an XML parser to verify whether an XML document is valid (i.e., its elements contain the proper attributes and appear in the proper sequence).
- A DTD expresses the set of rules for document structure using an EBNF (Extended Backus-Naur Form) grammar.
- In a DTD, an ELEMENT element type declaration defines the rules for an element. An ATTLIST attribute-list declaration defines attributes for a particular element.

12.9. KEY TERMS

δ entity reference (MathML), ⁢ entity reference (MathML), ∫ entity reference (MathML), .mml filename extension for MathML documents, /, forward slash in end tags, /, XPath root selector <!--...-->, XML comment tags, <? and ?> XML processing instruction delimiters, @, XPath attribute symbol, absolute addressing

12.10. EXERCISE

-
- Write a processing instruction that includes style sheet wap.xsl.
 - Write an XPath expression that locates contact nodes in letter.xml

12.11. REFERENCES:

1. WamanS.Jawadekar ,”Management Information Systems”,IIIrd Edition,
2. Gordon B.Davis&MargretheH.Olson“ Management Information systems, 2nd edition Tata MC-Graw HILL.

Module: 4

UNIT 13-: The Basics of Perl and CGI

Structure

- 13.0 Learning Objectives
- 13.1 Introduction
- 13.2 Perl
- 13.3 CGI Programming Origins and uses of Perl
- 13.4 Scalars and their operations
- 13.5 Assignment statements and simple input and output
- 13.6 Summary
- 13.7 Keywords
- 13.8 Exercises
- 13.9 References

13.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Analyze basic concepts of PERL
- Advantages of CGI programming
- The concept of CGI

13.1. Introduction

Perl is a popular language to script CGI programs. Important Perl features are introduced to help you write better CGI programs. Perl variables, conditionals, I/O, loops, function definition, patterns, the taint mode, and more will be described. Enough information is included for you to use Perl comfortably for CGI programming.

The CGI.pm module and how it helps CGI programming is emphasized. Cookies and their application in multi-step user transactions are explained and demonstrated with examples. Materials presented here give a concise and practical introduction to Perl CGI programming. With this background the reader can learn more about Perl and CGI with ease.

What is Perl?

Depending on whom you ask, Perl stands for “Practical Extraction and report Language” or “Pathologically Eclectic Rubbish Lister.” It is a powerful glue language useful for tying together the loose ends of computing life.

History

Perl is the natural outgrowth of a project started by Larry Wall in 1986. Originally intended as a configuration and control system for six VAXes and six SUNs located on opposite ends of the country, it grew into a more general tool for system administration on many platforms. Since its unveiling to programmers at large, it has become the work of a large body of developers. Larry Wall, however, remains its principle architect.

Although the first platform Perl inhabited was UNIX, it has since been ported to over 70

different operating systems including, but not limited to, Windows 9x/NT/2000, MacOS,

VMS, Linux, UNIX (many variants), BeOS, LynxOS, and QNX.

Uses of Perl

1. Tool for general system administration
2. Processing textual or numerical data
3. Database interconnectivity
4. Common Gateway Interface (CGI/Web) programming
5. Driving other programs! (FTP, Mail, WWW, OLE)

Philosophy & Idioms

The Virtues of a Programmer

Perl is a language designed to cater to the three chief virtues of a programmer.

- Laziness - develop reusable and general solutions to problems
- Impatience - develop programs that anticipate your needs and solve problems

for you.

· Hubris - write programs that you want other people to see (and be able to maintain)

There are many means to the same end

Perl provides you with more than enough rope to hang yourself. Depending on the problem, there may be several “official” solutions. Generally those that are approached using “Perl idioms” will be more efficient.

13.2 Perl

Perl is the Practical Extraction and Report Language and is freely available for downloading from the Comprehensive Perl Archive Network (www.perl.com/CPAN/). Perl is a portable, command line driven, interpreted programming/scripting language. Written properly, the same Perl code will run identically on UNIX, Windows, and Mac OS systems.

A Perl program (or script) consists of a sequence of commands and the source code file can be named arbitrarily but usually uses the .pl suffix. A Perl interpreter reads the source file and executes the commands in the order given. You may use any text editor to create Perl scripts. These scripts will work on any platform where the Perl interpreter has been installed.

The Perl scripting language is usually used in the following applications areas:

- Web CGI programming
- DOS and UNIX shell command scripts
- Text input parsing
- Report generation
- Text file transformations, conversions

Although Perl is not Web specific, our coverage of Perl is focused on the application of Perl in Web CGI programming.

Perl was created in the UNIX tradition of open source software. Perl 1.0 was released 18

December 1987 (the Perl birthday) by Larry Hall with the following description: Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many

system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of C, sed, awk, and sh, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of csh, Pascal, and even BASIC|PLUS.) Expression syntax corresponds quite closely to C expression syntax. If you have a problem that would ordinarily use sed or awk or sh, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then perl may be for you.

In 1989 Perl 3.0 was released and distributed for the first time under the GNU Public License with its now well-known copy left philosophy. Released in 1994 was Perl 5.0, a complete rewrite of Perl adding objects and a modular organization. The modular structure makes it easy for everyone to develop Perl modules to extend the functionalities of Perl. CGI.pm (CGI Perl Module) is just such a library (1995 by Lincoln Stein). This module makes CGI programming in Perl much easier and more powerful.

Many other Perl modules have been created. The CPAN (Comprehensive Perl Archive Network, www.cpan.org) was established to store and distribute Perl and Perl related software.

Because of its text processing ease, wide availability (it runs on all major platforms), and CGI programming abilities, Perl has become one of the most popular languages for CGI programming.

13.3 CGI Programming Origins and uses of Perl

A CGI is simply a program that is called by the webserver, in response to some action by a web visitor. This might be something simple like a page counter, or a complex formhandler. Shopping carts and e-commerce sites are driven by CGI programs. So are ad banners; they keep track of who has seen and clicked on an ad. CGI programs may be written in any programming language; we're just using Perl because it's fairly easy to learn. If you're already an expert in some other language and are just reading to get the basics, here it is: if

you're writing a CGI that's going to generate an HTML page, you must include this statement somewhere in the program before you print out anything else:

```
print "Content-type: text/html\n\n";
```

This is a content-type header that tells the receiving web browser what sort of data it is about to receive – in this case, an HTML document. If you forget to include it, or if you print something else before printing this header, you'll get an "Internal Server Error" when you try to access the CGI program.

Now let's try writing a simple CGI program. Enter the following lines into a new file, and name it "first.cgi". Note that even though the lines appear indented on this page, you do not have to indent them in your file. The first line (`#!/usr/bin/perl`) should start in column 1. The subsequent lines can start in any column.

Program : first.cgi Hello World Program

```
#!/usr/bin/perl -wT
```

```
print "Content-type: text/html\n\n";
```

```
print "Hello, world!\n";
```

the file permissions (or use your FTP program to change them). You will have to do this every time you create a new program; however, if you're editing an existing program, the permissions will remain the same and shouldn't need to be changed again.

Now go to your web browser and type the direct URL for your new CGI. For example:

```
http://www.cgi101.com/book/ch1/first.cgi
```

Your actual URL will depend on your ISP. If you have an account on cgi101, you're URL is:

```
http://www.cgi101.com/~youruserid/first.cgi
```


You should see a web page with “Hello, world!” on it.

Let’s try another example. Start a new file (or if you prefer, edit your existing first.cgi) and add some additional print statements. It’s up to your program to print out all of the HTML you want to display in the visitor’s browser, so you’ll have to include print statements for every HTML tag:

Program 1-2: second.cgi Hello World Program 2

```
#!/usr/bin/perl -wT

print "Content-type: text/html\n\n";

print "<html><head><title>Hello World</title></head>\n";

print "<body>\n";

print "<h2>Hello, world!</h2>\n";

print "</body></html>\n";
```

Source code: <http://www.cgi101.com/book/ch1/second-cgi.html>

➦ Working example: <http://www.cgi101.com/book/ch1/second.cgi>

Save this file, adjust the file permissions if necessary, and view it in your web browser. This time you should see “Hello, world!” displayed in a H2-size HTML header. Now not only have you learned to write your first CGI program, you’ve also learned your first Perl statement, the print function:

```
print "somestring";
```

This function will write out any string, variable, or combinations thereof to the current output channel. In the case of your CGI program, the current output is being printed to the visitor’s browser.

The `\n` you printed at the end of each string is the newline character. Newlines are not required, but they will make your program's output easier to read. You can write multiple lines of text without using multiple print statements by using the here-document syntax:

```
print <<EndMarker;
```

```
line1
```

```
line2
```

```
line3
```

```
etc.
```

```
EndMarker
```

You can use any word or phrase for the end marker (you'll see an example next where we use "EndOfHTML" as the marker); just be sure that the closing marker matches the opening marker exactly (it is case-sensitive), and also that the closing marker is on a line by itself, with no spaces before or after the marker.

Uses of Perl

1. Tool for general system administration
2. Processing textual or numerical data
3. Database interconnectivity
4. Common Gateway Interface (CGI/Web) programming
5. Driving other programs! (FTP, Mail, WWW, OLE)

Perl is the de facto standard for CGI programming for a number of reasons, but perhaps the most important are:

- **Socket Support**-create programs that interface seamlessly with Internet protocols. Your CGI program can send a Web page in response to a transaction and send a series of e-mail messages to inform interested people that the transaction happened.
- **Pattern Matching**-ideal for handling form data and searching text.
- **Flexible Text Handling**-no details to worry. The way that Perl handles strings, in terms of memory allocation and deallocation, fades into the background as you program. You simply can ignore the details of concatenating, copying, and creating new strings.

The advantage of an interpreted language in CGI applications is its simplicity in development, debugging, and revision. By removing the compilation step, you and I can move more quickly from task to task, without the frustration that can sometimes arise from debugging compiled programs. Of course, not any interpreted language will do. Perl has the distinct advantage of having an extremely rich and capable functionality.

There are some times when a mature CGI application should be ported to C or another compiled language. These are the Web applications where speed is important. If you expect to have a very active site, you probably want to move to a compiled language because they run faster.

13.4 Scalars and their operations

A scalar variable stores a single (scalar) value. Perl scalar names are prefixed with a dollar sign (\$), so for example, \$x, \$y, \$z, \$username, and \$url are all examples of scalar variable names. Here's how variables are set:

```
$foo = 1;
```

```
$name = "Fred";
```

```
$pi = 3.141592;
```

In this example \$foo, \$name, and \$pi are scalars. You do not have to declare a variable before using it, but its considered good programming style to do so. There are several different ways to declare variables, but the most common way is with the my function:

```
my $foo = 1;
```

```
my ($name) = "Fred";
```

```
my ($pi) = 3.141592;
```

my simultaneously declares the variables and limits their scope (the area of code that can see these variables) to the enclosing code block. You can declare a variable without giving it a value:

```
my $foo;
```

You can also declare several variables with the same my statement:

```
my ($foo, $bar, $blee);
```

You can omit the parentheses if you are declaring a single variable, however a list of variables must be enclosed in parentheses.

A scalar can hold data of any type, be it a string, a number, or whatnot. You can also use scalars in double-quoted strings:

```
my $fnord = 23;
```

```
my $blee = "The magic number is $fnord.";
```

Now if you print \$blee, you will get “The magic number is 23.” Perl interpolates the variables in the string, replacing the variable name with the value of that variable.

Let’s try it out in a CGI program. Start a new program called scalar.cgi:

Program 2-1: scalar.cgi Print Scalar Variables Program

```
#!/usr/bin/perl -wT
```

```
use CGI qw(:standard);
```

```
use CGI::Carp qw(warningsToBrowser fatalsToBrowser);
```

```
use strict;
```

```

my $email = "fnord\@cgi101.com";

my $url = "http://www.cgi101.com";

print header;

print start_html("Scalars");

print <<EndHTML;

<h2>Hello</h2>

<p>

My e-mail address is $email, and my web url is

<a href="$url">$url</a>.

</p>

EndHTML

print end_html;

```

2 Source code: <http://www.cgi101.com/book/ch2/scalar-cgi.html>

↪ Working example: <http://www.cgi101.com/book/ch2/scalar.cgi>

You may change the \$email and \$url variables to show your own e-mail address* and website URL. Save the program, `chmod 755 scalar.cgi`, and test it in your browser. You'll notice a few new things in this program. First, there's `use strict`. This is a standard Perl module that requires you to declare all variables. You don't have to use the `strict` module, but it's considered good programming style, so it's good to get in the habit of using it.

You'll also notice the variable declarations:

```

my $email = "fnord\@cgi101.com";

my $url = "http://www.cgi101.com";

```

Notice that the @-sign in the e-mail address is escaped with (preceded by) a backslash. This is because the @-sign means something special to Perl – just as the dollar sign indicates a scalar variable, the @-sign indicates an array, so if you want to actually use special characters like @, \$, and % inside a double-quoted string, you have to precede them with a backslash (\).

A better way to do this would be to use a single-quoted string for the e-mail address:

```
my $email = 'fnord@cgi101.com';
```

Single-quoted strings are not interpolated the way double-quoted strings are, so you can freely use the special characters \$, @ and % in them. However this also means you can't

use a single-quoted string to print out a variable, because

```
print '$fnord';
```

will print the actual string “\$fnord” . . . not the value stored in the variable named

\$fnord

13.5 Assignment statements and simple input and output

Assignment statements

- Perl's assignment statements are the same as in Java

```
$salary = 40000;
```

- Similar to Java, assignment statements return the assigned value as their value
 - We can use this in conditional statements (eg, inside a *while*)
- There are also the usual shorthand operators, eg

```
$salary += $raise;
```

- Note that all Perl statements (except those at the end of a block -- see control structures) are terminated by a semicolon

A CGI script will typically require some form of input in order to operate.

- In fact, only very trivial CGI scripts can be created without input. We have introduced HTML Forms as a prime means of CGI input.
- However, there are several other forms of input to a CGI script.

In this section we will study:

- What form of input a CGI can receive
- How a CGI receives input.
- How to process the input in a CGI Perl script
- How a useful Perl library makes this (and other) tasks easy.

Accepting Input from the Browser

A CGI script can receive data in one of four ways:

Environment Variables — It gets various information about the

browser, the server and the CGI script itself through specially named variables automatically created and setup by the server. More on these later. Standard Input — Data can be passed as standard input to CGI script.

Usually this is through the POST method of an HTML Form. (Standalone Perl scripts get standard input from the keyboard or a file.)

Arguments of the CGI Script — If you call a CGI script directly or use the GET method of HTML Form posting information is passed as arguments of the CGI script URL.

- Arguments are followed a ? after the CGI script URL and multiple arguments are separated by &.
- For example:

`http://host/cgi-bin?arg1&arg2`

The arguments are usually in the form of name/value pairs. Path Information — Files which may be read by a CGI script can be passed to a CGI script by appending the file path name to the end of the URL but before the ? and any arguments.

For example:

`http://host/cgi-bin/script/mypath/cgiinput?arg1&arg2`

Path information is useful if a CGI script requires data that

- Does not frequently change,
- Requires a lot of arguments and/or
- Does not rely on user input values.
- Path Information often refers to files on the Web server such as configuration files, temporary files or data files.

CGI Input via a URL

If you enter some data in the Text field and click on submit then the call to the CGI script looks something like

`http://myhost/minimal.pl?myfield=dddd`

where `minimal.pl` is the CGI script actioned by the form, If you want to call the CGI script yourself (bypassing the Form) you simply mimic the input above.

`Dave.Marshall/cgi-bin/minimal.pl?myfield=mydata` in the browser location bar.

Exercise: Change to Form method attribute to POST and observe the difference in the CGI call.

Input from the Console

- Although we will be talking mostly about using Perl in CGI scripts, it is useful to know how you can read data from the console
- The following code reads a single line from the standard input

The angles (`<` and `>`) indicate the **line input** operator

```
$data = <STDIN>;
```

- Generally, you don't want the newline character at the end:


```
$data = <STDIN>;
```

```
chomp $data;
```

- You can even shorten this to (although this might be less clear):

```
chomp($data = <STDIN>);
```

CGI Script Output

We have already mentioned that CGI scripts must adhere to standard input and output mechanism for the moment we will not worry about input to a CGI script.

- The Interface between browser and server
- Part of HTTP Protocol

CGI Script Output Format

In whatever language a CGI script is programmed it **MUST** send information back in the following format:

- The Output Header
- A Blank Line
- The Output Data

NOTE: Between the Header and Data there **MUST** be a blank line.

CGI Output Header

- A browser can accept input in a variety of forms.
- Depending on the specified form it will call different mechanisms to display the data.
- The output header of a CGI script must specify an output type to tell the server and eventually browser how to proceed with the rest of the CGI output. Three forms of Header Type. There are 3 forms of Header Type:

- Content-Type

- Location

- Status

Content-Type is the most popular type.

- We now consider this further.

- We will meet the other types later.

CGI Output Data

Depending on the Content-Type defined the data that follows the header declaration will vary:

- If it is HTML that follows then the CGI script must output

standard HTML syntax.

Example: To produce a Web page that the server sends to a browser with a simple line of text "Hello World!". A CGI script must output:

Content-Type: text/html

```
<html>
```

```
<head>
```

```
<title>Hello, world!</title>
```

```
</head>
```

```
<body>
```

```
<h1>Hello, world!</h1>
```

```
</body>
```

```
</html>
```

Output to the Console

- The *print* function directs output to the console
 - The arguments to *print* are one or more strings, separated by commas
 - No implicit newline character is added to the last string

```
print "One, ", $two, "buckle my shoe\n";
```

- The following program illustrates some of what we discussed up to now:

```
#!/usr/bin/perl -w
```

```
print "Name: ";
```

```
$name = <STDIN>;
```

```
chomp $name;
```

```
print "Surname: ";
```

```
$surname = <STDIN>;
```

```
chomp $surname;
```

```
print "Welcome, $name $surname!\n";
```

13.6 Summary

“CGI” stands for “Common Gateway Interface.” CGI is one method by which a web server can obtain data from (or send data to) databases, documents, and other programs, and presents that data to viewers via the web. More simply, a CGI is a program intended to be run on the web. A CGI program can be written in any programming language, but Perl is one of the most popular, Perl is the language we’ll be using.

Also need Perl and a web server (such as Apache) that is configured to allow you to run your own CGI programs. The book is written towards CGI programming on Unix, but you can also set up Apache and Perl on Mac OS X and Windows. I’ve written several online tutorials that will show you how to get started:

Windows: <http://www.cgi101.com/book/connect/windows.html>

How to set up Apache and Perl; how to configure Apache; where to write your programs; differences between CGI programs on Windows and Unix Mac OS X:
<http://www.cgi101.com/book/connect/mac.html>

How to configure Apache (which you already have installed); where to write your programs Unix:
<http://www.cgi101.com/book/connect/unix.html> How to upload programs to your Unix-based server;
Unix tutorial;

13.7 Keywords

Perl, CGI, CGI output Header, Scalars etc...

13.8 Exercises

13.0 Name one principal site on the World Wide Web to get more information about Perl.

13.1 Write a Perl program which asks the user for their name and address using three separate prompts. Test each line for null values. Print out the lines to standard output in a mailing label format.

13.2 Write a Perl program which reads a file from standard input and print it to standard output. Combine the program execution with redirection to copy a file.

13.3 What is CGI ?

13.4 List two common mistakes made when using CGI scripts and their systems.

13.5 What is server's role in running a CGI script?

13.9 References

- M. Deitel, P.J. Deitel, A.B. Goldberg: Internet & World Wide Web How to program, 3rd Edition, Pearson Education/ PHI, 2004.
- Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.

UNIT 14:- The Basics of Perl and CGI Structure

Structure

- 14.0 Learning Objectives
- 14.1 Control statements
- 14.2 Fundamentals of arrays
- 14.3 Hashes
- 14.4 References
- 14.5 Functions
- 14.6 Summary
- 14.7 Keywords
- 14.8 Exercises
- 14.9 References

14.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Understand about Control statements
- Understand about Fundamentals of arrays
- Understand about Hashes and Functions

14.1 Control Statements

Control structures include conditional statements, such as if/elsif/else blocks, as well as loops like foreach, for and while.

If Conditions

You've already seen if/elsif in action. The structure is always started by the word if, followed by a condition to be evaluated, then a pair of braces indicating the beginning and end of the code to be executed if the condition is true. The condition is enclosed in parentheses:

```
if (condition) {  
  
code to be executed  
  
}
```

The condition statement can be anything that evaluates to true or false. In Perl, any string is true except the empty string and 0. Any number is true except 0. An undefined value (or undef) is false. You can also test whether a certain value equals something, or doesn't equal something, or is greater than or less than something. There are different conditional test operators, depending on whether the variable you want to test is a string or a number:

Relational and Equality Operators

Test	Numbers	Strings
\$x is equal to \$y	<code>\$x == \$y</code>	<code>\$x eq \$y</code>
\$x is not equal to \$y	<code>\$x != \$y</code>	<code>\$x ne \$y</code>
\$x is greater than \$y	<code>\$x > \$y</code>	<code>\$x gt \$y</code>
\$x is greater than or equal to \$y	<code>\$x >= \$y</code>	<code>\$x ge \$y</code>
\$x is less than \$y	<code>\$x < \$y</code>	<code>\$x lt \$y</code>
\$x is less than or equal to \$y	<code>\$x <= \$y</code>	<code>\$x le \$y</code>

If it's a string test, you use the letter operators (eq, ne, lt, etc.), and if it's a numeric test, you use the symbols (==, !=, etc.). Also, if you are doing numeric tests, keep in mind that

`$x >= $y` is not the same as `$x => $y`. Be sure to use the correct operator!

Here is an example of a numeric test. If \$varname is greater than 23, the code inside the curly braces is executed:

```
if ($varname > 23) {  
  
# do stuff here if the condition is true
```

```
}
```

If you need to have more than one condition, you can add `elsif` and `else` blocks:

```
if ($varname eq "somestring") {  
    # do stuff here if the condition is true  
}  
  
elsif ($varname eq "someotherstring") {  
    # do other stuff  
}  
  
else {  
    # do this if none of the other conditions are met  
}
```

The line breaks are not required; this example is just as valid:

```
if ($varname > 23) {  
    print "$varname is greater than 23";  
}  
elsif ($varname == 23) {  
    print "$varname is 23";  
}  
else { print "$varname is less than 23"; }
```

You can join conditions together by using logical operators:

Logical Operators

Operator	Example	Explanation
<code>&&</code>	<code>condition1 && condition2</code>	True if condition1 and condition2 are both true

`|| condition1 || condition2` True if either condition1 or condition2 is true and condition1 and condition2 Same as `&&` but lower precedence or condition1 or condition2 Same as `||` but lower precedence. Logical operators are evaluated from left to right. Precedence indicates which operator is evaluated first, in the event that more than one operator appears on one line. In a case like this:

```
condition1 || condition2 && condition3
```

`condition2 && condition3` is evaluated first, then the result of that evaluation is

used in the `||` evaluation.

and and or work the same way as `&&` and `||`, although they have lower precedence than their symbolic counterparts.

Unless

unless is similar to if. Let's say you wanted to execute code only if a certain condition

were false. You could do something like this:

```
if ($varname != 23) {  
  
# code to execute if $varname is not 23  
  
}
```

The same test can be done using unless:

```
unless ($varname == 23) {  
  
# code to execute if $varname is not 23  
  
}
```

There is no "elseunless", but you can use an else clause:

```
unless ($varname == 23) {  
  
# code to execute if $varname is not 23  
  
} else {  
  
# code to execute if $varname IS 23
```



```
}
```

Validating Form Data

You should always validate data submitted on a form; that is, check to see that the form fields aren't blank, and that the data submitted is in the format you expected. This is typically done with if/elsif blocks.

Here are some examples. This condition checks to see if the "name" field isn't blank:

```
if (param('name') eq "") {  
  
  &dienice("Please fill out the field for your name.");  
  
}
```

You can also test multiple fields at the same time:

```
if (param('name') eq "" or param('email') eq "") {  
  
  &dienice("Please fill out the fields for your name  
and email address.");  
  
}
```

The above code will return an error if either the name or email fields are left blank.

param('fieldname') always returns one of the following:

undef or undefined fieldname is not defined in the form itself, or it's a checkbox/radio button field that wasn't checked. the empty string fieldname exists in the form but the user didn't type anything into that field (for text fields) one or more values whatever the user typed into the field(s) If your form has more than one field containing the same fieldname, then the values are stored sequentially in an array, accessed by param('fieldname').

You should always validate all form data – even fields that are submitted as hidden fields

in your form. Don't assume that your form is always the one calling your program. Any

external site can send data to your CGI. Never trust form input data.

Looping

Loops allow you to repeat code for as long as a condition is met. Perl has several loop control structures: `foreach`, `for`, `while` and `until`.

Foreach Loops

`foreach` iterates through a list of values:

```
foreach my $i (@arrayname) {  
  
    # code here  
  
}
```

This loops through each element of `@arrayname`, setting `$i` to the current array element

for each pass through the loop. You may omit the loop variable `$i`:

```
foreach (@arrayname) {  
  
    # $_ is the current array element  
  
}
```

This sets the special Perl variable `$_` to each array element. `$_` does not need to be declared (it's part of the Perl language) and its scope localized to the loop itself.

For Loops

Perl also supports C-style for loops:

```
for ($i = 1; $i < 23; $i++) {  
  
    # code here  
  
}
```

The `for` statement uses a 3-part conditional: the loop initializer; the loop condition (how

long to run the loop); and the loop re-initializer (what to do at the end of each iteration of the loop). In the above example, the loop initializes with `$i` being set to 1. The loop will

run for as long as \$i is less than 23, and at the end of each iteration \$i is incremented by 1 using the auto-increment operator (++).

The conditional expressions are optional. You can do infinite loops by omitting all three conditions:

```
for (;;) {  
  
# code here  
  
}
```

You can also write infinite loops with while.

While Loops

A while loop executes as long as particular condition is true:

```
while (condition) {  
  
# code to run as long as condition is true  
  
}
```

Until Loops

until is the reverse of while. It executes as long as a particular condition is NOT true:

```
until (condition) {  
  
# code to run as long as condition is not true  
  
}
```

Infinite Loops

An infinite loop is usually written like so:

```
while (1) {  
  
# code here
```

```
}
```

Obviously unless you want your program to run forever, you'll need some way to break out of these infinite loops. We'll look at breaking next.

Breaking from Loops

There are several ways to break from a loop. To stop the current loop iteration (and move on to the next one), use the next command:

```
foreach my $i (1..20) {  
  
    if ($i == 13) {  
  
        next;  
  
    }  
  
    print "$i\n";  
  
}
```

This example prints the numbers from 1 to 20, except for the number 13. When it reaches 13, it skips to the next iteration of the loop.

To break out of a loop entirely, use the last command:

```
foreach my $i (1..20) {  
  
    if ($i == 13) {  
  
        last;  
  
    }  
  
    print "$i\n";  
  
}
```

This example prints the numbers from 1 to 12, then terminates the loop when it reaches 13.

next and last only effect the innermost loop structure, so if you have something like this:

```
foreach my $i (@list1) {  
  
  foreach my $j (@list2) {  
  
    if ($i == 5 && $j == 23) {  
  
      last;  
  
    }  
  
  }  
  
  # this is where that last sends you  
  
}
```

The last command only terminates the innermost loop. If you want to break out of the outer loop, you need to use loop labels:

```
OUTER: foreach my $i (@list1) {  
  
  INNER: foreach my $j (@list2) {  
  
    if ($i == 5 && $j == 23) {  
  
      last OUTER;  
  
    }  
  
  }  
  
}  
  
# this is where that last sends you
```

The loop label is a string that appears before the loop command (foreach, for, or while). In this example we used OUTER as the label for the outer foreach loop and INNER for the inner loop label.

14.2 Fundamentals of Arrays

The Perl array variable uses the `@` prefix. For example (Ex: PerlArray),

```
#!/usr/bin/perl

@arr = ("aa", "bb", "cc", "dd"); ## creating an array

print "$arr[0]\n"; ## first array element is aa (B)

$arr[2]=7; ## third element set to 7 (C)

$m = $#arr; ## 3, last index of @arr (D)

$n = @arr; ## n is 4 length of @arr (E)

print "@arr\n"; ## aa bb 7 dd (F)

push(@arr, "xyz"); ## put on end of array (G)

print "@arr\n"; ## aa bb 7 dd xyz

$last = pop(@arr); ## pop off end of array (H)

print "@arr\n"; ## aa bb 7 dd
```

Note we use the scalar notation to retrieve or set values on an array using indexing (lines B-C). The special prefix `$#` returns the index of the last array element (line D) and -1 if the array has no elements. Hence, `$#arr+1` is the array length. Assigning an array to a scalar produces its length (line E). Displaying the entire array is as easy as printing it (line F). Use the Perl built-in function `push` to append a new element to the end of the array (line G) and the `pop` function to remove the last element from the array (line H). The function `shift` (unshift) deletes (adds) an element at the beginning of an array.

Executing this program produces the following output

```
aa

aa bb 7 dd

aa bb 7 dd xyz

aa bb 7 dd
```

Association Arrays

An association array, also known as a hash array, is an array with even number of elements.

Elements come in pairs, a key and a value. You can create an hash arrays with the notation:

```
( key1 => value1, key2 => value2, ... )
```

The keys serve as symbolic indices for the corresponding values on the association array.

Perl association array variables use the % pre-fix. For example (Ex: PerlAsso),

```
%asso = ( "a" => 7, "b" => 11 ); ## (1)
print "$asso{'a'}\n"; ## displays 7 (2)
print "$asso{'b'}\n"; ## displays 11 (3)
print "@asso{'a', 'b'}\n"; ## displays 7, 11 (4)
```

The symbol => (line 1) makes the association perfectly clear. But a comma works just as well.

```
%asso = ( "a", 7, "b", 11 );
```

To retrieve a value from an association array, use its key (lines 2-3). Note the \$ prefix is used with the key enclosed in curly braces ({}). To obtain a list of values from an association list, the @ prefix can be used (line 4). Use a non-existent key or a value as a key (\$asso{'z'}, \$asso{7} for example) and you'll get an undefined value (undef).

Assign a new value with a similar assignment where the key may or may not already be on the association array:

```
$asso{'c'} = 13;
```

To remove key-value pairs from a hash, use calls like:

```
delete( $asso{'c'} ); (deletes one pair)
delete( @asso{'a', 'c'} ); (deletes a list of pairs)
```

The keys (values) function produces an array of keys (values) of a given hash:

```
@all_keys = keys( %asso ) ## ('a', 'b', 'c')
```

```
@all_values = values ( %asso ) ## (7, 11, 13)
```

You may turn a hash into a regular array (line 5) and use numerical indexing (lines 6-7):

```
@myarr = %asso; ## (5)
```

```
print "$myarr[0]\n"; ## a (6)
```

```
print "$myarr[1]\n"; ## 7
```

```
print "$myarr[2]\n"; ## b (7)
```

The built-in association array %ENV contains all the environment variables transmitted to the Perl program.

14.3 Hashes

A hash (or associative array) is an unordered set of key/value pairs whose elements are indexed by their keys. Hash variable names have the form %foo.

Hash Variables and Literals

A literal representation of a hash is a list with an even number of elements (key/value pairs, remember?).

```
%foo = qw( fred wilma barney betty );
```

```
%foo = @foolist;
```

To add individual elements to a hash, all you have to do is set them individually:

```
$foo{fred} = "wilma";
```

```
$foo{barney} = "betty";
```

You can also access slices of hashes in a manner similar to the list case:

```
@foo{"fred", "barney"} = qw( wilma betty );
```


Hash Functions

The keys function returns a list of all the current keys for the hash in question.

```
@hashkeys = keys(%hash);
```

As with all other built-in functions, the parentheses are optional:

```
@hashkeys = keys %hash;
```

This is often used to iterate over all elements of a hash:

```
foreach $key (keys %hash) {  
  
    print $hash{$key}."\n";  
  
}
```

In a scalar context, the keys function gives the number of elements in the hash.

Conversely, the values function returns a list of all current values of the argument hash:

```
@hashvals = values(%hash);
```

The each function provides another means of iterating over the elements in a hash:

```
while (($key, $value) = each (%hash)) {  
  
    statements;  
  
}
```

You can remove elements from a hash using the delete function:

```
delete $hash{'key'};
```

14.4 & 14.5 References and Functions

Most serious programs require the definition of functions that can be called from anywhere.

You define functions with the sub keyword. A function can be placed anywhere in your

Perl source code file. Usually all functions are placed at the end of the file. For substantial

programming, functions and objects can be placed in separate packages or modules and then imported into a program with the use statement.

The general form of a function is:

```
sub functionName
```

```
{
```

```
  a sequence of statements
```

```
}
```

A call to the above may take any of these forms:

```
functionName(); ## no arg
```

```
functionName($a); ## one arg
```

```
functionName($aa, $bb); ## two args
```

```
functionName(@arr); ## array elements as args
```

```
functionName($aa, @arr); ## $aa and array elements as args
```

A function gets in-coming arguments in the special array `@_`. In a function definition, the

notations `$_[0]`, `$_[1]`, `$_[2]` are used to access the individual arguments.

Arguments, scalars, arrays and hashes, in a function call are passed to the receiving function as a at list.

Consider the function call

```
myfunc($total, @toys)
```

In `myfun`, `$_[0]` is `$x`; `$_[1]` is `$toys[0]`; `$_[2]` is `$toys[1]`; and so on. Furthermore, each `$_[i]` is a reference to the argument passed and modifying it will alter the data in the calling program.

To obtain a local copy of the passed arguments use for example `sub myfun`

```
{ my($a, $b, $c) = @_; // $a, $b, $c local to myfun
```

```
... // and have copies of passed data
```

```
}
```

Here \$a gets a copy of \$_[0], \$b a copy of \$_[1] and so on.

Use return value; to return a value for a function. If a function returns without executing a return, then the value is that of the last statement. Sometimes you need to include arrays and hashes in a function call unattended. This can be done by passing references to the arrays and hashes. In general, a reference is a symbol that leads to the construct to which it refers (like a pointer). References are scalars and are passed in function calls as such.

References are not hard to understand. The following points will help.

- Put a backslash in front of a variable to obtain a reference: \$ref_x = \\$x, ref_x = \@x, or ref_x = \%x.
- Now \$ref_x is a reference and can be used just like the symbol x to which it refers.
- The notation \$\$ref_x is the same as \$x, @\$ref_x is the same as @x, and %\$ref_x is the same as %x.

The following Perl program (Ex: PerlRef) shows a scalar \$x, an array @arr and a hash

%asso, how their references are obtained, and used. #!/usr/bin/perl

```
\ my $x = 3.1416, @arr = ("a", "b");
```

```
my %asso = ("one" => 7, "two" => 11);
```

```
my $ref_x = \$x; // three references
```

```
my $ref_foo = \@arr;
```

```
my $ref_bar = \%asso;
```

```
// using references
```

```
print "$$ref_x\n"; // 3.1416
```

```
print "$$ref_foo[1]\n"; // b
```

```
print "$$ref_bar{'two'}\n"; // 11
```

When references are passed in function calls, they can be used in the called function exactly

the same way.

Local Variables in Functions

In Perl, all variables are global within its module (source code file), unless declared local. In

a subroutine local variables are declared with either the keywords `my` or `local`.

- `local(var1, var2, ...)`; (dynamic nesting)
- `my(var1, var2, ...)`; (static lexical scoping)

A variable declared by `my` is known only within the function or source code file, in the same sense as local variables in C, C++, or Java. A variable declared by `local` is known within the function and other function it calls at run-time, in the same sense as `prog` variables in Common LISP. For most purposes, the `my` declaration will suffice.

As an example, let's write a function `htmlBegin`. This can be a handy CGI utility that sends out the leading part of the HTML code for an HTTP response (Ex: `HtmlBegin`).

The function receives two arguments: the name of a page title and a filename `$frontfile`.

The `$frontfile` is a partial HTML template that can be customized for the look and feel of a particular website.

```
sub htmlBegin
{
    my $title=$_[0]; ## page title

    my $frontfile=$_[1]; ## HTML template

    my $line;

    print "Content-type: text/html\r\n\r\n";

    if ( defined($frontfile) )
    { open(IN, $frontfile) || die "Can't open $frontfile";

    while ( $line=<IN> )
```

```

{ if ( $line =~ /XYZZZ/ )

{ $line =~ s/XYZZZ/$title/; } ## (1)

if ( $line =~ /XHTMLFRONT/ )

{ $line =~ s/XHTMLFRONT/$xhtml_front/; } ## (2)

print $line;

}

close(IN);

}

else

{ print ( "$xhtml_front" ## (3)

. "<head><title>$title</title>"

. "</head><body>\n" );

}

}

```

The variable \$xhtml front is set to the first lines needed for an XHTML file earlier in this program.

A sample call to this function is

```
htmlBegin("Request Confirmation", "webtong.front");
```

where the page title and an "HTML template" are supplied. The file webtong.front

XHTMLFRONT

```
<head><title>Webtong - XYZZZ</title>
```

```
<base href="http://www.webtong.com/feedback.html" />
```

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

```
</head><body>
```

```
<table class="layout" border="0" cellspacing="0"
```

```
cellpadding="0" width="100%">
```

```
<!-- top banner begin -->
```

```
...
```

```
<!-- top banner end -->
```

```
<h2>XYZZZ</h2></tr><tr>
```

```
<!-- left nav bar begin -->
```

14.6 Summary

In this lesson, you have learned the following:

- Perl is freely available from the Internet.
- Many resources are readily available.
- Basic statement syntax is similar to that of C Programming Language in terms of statements, operators and control constructs.
- Several variable types are supported by Perl.
- Perl has numerous built-in functions.
- The `chomp()` and `chop()` command is used to remove last character of strings.
- Numerical values equate to boolean true if nonzero and boolean false if zero.
- The if-else and unless-else statements are used to implement two-way branching.
- The `elsif` statement can be used to implement multi-way branching without excessive indentation.
- The while and for loops are entry-condition loops.
- The if, unless, while, and until statements have single-line forms.
- The foreach loop is another form of the for loop which is used for processing lists.
- The `warn()` and `die()` functions can be used to send messages to the user through standard error.

- The `exit()` function causes the program to terminate immediately.
- The `last`, `next`, and `redo` statements control program flow within a loop.

14.7 Keywords

Hashes, Array, Function, `for`, `while` etc...

14.8 Exercise

- Write a program which asks the user what language they would like to be greeted in. Give them a choice of English (Hello), Australian (G'day), Spanish (Buenos dias), or French (Bon jour). After making a selection, print out the corresponding greeting to standard output. Use
- The `die()` function to terminate the program with a message if the user selects an unknown language code.
- Modify the program in exercise 1 to greet a list of people (John, Duncan, Hector, and Rene) hardcoded into a `foreach` list with a language code following each name. Greet each person on a separate line of standard output with the appropriate greeting. Use the `warn()` function to notify the user if an unknown language code is specified.
- Write a program which generates a Fibonacci series of a length specified by the user. The Fibonacci series starts with 1 and 1 and always creates the next number in the series by summing the previous two. The series begins with: 1, 1, 2, 3, 5, 8, .

14.9 References

- M. Deitel, P.J. Deitel, A.B. Goldberg: Internet & World Wide Web How to program, 3rd Edition, Pearson Education/ PHI, 2004.
- Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.

UNIT 15-: The Basics of Perl and CGI

Structure

- 15.0 Learning Objectives
- 15.1 Pattern matching
- 15.2 File input and output
- 15.3 The Common Gateway Interface
- 15.4 CGI linkages
- 15.5 Query string format
- 15.6 CGI.pm module
- 15.7 A survey example
- 15.8 Cookies
- 15.9 Summary
- 15.10 Keywords
- 15.11 Exercises
- 15.12 References

15.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Understand about Pattern matching
- Understand about File input and output
- Understand about The Common Gateway Interface
- Understand about CGI linkages

15.1 Pattern Matching

Regular Expressions

Regular expressions are patterns to be matched against a string. The two basic operations performed using patterns are matching and substitution:

Matching */pattern/*

Substitution *s/pattern/newstring/*

The simplest kind of regular expression is a literal string. More complicated expressions include *metacharacters* to represent other characters or combinations of them.

The [...] construct is used to list a set of characters (a *character class*) of which *one* will match. Ranges of characters are denoted with a hyphen (-), and a negation is denoted with a circumflex (^). Examples of character classes are shown below:

[a-zA-Z] Any single letter

[0-9] Any digit

[^0-9] Any character **not** a digit

Some common character classes have their own predefined symbols:

Code Matches

. Any character

\d A digit, such as [0-9]

\D A nondigit, same as [^0-9]

\w A word character (alphanumeric) [a-zA-Z_0-9]

\W A nonword character [^a-zA-Z_0-9]

\s A whitespace character [\t\n\r\f]

\S A non-whitespace character [^\t\n\r\f]

Regular expressions also allow for the use of both variable interpolation and *backslashed representations* of certain characters:

Code Matches

\n Newline

\r Carriage return

\t Tab

\f Formfeed

\/ Literal forward slash

Anchors don't match any characters; they match places within a string.

Assertion Meaning

^ Matches at the beginning of string

\$ Matches at the end of string

\b Matches on word boundary

\B Matches except at word boundary

\A Matches at the beginning of string

\Z Matches at the end of string or before a newline

\z Matches only at the end of string

Quantifiers are used to specify how many instances of the previous element can match.

Maximal Minimal Allowed Range

{n,m} **{n,m}?** Must occur at least **n** times, but no more than **m** times

{n,} **{n,}?** Must occur at least **n** times

{n} **{n}?** Must match exactly **n** times

*** *?** 0 or more times (same as **{0,}**)

+ +? 1 or more times (same as **{1,}**)

? ?? 0 or 1 time (same as **{0,1}**)

It is important to note that quantifiers are greedy by nature. If two quantified patterns are represented in the same regular expression, the leftmost is greediest. To force your quantifiers to be non-greedy, append a question mark.

If you are looking for two possible patterns in a string, you can use the alternation operator (**|**). For example,

/you|me|him|her/;

will match against any one of these four words. You may also use parentheses to provide boundaries for alternation:

/And(y|rew)/;

will match either “Andy” or “Andrew”.

Parentheses are used to group characters and expressions. They also have the effect of “remembering” parts of a matched pattern for further processing. To recall the “memorized” portion of the string, include a backslash followed by an integer representing the location of the parentheses in the expression:

/fred(.)barney\1/;

Outside of the expression, these “memorized” portions are accessible as the special variables **\$1**, **\$2**, **\$3**, etc. Other special variables are as follows:

\$& Part of string matching regexp

\$` Part of string *before* the match

\$' Part of string *after* the match

Regular expression grouping precedence Parentheses **()** **(?:)** Quantifiers **?** **+** ***** **{m,n}** **??** **+** **?** ***?** Sequence and anchoring **abc** **^** **\$** **\A** **\Z** **(?=)** **(?!)**

Alternation **|**

To select a target for matching/substitution other than the default variable (**\$_**), use the

`=~` operator:

`$var =~ /pattern/;`

Operators

`m/pattern/gimosx`

The “match” operator searches a string for a pattern match. The preceding “m” is usually omitted. The trailing modifiers are as follows

Modifier Meaning

g Match globally; find all occurrences

i Do case-insensitive matching

m Treat string as multiple lines

o Only compile pattern once

s Treat string as a single line

x Use extended regular expressions

`s/pattern/replacement/egimosx`

Searches a string for a pattern, and replaces any match with replacement. The trailing modifiers are all the same as for the match operator, with the exception of “e”, which evaluates the right-hand side as an expression. The substitution operator works on the default variable (`$_`), unless the `=~` operator changes the target to another variable.

`tr/pattern1/pattern2/cds`

This operator scans a string and, character by character, replaces any characters matching **pattern1** with those from **pattern2**. Trailing modifiers are:

Modifier Meaning

c Complement pattern1

d Delete found but unreplaced characters

s Squash duplicated replaced characters

This can be used to force letters to all uppercase:

```
tr/a-z/A-Z/;
```

```
@fields = split(pattern,$input);
```

Split looks for occurrences of a regular expression and breaks the input string at

those points. Without any arguments, split breaks on the whitespace in \$_:

```
@words = split; is equivalent to
```

```
@words = split(/\s+/, $_);
```

```
$output = join($delimiter,@inlist);
```

Join, the complement of split, takes a list of values and glues them together with

the provided delimiting string.

15.2 Files and I/O

Filehandles

A filehandle is the name for the connection between your Perl program and the operating system. Filehandles follow the same naming conventions as labels, and occupy their own namespace. Every Perl program has three filehandles that are automatically opened for it: STDIN,

STDOUT, and STDERR:

STDIN Standard input (keyboard or file)

STDOUT Standard output (print and write send output here)

STDERR Standard error (channel for diagnostic output)

Filehandles are created using the open() function:

```
open(FILE,"filename");
```

You can open files for reading, writing, or appending:

open(FILE,"> newout.dat") Writing, creating a new file

open(FILE,">> oldout.dat") Appending to existing file

open(FILE,"< input.dat") Reading from existing file

As an aside, under Windows, there are a number of ways to refer to the full path to a file:

"c:\\temp\\file" Escape the backslash in double quotes

'c:\\temp\\file' Use proper path in single quotes

"c:/temp/file" UNIX-style forward slashes

It is important to realize that calls to the **open()** function are not always successful. Perl will not (necessarily) complain about using a filehandle created from a failed **open()**.

This is why we test the condition of the open statement:

open(F,"< badfile.dat") or die "open: \$!"

You may wish to test for the existence of a file or for certain properties before opening it.

Fortunately, there are a number of file test operators available:

File test Meaning

-e file File or directory exists

-T file File is a text file

-w file File is writable

-r file File is readable

-s file File exists and has nonzero length

These operators are usually used in a conditional expression:

```
if (-e myfile.dat) {  
  
open(FILE,"< myfile.dat") or die "open: $!\n";  
  
}
```

Even more information can be obtained about a given file using the **stat()** function

Using file handles

After a file has been opened for reading you can read from it using the diamond operator

just as you have already done for STDIN:

```
$_ = <FILE>; or
```

```
while (<FILE>) {
```

```
statements;
```

```
}
```

To print to your open output file, use the filehandle as the first argument to the print statement (N.B. no commas between the filehandle and the string to print).

```
print FILE "Look Ma! No hands!\n";
```

To change the default output filehandle from STDOUT to another one, use select:

```
select FILE;
```

From this point on, all calls to print or write without a filehandle argument will result in output being directed to the selected filehandle. Any special variables related to output will also then apply to this new default. To change the default back to STDOUT, select it:

```
select STDOUT;
```

When you are finished using a filehandle, close it using close():

```
close(FILE);
```

Formats

Perl has a fairly sophisticated mechanism for generating formatted output. This involves using pictorial representations of the output, called templates, and the function **write**.

Using a format consists of three operations:

1. Defining the format (template)
2. Loading data to be printed into the variable portions of the format
3. Invoking the format.

Format templates have the following general form:

format FORMATNAME =

fieldline

\$value_one, \$value_two, ...

Everything between the “=” and the “.” is considered part of the format and everything (in the fieldlines) will be printed exactly as it appears (whitespace counts). Fieldlines permit variable interpolation via fieldholders:

Hi, my name is @<<<<<, and I’m @< years old.Fieldline

\$name, \$age Valueline

Fieldholders generally begin with a @ and consist of characters indicated alignment/type.

@<<<< Four character, left-justified field

@>>>> Four character, right-justified field

@||| Four character, centered field

@###.## Six character numeric field, with two decimal places

@* Multi-line field (on line by itself – for blocks of text)

^<<<< Five character, “filled” field (“chews up” associated variables)

The name of the format corresponds to the name of a filehandle. If write is invoked on a filehandle, then the corresponding format is used. Naturally then, if you’re printing to standard output, then your format name should be STDOUT. If you want to use a format

with a name other than that of your desired filehandle, set the \$~ variable to the format name.

There are special formats which are printed at the top of the page. If the active format name is FNAME, then the “top” format name is FNAME_TOP. The special variable \$% keeps a count of how many times the “top” format has been called and can be used to number pages.

Manipulating files & directories

The action of opening a file for writing creates it. Perl also provides functions to manipulate files without having to ask the operating system to do it.

unlink(filename)

Delete an existing file. Unlink can take a list of files, or wildcard as an argument as well: unlink(<*.bak>)

rename(oldname, newname)

This function renames a file. It is possible to move files into other directories by specifying a path as part of the new name. Directories also have some special function associated with them **mkdir(dirname, mode)**. Create a new directory. The “mode” specifies the permissions (set this to 0777 to be safe).

rmdir(dirname)

Removes (empty) directories

chdir(dirname)

Change current working directory to dirname. File and directory attributes can be modified as well:

chmod(permission, list of files)

Change the permissions of files or directories:

666 = read and write

444 = read only

777 = read, write, and executable

utime(ctime, mtime, list of files)

Modify timestamps on files or directories. “atime” is the time of the most recent access, and “mtime” is the time the file/directory was last modified.

15.3 Common Gateway Interfaces

Perl is the most commonly used language for CGI programming on the World Wide Web. The Common Gateway Interface (CGI) is an essential tool for creating and managing comprehensive websites. With CGI, you can write scripts that create interactive, user-driven applications.

Simple CGI Programs

CGI programs are invoked by accessing a URL (uniform resource locator) describing their coordinates:

`http://www.mycompany.com/cgi-bin/program.plx`

even though the actual location of the script on the hard drive of the server might be something like:

`c:\webserver\bin\program.plx`

The simplest CGI programs merely write HTML data to STDOUT (which is then displayed by your browser):

```
print << ENDOFTEXT;
```

```
Content-type: text/html
```

```
<HTML>
```

```
<HEAD><TITLE>Hello, World!</TITLE></HEAD>
```

```
<BODY>
```

```
<H1>Hello, World!</H1>
```

```
</BODY>
```

```
</HTML>
```

```
ENDOFTEXT
```

15.4 CGI Linkage

CGI programs often are stored in a directory named

cgi-bin

Some CGI programs are in machine code, but Perl programs are usually kept in source form, so perl must be run on them. A source file can be made to be “executable” by adding a line at their beginning that specifies that a language processing program be run on them

First For Perl programs, if the perl system is stored in /usr/local/bin/perl, as is often is in UNIX systems, this is `#!/usr/local/bin/perl -w`. An HTML document specifies a CGI program with the hypertext reference attribute, href, of an anchor tag, `<a>`, as in `<a href =`

`"http://www.cs.uccs.edu/cgi-bin/reply.pl">`

Click here to run the CGI program, reply.pl

``

`<!-- reply.html - calls a trivial cgi program-->`

`<html>`

`<head>`

`<title>`

HTML to call the CGI-Perl program reply.pl

`</title>`

`</head>`

`<body>`

This is our first CGI-Perl example

`<a href =`

`"http://www.cs.ucp.edu/cgi-bin/reply.pl">`

Click here to run the CGI program, reply.pl

</body>

</html>

The connection from a CGI program back to the requesting browser is through standard output, usually through the server. The HTTP header needs only the content type, followed by a blank line, as is created with:

```
print "Content-type: text/html \n\n";
```

```
#!/usr/local/bin/perl
```

```
# reply.pl – a CGI program that returns a
```

```
# greeting to the user
```

```
print "Content-type: text/html \n\n",
```

```
"<html> <head> \n",
```

```
"<title> reply.pl example </title>",
```

```
" </head> \n", "<body> \n",
```

```
"<h1> Greetings from your Web server!",
```

```
" </h1> \n </body> </html> \n";
```

15.5 Query String Format

A query string includes names and values of widgets. Widget values are always coded as strings. The form of a name/value pair in a query string is:

name=value

If the form has more than one widget, their values are separated with ampersands milk=2&payment=visa. Each special character is coded as a percent sign and a two character hexadecimal number (the ASCII code for the character). Some browsers code spaces a plus signs, rather than as %20

15.6 The CGI.pm Module

A Perl module serves as a library the Perl use declaration is used to make a module available to a program. To make only part of a module available, specify the part name after a colon (For our purposes, only the standard part of the CGI module is needed) use `CGI ":standard"`; Common CGI.pm Functions “Shortcut” functions produce tags, using their parameters as attribute values.

e.g., `h2("Very easy!");` produces

```
<h2> Very easy! </h2>
```

In this example, the parameter to the function `h2` is used as the content of the `<h2>` tag

Tags can have both content and attributes. Each attribute is passed as a name/value pair, just as in a hash literal. Attribute names are passed with a preceding dash

```
textarea(-name => "Description",
```

```
rows => "2",cols => "35"
```

```
);
```

Produces:

```
<textarea name="Description" rows=2
```

```
cols=35> </textarea>
```

If both content and attributes are passed to a function, the attributes are specified in a hash literal as the first parameter `a({-href => "fruit.html"}, "Press here for fruit descriptions");`

Output: ` Press here for fruit descriptions`

Tags and their attributes are distributed over the parameters of the function

```
ol(li({-type => "square"},
```

```
["milk", "bread", "cheese"]));
```

Output: ``

```
<li type="square"milk</li>
```

```
<li type="square"bread</li>
```

```
<li type="square"cheese</li>
```

```
</ol>
```

CGI.pm also includes non-shortcut functions, which produce output for return to the user

A call to header() produces:

```
Content-type: text/html;charset=ISO-8859-1
```

```
-- blank line --
```

The start_html function is used to create the head of the return document, as well as the

<body> tag

The parameter to start_html is used as the title of the document

```
start_html("Bill's Bags");
```

```
DOCTYPE html PUBLIC
```

```
"-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
"DTD/xhtml11-transitional.dtd">
```

```
<html xmlns=
```

```
"http://www.w3.org/1999/xhtml lang="en-US">
```

```
<head><title>Bill's Bags</title>
```

```
</head><body>
```

The param function is given a widget's name; it returns the widget's value

If the query string has name=Abraham in it, param("name") will return "Abraham"

The end_html function generates </body></html>

SHOW popcorn.html , its display, and popcorn.pl

15.7 A Survey Example

We will use a form to collect survey data from users. The program needs to accumulate survey results, which must be stored between form submissions. Store the current results in a file on the server. Because of concurrent use of the file, it must be protected from corruption by blocking other accesses while it is being updated. Under UNIX, this can be done with the Perl function, flock, using the parameter value 2 to specify a lock operation and 8 to specify an unlock operation

--> SHOW conelec.html and its display

Two CGI programs are used for this application, one to collect survey submissions and record the new data, and one to produce the current totals. The file format is eight lines, each having seven values, the first four for female responses and the last four for male responses. The program to collect and record form data must:

1. Decode the data in the query string
2. Determine which row of the file must be modified
3. Open, lock, read, unlock, and close the survey data file
4. Split the affected data string into numbers and store them in an array
5. Modify the affected array element and join the array back into a string
6. Open, lock, write, unlock, and close the survey data file

--> SHOW conelec1.pl

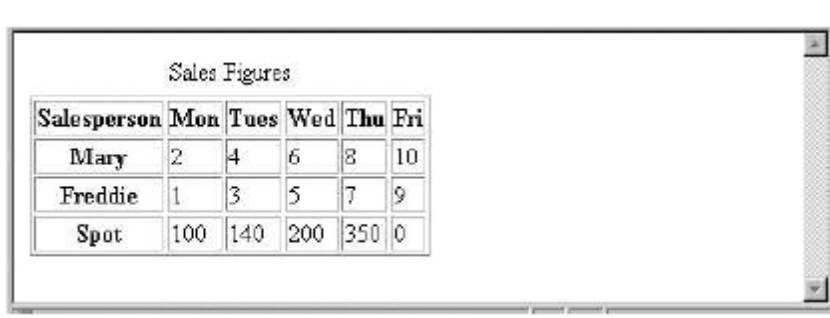
Tables are easier to specify with CGI.pm. The table is created with the table function. The border attribute is specified as a parameter. The table's caption is created with a call to caption, as the second parameter to table. Each row of the table is created with a call to

Tr.

- i. A heading row is created with a call to th
- ii. Data cells are created with calls to td
- iii. The calls to Tr, th, and td require references as parameters
- iv. Suppose we have three arrays of sales numbers, one for each of three salespersons; each array has one value for each day of the work week

We want to build a table of this information, using CGI.pm

```
table({-border => "border"},  
  
caption("Sales Figures"),  
  
Tr(  
  
[th(["Salesperson", "Mon", "Tues",  
  
"Wed", "Thu", "Fri"]),  
  
th("Mary").td(\@marysales),  
  
th("Freddie").td(\@freddiesales),  
  
th("Spot").td(\@spotsales),  
  
]  
  
)  
  
)
```



The screenshot shows a web browser window with a table titled "Sales Figures". The table has 6 columns: Salesperson, Mon, Tues, Wed, Thu, and Fri. The rows are Mary, Freddie, and Spot. The data values are: Mary (2, 4, 6, 8, 10), Freddie (1, 3, 5, 7, 9), and Spot (100, 140, 200, 350, 0).

Salesperson	Mon	Tues	Wed	Thu	Fri
Mary	2	4	6	8	10
Freddie	1	3	5	7	9
Spot	100	140	200	350	0

The program that produces current results must:

1. Open, lock, read the lines into an array of strings, unlock, and close the data file
2. Split the first four rows (responses from females) into arrays of votes for the four age groups
3. Unshift row titles into the vote rows (making them the first elements)
4. Create the column titles row with th and put its address in an array
5. Use td on each rows of votes
6. Push the addresses of the rows of votes onto the row address array
7. Create the table using Tr on the array of row addresses

8. Repeat Steps 2-7 for the last four rows of data (responses from males)

➔ SHOW conelec2.pl

15.8 Cookies

A session is the collection of all of the requests made by a particular browser from the time the browser is started until the user exits the browser. The HTTP protocol is stateless but, there are several reasons why it is useful for the server to relate a request to a session Shopping carts for many different simultaneous customers, Customer profiling for advertising, Customized interfaces for specific clients, Approaches to storing client information:

- Store it on the server – too much to store!
- Store it on the client machine - this works A cookie is an object sent by the server to the client
- Cookies are created by some software system on the server (maybe a CGI program)
- Every HTTP communication between the browser and the server includes information in its header about the message
- At the time a cookie is created, it is given a lifetime
- Every time the browser sends a request to the server that created the cookie, while the cookie is still alive, the cookie is included
- A browser can be set to reject all cookies
- CGI.pm includes support for cookies `cookie(-name => a_cookie_name,`
`-value => a_value,`
`-expires => a_time_value);`
- The name can be any string
- The value can be any scalar value
- The time is a number followed by a unit code (d, s, m, h, M, y)

- Cookies must be placed in the HTTP header at the time the header is created `header(-cookie => $my_cookie);`

- To fetch the cookies from an HTTP request, call `cookie` with no parameters

- A hash of all current cookies is returned

- To fetch the value of one particular cookie, send the cookie's name to the `cookie` function

```
$age = cookie(age);
```

- Example:

A cookie that tells the client the time of his or her last visit to this site

- Use the Perl function, `localtime`, to get the parts of time

```
($sec, $min, $hour, $mday, $mon, $year,
```

```
$wday, $yday, $isdst) = localtime;
```

→`SHOW day_cookie.pl`

15.9 Summary

In this lesson, you have learned:

- Perl uses extended regular expressions to assist in defining patterns.
- The match operators, `=~` and `!~` are used to test for a pattern contained in a string.
- Escape characters are used to represent commonly used character sets and pattern anchors.
- The substitution operator is used to replace and delete subpatterns in a string.
- The translation operator is used to convert between sets of characters.
- Perl supports redirection from the command line.
- Files can be opened for read, write, or append operations.
- A file is opened for a read operation if no other operation is specified.
- File handles are used to reference open files.
- The `<>` operator is used to read from a list of files specified on the command line.
- The `close()` function is used to close a file.
- File test operators can be used to determine a variety of file conditions.

- Pipes can be opened to move data between a Perl process and another shell process.

15.10 Keywords

Opening a File, Reading Lines from a File, Opening a File for Write and Append, Closing a File, The printf() and sprintf() functions, Controlling Output Buffers, The rename() and unlink functions, Using File Test Operators

15.11 Exercise

- What does it mean to append to a file?
 - Show an example of opening a file for appending.
 - Show an example function call to convert an integer to a hex string, and save it in a variable.
 - What function can you use to change the name of a file?
 - What functions can you use to remove a file?
 - What operator can be used to determine if a file exists and has read permission?
 - Write a one line program to get the number of days since file a_file was last modified.
 - Write a program that mimics a basic Unix egrep utility, that is, one which accepts a pattern (string representing an extended regular expression) and a list of files from the command line and prints out all lines which contain the pattern.
 - Write a program which accepts a word and a file name from the command line and lists the line number and position in each line in which the word occurs.
 - Write a program which accepts a list of file names from the command line and strips out all characters except numbers. Count the number of times each individual digit occurs.
 - Write a program which mimics the javadoc utility by printing only the text in a file which falls between the /* * and */ comment delimiters.
-

15.12 References

- M. Deitel, P.J. Deitel, A.B. Goldberg: Internet & World Wide Web How to program, 3rd Edition, Pearson Education/ PHI, 2004.
- Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.

UNIT 16-: Servlets and JSP

Structure

- 16.0 Learning Objectives
- 16.1 Introduction
- 16.2 Servlets and Java Server Pages
- 16.3 Overview of Servlets
- 16.4 Servlet details
- 16.5 A survey example, storing information on Clients
- 16.6 Java Server Pages
- 16.7 Summary
- 16.8 Keywords
- 16.9 Exercises
- 16.10 References

16.0 LEARNING OBJECTIVES

After studying this unit, you will be able to

- Understand
- Advantage
- The concept of

16.1. Introduction

Servlets are modules that run inside request/response-oriented servers, such as Java-enabled web servers. Functionally they operate in a very similar way to CGI scripts, however, being Java based they are more platform independent

16.2 Servlets and Java Server Pages

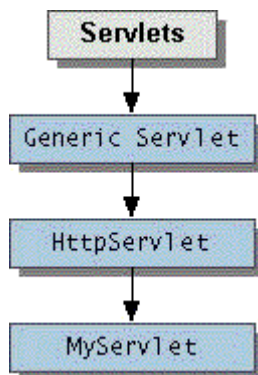
Some Example Applications

A few of the many applications for servlets include,

- Processing data POSTed over HTTPS using an HTML form, including purchase order or credit card data. A servlet like this could be part of an order-entry and processing system, working with product and inventory databases, and perhaps an on-line payment system.
- Allowing collaboration between people. A servlet can handle multiple requests concurrently; they can synchronize requests to support systems such as on-line conferencing.
- Forwarding requests. Servlets can forward requests to other servers and servlets. This allows them to be used to balance load among several servers that mirror the same content. It also allows them to be used to partition a single logical service over several servers, according to task type or organizational boundaries.

Servlet Architecture Overview

The central abstraction in the Servlet API is the Servlet interface. All servlets implement this interface, either directly or, more commonly, by extending a class that implements it such as `HttpServlet`. The inheritance hierarchy looks as follows.



The Servlet interface provides the following methods that manage the servlet and its communications with clients.

- **destroy()**
Cleans up whatever resources are being held and makes sure that any persistent state is synchronized with the servlet's current in-memory state.
- **getServletConfig()**
Returns a servlet config object, which contains any initialization parameters and startup configuration for this servlet.
- **getServletInfo()**
Returns a string containing information about the servlet, such as its author, version, and copyright.
- **init(ServletConfig)**
Initializes the servlet. Run once before any requests can be serviced.
- **service(ServletRequest,ServletResponse)**
Carries out a single request from the client.

Servlet writers provide some or all of these methods when developing a servlet.

When a servlet accepts a service call from a client, it receives two objects, ServletRequest and ServletResponse. The ServletRequest class encapsulates the communication from the client to the server, while the ServletResponse class encapsulates the communication from the servlet back to the client.

The ServletRequest interface allows the servlet access to information such as the names of the parameters passed in by the client, the protocol (scheme) being used by the client, and the names of the remote host that made the request and the server that received it. It also provides the servlet with access to the input stream, ServletInputStream, through which the servlet gets data from clients that are using application protocols such as the HTTP POST and PUT methods. Subclasses of ServletRequest allow the servlet to retrieve more protocol-specific data. For example, HttpServletRequest contains methods for accessing HTTP-specific header information.

The `ServletResponse` interface gives the servlet methods for replying to the client. It allows the servlet to set the content length and mime type of the reply, and provides an output stream, `ServletOutputStream`, and a `Writer` through which the servlet can send the reply data. Subclasses of `ServletResponse` give the servlet more protocol-specific capabilities. For example, `HttpServletResponse` contains methods that allow the servlet to manipulate HTTP-specific header information.

The classes and interfaces described above make up a basic Servlet. HTTP servlets have some additional objects that provide session-tracking capabilities. The servlet writer can use these APIs to maintain state between the servlet and the client that persists across multiple connections during some time period.

Servlet Lifecycle

Servers load and run servlets, which then accept zero or more requests from clients and return data to them. They can also remove servlets. These are the steps of a servlets lifecycle. The next paragraphs describe each step in more detail, concentrating on concurrency issues.

When a server loads a servlet, it runs the servlet's `init` method. Even though most servlets are run in multi-threaded servers, there are no concurrency issues during servlet initialization. This is because the server calls the `init` method once, when it loads the servlet, and will not call it again unless it is reloading the servlet. The server can not reload a servlet until after it has removed the servlet by calling the `destroy` method. Initialization is allowed to complete before client requests are handled (that is, before the service method is called) or the servlet is destroyed.

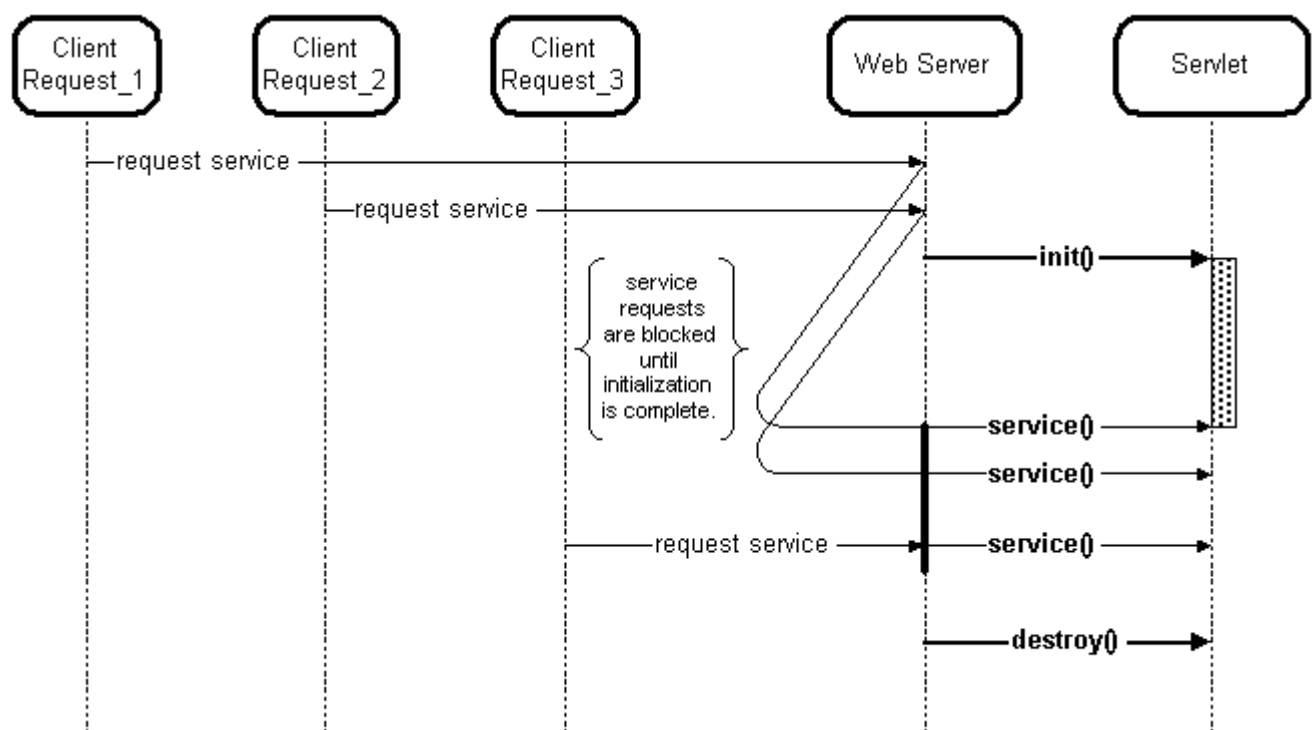
After the server loads and initializes the servlet, the servlet is able to handle client requests. It processes them in its service method. Each client's request has its call to the service method run in its own servlet thread: the method receives the client's request, and sends the client its response.

Servlets can run multiple service methods at a time. It is important, therefore, that service methods be written in a thread-safe manner. If, for some reason, a server should not run multiple service methods concurrently, the servlet should implement

the `SingleThreadModel` interface. This interface guarantees that no two threads will execute the servlet's service methods concurrently.

Servlets run until they are removed from the service. When a server removes a servlet, it runs the servlet's destroy method. The method is run once; the server will not run it again until after it reloads and reinitializes the servlet.

The diagram below shows the basic interactions between clients, a web server, and a servlet registered with the web server.



When the destroy method runs, however, other threads might be running service requests. If, in cleaning up, it is necessary to access shared resources, that access should be synchronized. During a servlet's lifecycle, it is important to write thread-safe code for destroying the servlet and, unless the servlet implements the `SingleThreadModel` interface, servicing client requests.

Writing the Servlet

Servlets implement the `javax.servlet.Servlet` interface. While servlet writers can develop servlets by implementing the interface directly, this is usually not required. Because most servlets extend

web servers that use the HTTP protocol to interact with clients, the most common way to develop servlets is by specializing the `javax.servlet.http.HttpServlet` class. This tutorial concentrates on describing this method of writing servlets.

The `HttpServlet` class implements the `Servlet` interface by extending the `GenericServlet` base class, and provides a framework for handling the HTTP protocol. Its service method supports standard HTTP/1.1 requests by dispatching each request to a method designed to handle it.

By default, servlets written by specializing the `HttpServlet` class can have multiple threads concurrently running its service method. If, you would like to have only a single thread running a single service method at a time, then in addition to extending the `HttpServlet`, your servlet should also implement the `SingleThreadModel` interface. This does not involve writing any extra methods, merely declaring that the servlet implements the interface. For example,

```
public class SurveyServlet extends HttpServlet

implements SingleThreadModel {

    /* typical servlet code, with no threading concerns

    * in the service method. No extra code for the

    * SingleThreadModel interface. */

}
```

Interacting with Clients

Servlet writers who are developing HTTP servlets that specialize the `HttpServlet` class should override the method or methods designed to handle the HTTP interactions that their servlet will handle. The candidate methods include,

- `doGet`, for handling GET, conditional GET and HEAD requests
- `doPost`, for handling POST requests

- doPut, for handling PUT requests
- doDelete, for handling DELETE requests

By default, these methods return a BAD_REQUEST (400) error. An example HTTP servlet that handles GET and HEAD requests follows; it specializes the doGet method. The second example is also provided. It handles POST requests from a form by specializing the doPost method.

The HttpServlet's service method, by default, also calls the doOptions method when it receives an OPTIONS request, and doTrace when it receives a TRACE request. The default implementation of doOptions automatically determines what HTTP options are supported and returns that information. The default implementation of doTrace causes a response with a message containing all of the headers sent in the trace request. These methods are not typically overridden.

Whatever method you override, it will take two arguments. The first encapsulates the data from the client, and is an HttpServletRequest. The second encapsulates the response to the client, and is an HttpServletResponse. The following paragraphs discuss their use.

An HttpServletRequest object provides access to HTTP header data, such as any cookies found in the request and the HTTP method with which the request was made. It, of course, allows the you to obtain the arguments that the client sent as part of the request. How you access the client data might depend on the HTTP method of the request.

- For any HTTP method, you can use the getParameterValues method, which will return the value of a named parameter. (The method getParameterNames provides the names of the parameters.) You can also manually parse the request.
- For requests using the HTTP GET method, the getQueryString method will return a String to be parsed.
- For HTTP methods POST, PUT, and DELETE, you have the choice between two methods. If you expect text data, then it can be read using the BufferedReader returned by

the `getReader` method, if you expect binary data, then it should be read with the `ServletInputStream` returned by the `getInputStream` method.

Note that you should use either the `getParameterValues` method or one of the methods that allow you to parse the data yourself. They cannot be used together in a single request.

For responding to the client, an `HttpServletResponse` object provides two ways of returning the response data to the user. You can use the writer returned by the `getWriter` method or the output stream returned by the `getOutputStream` method. You should use `getWriter` to return text data to the user, and `getOutputStream` for binary data.

Before accessing the `Writer` or `OutputStream`, HTTP header data should be set. The `HttpServletResponse` class provides methods to access the header data, such as the content type and encoding, and content length of the response. After you set the headers, you may obtain the writer or output stream and send the body of the response to the user. Closing the writer or output stream after sending the response to the client allows the server to know when the response is complete.

Example of an HTTP Servlet that handles the GET and HEAD methods

```
/**
```

```
 * This is a simple example of an HTTP Servlet. It responds to the GET
```

```
 * and HEAD methods of the HTTP protocol.
```

```
 */
```

```
public class SimpleServlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
        throws ServletException, IOException {
```

```
        // set header field first
```

```
        res.setContentType("text/html");
```

```

// then get the writer and write the response data

PrintWriter out = res.getWriter();

out.println("<HEAD><TITLE>SimpleServlet Output</TITLE></HEAD><BODY>");

out.println("<h1> SimpleServlet Output </h1>");

out.println("<P>This is output is from SimpleServlet.");

    out.println("</BODY>");

    out.close();

}

public String getServletInfo() {

    return "A simple servlet";

}

}

```

The example above shows the code for the entire servlet. The `doGet` method, because it is returning text to the client, uses the `HttpServletResponse`'s `getWriter` method. It sets the response header field, content type, before writing the body of the response, and closes the writer after writing the response.

In addition to `doGet`, there is a second method, `getServletInfo`. More information on the `getServletInfo` method appears in [later section](#). Because this servlet is an example shipped with the release, it is already compiled. To try the servlet, run it in the [servletrunner](#) section.

Example of an HTTP Servlet that handles the POST method

The following example processes data POSTed by a form. The form looks like this:

```
<html>

<head><title>JdcSurvey</title></head>

<body>

<form action=http://demo:8080/servlet/survey method=POST>

  <input type=hidden name=survey value=Survey01Results>

  <BR><BR>How Many Employees in your Company?<BR>

  <BR>1-100<input type=radio name=employee value=1-100>

  <BR>100-200<input type=radio name=employee value=100-200>

  <BR>200-300<input type=radio name=employee value=200-300>

  <BR>300-400<input type=radio name=employee value=300-400>

  <BR>500-more<input type=radio name=employee value=500-more>

  <BR><BR>General Comments?<BR>

  <BR><input type=text name=comment>

  <BR><BR>What IDEs do you use?<BR>

  <BR>JavaWorkShop<input type=checkbox name=ide value=JavaWorkShop>

  <BR>J++<input type=checkbox name=ide value=J++>

  <BR>Cafe'<input type=checkbox name=ide value=Cafe'>

  <BR><BR><input type=submit><input type=reset>

</form>
```

</body>

</html>

The servlet writes the form data to a file, and responds to the user with a thank you message. The doPost method of the servlet looks like this:

16.3 Overview of Java servlets

Servlets are protocol and platform independent server-side software components, written in Java. They run inside a Java enabled server or application server, such as the WebSphere Application Server. Servlets are loaded and executed within the Java Virtual Machine (JVM) of the Web server or application server, in much the same way that applets are loaded and executed within the JVM of the Web client. Since servlets run inside the servers, however, they do not need a graphical user interface (GUI). In this sense, servlets are also faceless objects. Servlets more closely resemble Common Gateway Interface (CGI) scripts or programs than applets in terms of functionality. As in CGI programs, servlets can respond to user events from an HTML request, and then dynamically construct an HTML response that is sent back to the client.

16.4 Survey examples

In this section, we will build on the foundation in the previous sections, by describing some servlets that demonstrate additional capabilities and concepts of the Java Servlet API.

Simple HTTP servlet

We begin with a look at a very simple servlet, *SimpleHttpServlet*

```
package itso.servjsp.servletapi;

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class SimpleHttpServlet extends HttpServlet {
```

```

protected void service(HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException {

res.setContentType("text/html");

PrintWriter out = res.getWriter();

out.println("<HTML><TITLE>SimpleHttpServlet</TITLE><BODY>");

out.println("<H2>Servlet API Example - SimpleHttpServlet</H2><HR>");

out.println("<H4>This is about as simple a servlet as it gets!</H4>");

out.println("</BODY><HTML>");

out.close();

}

}

```

As the title indicates, *SimpleHttpServlet* is a very simple HTTP servlet that accepts a request and writes a response. Let's break out the components of this servlet so we can discuss them individually.

Basic servlet structure

We have defined this servlet to be part of an *itso.servjsp.servletapi* Java package. This is the naming convention used for all the servlet examples in this chapter. *SimpleHttpServlet package declaration*. The import statements used to give us access to other Java packages. The import of *java.io* is so that we have access to some standard IO classes. More importantly, the *javax.servlet.** and *javax.servlet.http.** import statements give us access to the Java Servlet API set of classes and interfaces.

```

Package itso.servjsp.servletapi;

import java.io.*;

import javax.servlet.*;

```

```

import javax.servlet.http.*;

public class SimpleHttpServlet extends HttpServlet {

protected void service(HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException {

res.setContentType("text/html");

PrintWriter out = res.getWriter();

out.println("<HTML><TITLE>SimpleHttpServlet</TITLE><BODY>");

out.println("<H2>Servlet API Example - SimpleHttpServlet</H2><HR>");

out.println("<H4>This is about as simple a servlet as it gets!</H4>");

out.println("</BODY><HTML>");

out.close();

}

}

```

package itso.servjsp.servletapi; this statements used to give us access to other Java packages. The import of java.io is so that we have access to some standard IO classes. More importantly, the javax.servlet.* and javax.servlet.http.* import statements give us access to the Java Servlet API set of classes and interfaces.

```

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class SimpleHttpServlet extends HttpServlet {

protected void service (HttpServletRequest req, HttpServletResponse res)

```



```

throws ServletException, IOException {

res.setContentType("text/html");

PrintWriter out = res.getWriter();

out.println("<HTML><TITLE>SimpleHttpServlet</TITLE><BODY>");

out.println("<H2>Servlet API Example - SimpleHttpServlet</H2><HR>");

out.println("<H4>This is about as simple a servlet as it gets!</H4>");

out.println("</BODY><HTML>");

out.close();

}

```

Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java page, so we first set the response content type to text/html by coding `res.setContentType("text/html")`. Next, we request a `PrintWriter` object to write text to the response by coding `PrintWriter out = res.getWriter()`. We could also have used a `ServletOutputStream` object to write out our response, but `getWriter` gives us more flexibility with Internationalization. In either case, the content type of the response must be set before references to these objects can be made.

The remaining `out.println` statements write our HTML to the `PrintWriter`, which is sent back to the client as our response. It is pretty simple HTML, so we do not display it here. We use `out.close` more for completeness, because the Web application server automatically closes the `PrintWriter` when the service method exits.

How the servlet gets invoked We could invoke this servlet with either a GET or POST form action method; the service method will execute for either. If we knew something about how

this servlet was ultimately to be called, for instance, what the HTML form method was going to be, we could have implemented the above functionality through specific `doGet` or `doPost` methods. The result would be the same. The simplest way to invoke the servlet would be by specifying a URL in the Web browser. This does not work for every servlet, but would work for the above example. A URL forces the Web browser to send the request using GET, similar to the way a standard HTML page is requested. The above servlet could be invoked from the Web browser with the URL:

`http://host/servlet/itso.servjsp.servletapi.SimpleHttpServlet`

`http://host/itso.servjsp/servlet/itso.servjsp.servletapi.SimpleHttpServlet`

Note: the second form invokes a servlet in a Web application.

Running the servlet

At this point we have not discussed the specifics of running servlets in a Web server environment. If you want to run this servlet, “Development and testing with VisualAge for Java”, code the SimpleHttpServlet, and run it under the WebSphere Test Environment. The WebSphere Test Environment provides a simulated Web server environment within the VisualAge for Java product and enables you to test and debug your servlets. “WebSphere Application Server”, we discuss deploying servlets to the actual application server environment.

Servlets

HTML form generator servlet

We next look at another simple HTTP servlet, HTMLFormGenerator.

HTML form generator servlet

```
package itso.servjsp.servletapi;

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class HTMLFormGenerator extends HttpServlet {

    public void init(ServletConfig config) throws ServletException {

        super.init(config);

        System.out.println("In the init() method of HTMLFormGenerator");

    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```

throws ServletException, IOException {

performTask(req, res, "POST",

"itso.servjsp.servletapi.HTMLFormHandler");

// "/itso.servjsp/servlet/itso.servjsp.servletapi.HTMLFormHandler");

}

public void performTask(HttpServletRequest req, HttpServletResponse res,

String method, String url) throws ServletException, IOException {

res.setContentType("text/html");

PrintWriter out = res.getWriter();

out.println("<HTML><TITLE>HTMLFormGenerator</TITLE><BODY>");

out.println("<H2>Servlet API Example - HTMLCreatingServlet</H2><HR>");

out.println("<FORM METHOD=\"\" + method + \"\" ACTION=\"\" + url + \"\">");

out.println("<H2>Tell us something about yourself: </H2>");

out.println("<B>Enter your name: </B>");

out.println("<INPUT TYPE=TEXT NAME=firstname><BR>");

out.println("<B>Select your title: </B>");

out.println("<SELECT NAME=title>");

out.println("<OPTION VALUE=\"Web Developer\">Web Developer");

out.println("<OPTION VALUE=\"Web Architect\">Web Architect");

out.println("<OPTION VALUE=\"Other\">Other");

out.println("</SELECT><BR>");

out.println("<B>Which tools do you have experience with: </B><BR>");

```

```

out.println("<INPUT TYPE=checkbox NAME=\"tools\"
VALUE=\"WebSphere Application Server\">WebSphere Application Server<BR>");

out.println("<INPUT TYPE=checkbox NAME=\"tools\"
VALUE=\"WebSphere Studio\">WebSphere Studio<BR>");

out.println("<INPUT TYPE=checkbox NAME=\"tools\"
VALUE=\"VisualAge for Java\">VisualAge for Java<BR>");

out.println("<INPUT TYPE=checkbox NAME=\"tools\"
VALUE=\"IBM Http Web Server\">IBM Http Web Server<BR>");

out.println("<INPUT TYPE=checkbox NAME=\"tools\" VALUE=\"DB2 UDB\">DB2 UDB<BR>");

out.println("<INPUTTYPE=\"SUBMIT\"NAME=\"SENDPOST\"NAME=\"SENDPOST\" ");

out.println("</FORM>");

out.println("</BODY><HTML>");

out.close();

System.out.println("In the doGet method");

}

}

```

Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java init method This servlet implements the init method. The init method only prints a message to standard output and call the super-class constructor. As we mentioned before, the init method is called only once, when the servlet is loaded. This message, therefore, should only be printed to the Web server's console or log once (wherever standard output is defined), regardless of how many times the servlet is actually invoked.

doGet method We decided that this servlet is always called through a GET request, we have chosen to implement the doGet method, instead of the more generic service method. We developed a performTask method to which we pass a method posting type and a target URL.

Response object

The HTML page that this servlet generates is a bit more complex than the previous example. It actually builds an HTML form that can be used in the future to call other servlets. This is not the same as a servlet calling a servlet, which is a server-side process. Here, we are just using one servlet to generate the HTML back to the browser, so we can call our other example servlets, and we do not have to create separate HTML files for each servlet. Notice that this servlet has many `out.println` statements. This is just the HTML that is written back to the browser. Despite the size of this servlet, it is still only doing one simple thing, writing HTML output. Invoking the servlet This servlet can be invoked directly by a URL command:

`http://hostname/servlet/itso.servjsp.servletapi.HTMLFormGenerator`

`http://hostname/webappname/servlet/itso.xxxx <== with web application`

Notice the output line for the form that this servlet generates in the `performTask` method:

```
<FORM METHOD="POST" ACTION="itso.servjsp.servletapi.HTMLFormHandler">
```

This line demonstrates another way of invoking a servlet, in this case from a Web browser using a form action event. The form is generated by the `HTMLFormGenerator` servlet.

Note: The relative URL in the action is added to the current prefix of the generating servlet, such as `http://hostname/.../servlet/`.

Servlets Servlet output

The HTML Page that this servlet generates.

HTML form generator servlet: response output. HTML form processing servlet.

We next look at a servlet that processes HTML form data. The `HTMLFormHandler` servlet.

HTML form handler servlet (part 1)

```
package itso.servjsp.servletapi;
```

```
import java.io.*;
```

```
import java.util.*;
```

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```

public class HTMLFormHandler extends HttpServlet {

    public void init (ServletConfig srvCfg) throws ServletException {

        super.init(srvCfg);

    }

```

Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java

HTML form handler servlet (part 2)

```

    public void doPost(HttpServletRequest req, HttpServletResponse res)

        throws ServletException, IOException {

        res.setContentType("text/html"); //must be before first ref to PrintWriter

        PrintWriter out = res.getWriter();

        out.println("<HTML><TITLE>HTMLFormHandler</TITLE></BODY>");

        out.println("<H2>Servlet API Example - HTMLFormHandler</H2><HR>");

        //Retrieving the single-value parameters

        out.println("Hi <B>" + req.getParameter("firstname") + ",</B><P>");

        out.println("I see you are a <B>" + req.getParameter("title") + ",</B><P>");

        out.println("And have worked with the following tools: <BR>");

        //Retrieving the multi-value parameters

        String vals[] = (String []) req.getParameterValues("tools");

        if (vals != null) {

            for(int i = 0; i<vals.length; i++)

                out.println("<B>" + vals[i] + "</B><BR>");

        }

```

```

else out.println("<B> None </B><BR>");

out.println("<HR>");

getReqInfo(req, out); //gets the standard request information

out.println("</BODY></HTML>");

out.close();

}

public void getReqInfo(HttpServletRequest req, PrintWriter out)

throws ServletException, IOException {

out.println("<H4><B>Additional Request Information:</B></H4>");

out.println("<B>Request method:</B> " + req.getMethod() + "<BR>");

out.println("<B>Request URI:</B> " + req.getRequestURI() + "<BR>");

out.println("<B>Request protocol:</B> " + req.getProtocol() + "<BR>");

out.println("<B>Request scheme:</B> " + req.getScheme() + "<BR>");

out.println("<B>Servlet path:</B> " + req.getServletPath() + "<BR>");

out.println("<B>Servlet name:</B> " + req.getServerName() + "<BR>");

out.println("<B>Servlet port:</B> " + req.getServerPort() + "<BR>");

out.println("<B>Path info:</B> " + req.getPathInfo() + "<BR>");

out.println("<B>Path translated:</B> " + req.getPathTranslated() + "<BR>");

out.println("<B>Character encoding:</B> " + req.getCharacterEncoding() + "<BR>");

out.println("<B>Query string:</B> " + req.getQueryString() + "<BR>");

out.println("<B>Content length:</B> " + req.getContentLength() + "<BR>");

out.println("<B>Content type:</B> " + req.getContentType() + "<BR>");

```

```

out.println("<B>Remote user:</B> " + req.getRemoteUser() + "<BR>");

out.println("<B>Remote address:</B> " + req.getRemoteAddr() + "<BR>");

out.println("<B>Remote host:</B> " + req.getRemoteHost() + "<BR>");

out.println("<B>Authorization scheme:</B> " + req.getAuthType() + "<BR>");

}

} //end of class

```

Request object handling

So far, all of our servlet examples have only used the response object, but not the request object. This example shows how to process the data in the request. We assume that this servlet is always called using a POST request, and have therefore chosen to implement the doPost request handling method. doPost method Incidentally, this servlet has been designed to handle the particular type of request from the HTML page that was generated in the previous servlet example. In that HTML page, the user could fill out information in the form and submit it. The action in the HTML form causes the HTMLFormHandler servlet to be invoked, and the doPost request handler method to be called:

```
<FORM METHOD="POST" ACTION="itso.servjsp.servletapi.HTMLFormHandler">
```

In the doPost method, we handle the HttpServletResponse in the same way as before, except that this time, we are also handling the HttpServletRequest. Getting form values

We use the getParameter method of the request to extract the values of the request parameters (name/value fields passed in from the HTML page). We extract parameters named firstname and title from the request:

```

req.getParameter("firstname")

req.getParameter("title")

```

These are two of the input fields that were passed from the HTML form. The getParameter method requires as an argument the name of the parameter that we want to extract (so it must be known), and returns the value of that parameter, or null. To get a list of the all parameter names, we could use the getParameterNames method. This method returns an enumeration of all the parameter names in the request, which we could then iterate through to get the individual parameter values. To extract the value

of the tools parameter, however, we must apply a slightly different technique. The tools' parameter is a multi-value input field (in this case, a checkbox). Because there could be more than one value to extract, we

use the `getParameterValues` method, which returns an array of values. General request properties We can pull environment properties and other information about the client

from the `HttpServletRequest` object and echo them to the response. We choose to put all this code in a separate method, `getReqInfo`, for ease of use.



Simple counter servlet

SimpleCounter is another simple servlet, but here we have an instance counter variable that is initialized in the `init` method package `itso.srv.jsp.servletapi`;

```
import java.io.*;
```

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```
public class SimpleCounter extends HttpServlet {
```

```

private int calledCount;

public void init(ServletConfig config) throws ServletException {

    super.init(config);

    calledCount = 0;

}

protected void service(HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException {

    res.setContentType("text/html");

    PrintWriter out = res.getWriter();

    out.println("<HTML><BODY>");

    out.println("<H2>Servlet API Example - SimpleCounter</H2><HR>");

    ++calledCount;

    out.println("<H4>This servlet has been called: " + calledCount +

        " times.</H4>");

    out.println("</BODY><HTML>");

    out.close();

}

}

```

this global variable. This introduces some issues that you have to consider when designing your servlets.

Multi-Threaded

Because our requests to this servlet are handled in threads against the same servlet object, we must implement mechanisms to guarantee thread safety for these shared instance variables, because we can update them in separate threads. In other words, there is no guarantee that the line that increments the

counter and the line that prints out the counter will be executed asynchronously within a thread. So, we must identify critical sections of code, and synchronize these sections if appropriate. There are many books that deal with concurrent programming issues, therefore we do not describe how to do this, but it is an important point to remember when designing your servlets. Please refer to Appendix E,

“Related publications” on page 433 for a list of useful references.

Servlet initialization parameters

The *SimpleInitServlet* servlet shows how to retrieve initialization parameters from the servlet configuration.

ServletConfig object

The *ServletConfig* object is a parameter that can be passed into the *init* method of the servlet. You can also get the *ServletConfig* object from the request object through the *getServletConfig* method, but it is most commonly used in the *init* method to initialize the servlet’s instance variables.

Methods of the *ServletConfig* object allow us to extract the parameter information from this object. This parameter information is in a name/value pairs format, and can be stored in a file in XML format. We do not have to read the file, however, because the methods of the class provide us with some handy helper methods.

What this servlet does

This servlet simply extracts the parameter information from the configuration file, and stores those values in instance variables. It then echoes this information back to the client that invoked the servlet. In a real-life application, these variables would most likely be used to make a connection to the database, and this connection would be stored in a global

instance variable for later use in the *doGet* method.

Servlet configuration file

The statement `mydriver = config.getInitParameter("driver")` extracts the *driver* parameter by name from the configuration file, and stores it in a global instance variable. The parameter information itself is actually stored in XML format, in a file called *SimpleInitServlet.servlet*. This file must be found through the class path. Where this file actually exists depends on your application server implementation.

```
package itso.servjsp.servletapi;
```

```

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class SimpleInitServlet extends HttpServlet {

    protected String mydriver;

    protected String myurl;

    protected String myuserID;

    protected String mypassword;

    public void init(ServletConfig config) throws ServletException {

        super.init(config);

        mydriver = config.getInitParameter("driver");

        myurl = config.getInitParameter("URL");

        myuserID = config.getInitParameter("userID");

        mypassword = config.getInitParameter("password");

    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)

        throws ServletException, IOException {

        res.setContentType("TEXT/HTML");

        PrintWriter out = res.getWriter();

        out.println("<HTML>");

        out.println("<TITLE>Date Display</TITLE>");

        out.println("<BODY>");

```

```

out.println("<H2>Servlet Initialization Parameters (ServletConfig):

</H2><HR>");

out.println("<B>driver: </B>" + mydriver + "</BR>");

out.println("<B>url: </B>" + myurl + "</BR>");

out.println("<B>password: </B>" + mypassword + "</BR>");

out.println("<B>userID: </B>" + myuserID + "</BR>");

out.println("</BODY></HTML>");

out.close();

}

}

```

VisualAge for Java testing, the file can be put into

d:\IBMVJava\ide\project_resources\..yourproject..\itso\servjsp\servletapi.

The XML configuration file used in this example. Here we have specified four parameters, for demonstration purposes only. These could be used to make a connection to a database.

Understanding the configuration file format

Figure 47 shows the XML format of a configuration file. The WebSphere Application Server supports XML configuration files in this format. *General XML configuration file format*

```

<?xml version="1.0"?>

<servlet>

<code>itso.servjsp.servletapi.SimpleInitServlet</code>

<init-parameter value="COM.ibm.db2.jdbc.app.DB2Driver" name="driver"/>

<init-parameter value="itso" name="password"/>

<init-parameter value="jdbc:db2:sample" name="URL"/>

```

```

<init-parameter value="itso" name="userID"/>

</servlet>

<?xml version="1.0"?>

<servlet>

<code>itso.servjsp.servletapi.SimplePageListServlet</code>

<description>Shows how to use PageListServlet class</description>

<init-parameter name="name1" value="value1"/>

<page-list>

<default-page>

<uri>/index.jsp</uri>

</default-page>

<error-page>

<uri>/error.jsp</uri>

</error-page>

<page>

<uri>/itso/OutputA.jsp</uri>

<page-name>pageA</page-name>

</page>

<page>

<uri>itso/OutputB.jsp</uri>

<page-name>pageB</page-name>

</page>

```

</page-list>

</servlet>

Some of the parameters are beyond the scope of this section, however, we describe a few of the more important parameters that you should know. The elements (also known as tags) are:

- *servlet*: The root element. The `XMLServletConfig` class automatically generates this element.
- *code*: The class name of the servlet (without the `.class` extension), even if the servlet is in a JAR file.
- *init-parameter*: The attributes of this element specify a name/value pair to be used as an initialization parameter. A servlet can have multiple initialization parameters, each within its own *init-parameter* element.
- *page-list*: The elements within this tag specify JavaServer Pages that may be called by the servlet.

HTTP request handling utility servlet

We next look at a servlet, *ServletEnvironmentSnoop*. Because the source for this servlet is rather large. This is a good utility servlet that extracts a lot of information from the request, and echoes its contents back to the client in the response. You should spend some time looking through the source code to see what kind of data can be extracted from a request object, and how to manipulate that data. Use this servlet as a future reference. Sample output of this servlet is also included in the appendix.

The *ServletEnvironmentSnoop* servlet demonstrates the handling of the following request data:

- Request information—HTTP specific request information
- Request header— data passed in the header of the request, such as the character and encoding sets
- Request parameters—name/value pairs of parameter data
- Request attribute names—attributes of the class
- Request cookies—an array containing all cookies present in the request
- Servlet configuration—values used for initializing the servlet
- Servlet context attributes—information about the environment where the application server is running
- Session information—session data associated with the request

16.5 Storing information on Clients

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying returning users:

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them using JSP programs.

The Anatomy of a Cookie:

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this:

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
           path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```


As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this:

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

A JSP script will then have access to the cookies through the request method *request.getCookies()* which returns an array of *Cookie* objects.

Servlet Cookies Methods:

Following is the list of useful methods associated with Cookie object which you can use while manipulating cookies in JSP:

S.N.	Method & Description
1	public void setDomain(String pattern) This method sets the domain to which cookie applies, for example tutorialspoint.com.
2	public String getDomain() This method gets the domain to which cookie applies, for example tutorialspoint.com.

3	public void setMaxAge(int expiry) This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	public int getMaxAge() This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
5	public String getName() This method returns the name of the cookie. The name cannot be changed after creation.
6	public void setValue(String newValue) This method sets the value associated with the cookie.
7	public String getValue() This method gets the value associated with the cookie.
8	public void setPath(String uri) This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	public String getPath() This method gets the path to which this cookie applies.
10	public void setSecure(boolean flag) This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
11	public void setComment(String purpose) This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
12	public String getComment() This method returns the comment describing the purpose of this cookie, or null if the cookie has no

```
comment.
```

Setting Cookies with JSP:

Setting cookies with JSP involves three steps:

(1) Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters:

```
[ ] ( ) = , " / ? @ : ;
```

(2) Setting the maximum age: You use `setMaxAge` to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

```
cookie.setMaxAge(60*60*24);
```

(3) Sending the Cookie into the HTTP response headers: You use `response.addCookie` to add cookies in the HTTP response header as follows:

```
response.addCookie(cookie);
```

Example:

Let us modify our [Form Example](#) to set the cookies for first and last name.

```
<%  
    // Create cookies for first and last names.  
    Cookie firstName = new Cookie("first_name",  
                                   request.getParameter("first_name"));  
    Cookie lastName = new Cookie("last_name",  
                                   request.getParameter("last_name"));  
  
    // Set expiry date after 24 Hrs for both the cookies.  
    firstName.setMaxAge(60*60*24);
```

```

lastName.setMaxAge(60*60*24);

// Add both the cookies in the response header.
response.addCookie( firstName );
response.addCookie( lastName );
%>
<html>
<head>
<title>Setting Cookies</title>
</head>
<body>
<center>
<h1>Setting Cookies</h1>
</center>
<ul>
<li><p><b>First Name:</b>
    <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last Name:</b>
    <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>

```

Let us put above code in main.jsp file and use it in the following HTML page:

```

<html>
<body>
<form action="main.jsp" method="GET">
First Name: <input type="text" name="first_name">

```

```
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Keep above HTML content in a file `hello.jsp` and put `hello.jsp` and `main.jsp` in `<Tomcat-installation-directory>/webapps/ROOT` directory. When you would access `http://localhost:8080/hello.jsp`, here is the actual output of the above form.

FirstName:

Last Name:

Try to enter First Name and Last Name and then click submit button. This would display first name and last name on your screen and same time it would set two cookies `firstName` and `lastName` which would be passed back to the server when next time you would press Submit button.

Next section would explain you how you would access these cookies back in your web application.

Reading Cookies with JSP:

To read cookies, you need to create an array of `javax.servlet.http.Cookie` objects by calling the `getCookies()` method of `HttpServletRequest`. Then cycle through the array, and use `getName()` and `getValue()` methods to access each cookie and associated value.

Example:

Let us read cookies which we have set in previous example:

```
<html>
<head>
<title>Reading Cookies</title>
</head>
```

```

<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
    Cookie cookie = null;
    Cookie[] cookies = null;
    // Get an array of Cookies associated with this domain
    cookies = request.getCookies();
    if( cookies != null ){
        out.println("<h2> Found Cookies Name and Value</h2>");
        for (int i = 0; i < cookies.length; i++){
            cookie = cookies[i];
            out.print("Name : " + cookie.getName( ) + ", ");
            out.print("Value: " + cookie.getValue( )+" <br/>");
        }
    }else{
        out.println("<h2>No cookies founds</h2>");
    }
%>
</body>
</html>

```

Now let us put above code in main.jsp file and try to access it. If you would have set first_name cookie as "John" and last_name cookie as "Player" then running *http://localhost:8080/main.jsp* would display the following result:

Found Cookies Name and Value

Name : first_name, Value: John
Name : last_name, Value: Player

Delete Cookies with JSP:

To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps:

- Read an already existing cookie and store it in Cookie object.
- Set cookie age as zero using **setMaxAge()** method to delete an existing cookie.
- Add this cookie back into response header.

Example:

Following example would delete an existing cookie named "first_name" and when you would run main.jsp JSP next time it would return null value for first_name.

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
    Cookie cookie = null;
    Cookie[] cookies = null;
    // Get an array of Cookies associated with this domain
    cookies = request.getCookies();
    if( cookies != null ){
```

```

out.println("<h2> Found Cookies Name and Value</h2>");
for (int i = 0; i < cookies.length; i++){
    cookie = cookies[i];
    if((cookie.getName( )).compareTo("first_name") == 0 ){
        cookie.setMaxAge(0);
        response.addCookie(cookie);
        out.print("Deleted cookie: " +
            cookie.getName( ) + "<br/>");
    }
    out.print("Name : " + cookie.getName( ) + ", ");
    out.print("Value: " + cookie.getValue( )+" <br/>");
}
}else{
    out.println(
        "<h2>No cookies founds</h2>");
}
%>
</body>
</html>

```

Now let us put above code in main.jsp file and try to access it. It would display the following result:

```

Cookies Name and Value
Deleted cookie : first_name
Name : first_name, Value: John
Name : last_name, Value: Player

```

Now try to run *http://localhost:8080/main.jsp* once again and it should display only one cookie as follows:

Found Cookies Name and Value

Name : last_name, Value: Player

You can delete your cookies in Internet Explorer manually. Start at the Tools menu and select Internet Options. To delete all cookies, press Delete Cookies.

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Still there are following three ways to maintain session between web client and web server:

Cookies:

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

Hidden Form Fields:

A web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type="hidden" name="sessionid" value="12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.

This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting:

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.

For example, with `http://tutorialspoint.com/file.htm;sessionid=12345`, the session identifier is attached as `sessionid=12345` which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.

The session Object:

Apart from the above mentioned three ways, JSP makes use of servlet provided `HttpSession` Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

By default, JSPs have session tracking enabled and a new `HttpSession` object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:

```
<%@ page session="false" %>
```

The JSP engine exposes the `HttpSession` object to the JSP author through the implicit **session** object. Since **session** object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or `getSession()`.

Here is a summary of important methods available through session object:

S.N.	Method & Description
1	public Object getAttribute(String name) This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	public Enumeration getAttributeNames() This method returns an Enumeration of String objects containing the names of

	all the objects bound to this session.
3	public long getCreationTime() This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	public String getId() This method returns a string containing the unique identifier assigned to this session.
5	public long getLastAccessedTime() This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
6	public int getMaxInactiveInterval() This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	public void invalidate() This method invalidates this session and unbinds any objects bound to it.
8	public boolean isNew() This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	public void removeAttribute(String name) This method removes the object bound with the specified name from this session.
10	public void setAttribute(String name, Object value) This method binds an object to this session, using the name specified.
11	public void setMaxInactiveInterval(int interval) This method specifies the time, in seconds, between client requests before the

servlet container will invalidate this session.

Session Tracking Example:

This example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
<%@ page import="java.io.*,java.util.*" %>

<%

    // Get session creation time.

    Date createTime = new Date(session.getCreationTime());

    // Get last access time of this web page.

    Date lastAccessTime = new Date(session.getLastAccessedTime());


    String title = "Welcome Back to my website";

    Integer visitCount = new Integer(0);

    String visitCountKey = new String("visitCount");

    String userIDKey = new String("userID");

    String userID = new String("ABCD");


    // Check if this is new comer on your web page.

    if (session.isNew()){

        title = "Welcome to my website";

        session.setAttribute(userIDKey, userID);

        session.setAttribute(visitCountKey, visitCount);
```

```

}

visitCount = (Integer)session.getAttribute(visitCountKey);

visitCount = visitCount + 1;

userID = (String)session.getAttribute(userIDKey);

session.setAttribute(visitCountKey, visitCount);

%>

<html>

<head>

<title>Session Tracking</title>

</head>

<body>

<center>

<h1>Session Tracking</h1>

</center>

<table border="1" align="center">

<tr bgcolor="#949494">

<th>Session info</th>

<th>Value</th>

</tr>

<tr>

<td>id</td>

<td><% out.print( session.getId()); %></td>

```

```

</tr>

<tr>

  <td>Creation Time</td>

  <td><% out.print(createTime); %></td>

</tr>

<tr>

  <td>Time of Last Access</td>

  <td><% out.print(lastAccessTime); %></td>

</tr>

<tr>

  <td>User ID</td>

  <td><% out.print(userID); %></td>

</tr>

<tr>

  <td>Number of visits</td>

  <td><% out.print(visitCount); %></td>

</tr>

</table>

</body>

</html>

```

Now put above code in main.jsp and try to access *http://localhost:8080/main.jsp*. It would display the following result when you would run for the first time:

Welcome to my website

Session Infomation

Session info	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

Now try to run the same JSP for second time, it would display following result.

Welcome Back to my website

Session Infomation

info type	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

Deleting Session Data:

When you are done with a user's session data, you have several options:

- **Remove a particular attribute:** You can call *public void removeAttribute(String name)* method to delete the value associated with a particular key.
- **Delete the whole session:** You can call *public void invalidate()* method to discard an entire session.
- **Setting Session timeout:** You can call *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.
- **Log the user out:** The servers that support servlets 2.4, you can call **logout** to log the client out of the Web server and invalidate all sessions belonging to all the users.
- **web.xml Configuration:** If you are using Tomcat, apart from the above mentioned methods, you can configure session time out in web.xml file as follows.

```
<session-config>

    <session-timeout>15</session-timeout>

</session-config>
```

The timeout is expressed as minutes, and overrides the default timeout which is 30 minutes in Tomcat.

The `getMaxInactiveInterval()` method in a servlet returns the timeout period for that session in seconds. So if your session is configured in web.xml for 15 minutes, `getMaxInactiveInterval()` returns 900.

16.6 Summary

In this unit we introduce you to Java servlet concepts.

We provide an overview of the Java Servlet API, and discuss the servlet runtime environment and life-cycle. Servlet examples are provided which demonstrate basic to advanced servlet functionality. Finally, we discuss some common servlet interaction techniques, such as servlet filtering and chaining. If you are already familiar with Java servlets, we suggest you still skim through this chapter. We present some concepts here that are built on in subsequent chapters. This will familiarize you with the naming conventions used, and provide some continuity in the reading.

Servlets are protocol and platform independent server-side software components, written in Java. They run inside a Java enabled server or application server, such as the WebSphere Application Server. Servlets are loaded and executed within the Java Virtual Machine (JVM) of the Web

server or application server, in much the same way that applets are loaded and executed within the JVM of the Web client. Since servlets run inside the servers, however, they do not need a graphical user interface (GUI). In this sense, servlets are also faceless objects.

Servlets more closely resemble Common Gateway Interface (CGI) scripts or programs than applets in terms of functionality. As in CGI programs, Servlets can respond to user events from an HTML request, and then dynamically construct an HTML response that is sent back to the client.

16.7 Keyword

Servlet Builder, ServletConfig, ServletContext, ServletEngine, ServletOutputStream, ServletRequest, ServletResponse

16.8 Exercise

- Write a simple JSP page that prints out a series of consecutive numbers by using a Java loop construction. The output could be for example:
 - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- Modify your first JSP page (1a) such that it prints out the series of numbers given below using a Java loop: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
- Modify the JSP page further such that it prints in each line the sum of the squared numbers iterated through so far, as shown below:
 - $1^2 = 1$
 - $1^2 + 2^2 = 1 + 4 = 5$
 - $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$... and so on
- In order to make the list printed out by the previous JSP page easier to read, change the font color every other line.
- Write a JSP page, which presents 75 random integer numbers from $\{0, 1, \dots, 10\}$ in a table with 5 columns and 28 rows as follows: the random numbers are listed in the first

threecolumns, with the sum and the average of the random numbers of a particular row given in the last two columns. Headers for the five columns (“Trial 1”, “Trial 2”, “Trial 3”, “Horizontal Sum”, “Horizontal Average”) are printed in the upper row. Finally, the sum and average of the three columns are printed in the last two rows and the according headers (“Vertical Sum”, and “Vertical Average”) are given in fourth column. Try to make the table easy to read by, for example, limiting the number of displayed digits, using bold fonts, choosing appropriate font and background colors.

- Implement above exercise as a Servlet.
- Modify the Servlet created in exercise above so that the random numbers only change on anew session HINT: on a new session, initialize Random() with the current time and save time; on an existing session, initialize Random() with time saved previously in the session scope.
- Responding to a User Reques : Write a JSP application comprising of two parts. The page “CollectUserInput.jsp” provides three fields which enable the user to specify input for a random number experiment: the number of trials Ntrial, the minimum random number Nmin, and the maximum random number Nmax. By clicking the “Submit” button, the “EvaluateRandomNumberExperiment.jsp” page is requested. This JSP presents these numbers, performs an experiment where NTrial random numbers between Nmin and Nmax are drawn and lists the relative frequencies (how often this number was drawn in percent) for all the numbers between Nmin and Nmax.

16.8 References

- M. Deitel, P.J. Deitel, A.B. Goldberg: Internet & World Wide Web How to program, 3rd Edition, Pearson Education/ PHI, 2004.
- Robert W. Sebesta: Programming the World Wide Web, 4th Edition, Pearson Education, 2008.