Module – 1
Introduction to HTML

1.1 A Brief Introduction to the Internet,
1.2 WWW,
1.3 Web Browsers and Web Servers,
1.4 URLs,
1.5 MIME,
1.6 HTTP,
1.7 Security,
1.8 The Web Programmers Toolbox.
1.9 XHTML: Basic syntax,
1.10 Standard structure,
1.11 Basic text markup,
1.12 Images,
1.13 Hypertext Links.

1.1 A BRIEF INTRODUCTION ABOUT THE INTERNET

1.1.1 Origins:

- **1960s**
  - U.S. Department of Defence (DoD) became interested in developing a new large-scale computer network
  - The purposes of this network were communications, program sharing, and remote computer access for researchers working on defence-related contracts.
  - The DoD's Advanced Research Projects Agency (ARPA) funded the construction of the first such network. Hence it was named as ARPAnet.
  - The primary early use of ARPAnet was simple text-based communications through e-mail.
- **late 1970s and early 1980s**
  - BITNET, which is an acronym for ***B**ecause **I**t's **T**ime **NET**work*, began at the City University of New York. It was built initially to provide electronic mail and file transfers.
  - CSNET is an acronym for ***C**omputer **S**cience **NET**work*. Its initial purpose was to provide electronic mail.
- **1990s**
  - NSFnet which was created in 1986 replaced ARPAnet by 1990.
  - It was sponsored by the National Science Foundation (NSF).
  - By 1992 NSFnet, connected more than 1 million computers around the world.

- In 1995, a small part of NSFnet returned to being a research network. The rest became known as the *Internet.*

1.1.2 What the Internet is:

- The Internet is a huge collection of computers connected in a communications network.
- The Transmission Control Protocol/Internet Protocol (TCP/IP) became the standard for computer network connections in 1982.
- Rather than connecting every computer on the Internet directly to every other computer on the Internet, normally the individual computers in an organization are connected to each other in a local network. One node on this local network is physically connected to the Internet.
- So, the Internet is actually a *network of networks*, rather than a network of computers.
- Obviously, all devices connected to the Internet must be uniquely identifiable.

1.1.3 Internet Protocols (IP) Addresses

- The Internet Protocol (IP) address of a machine connected to the Internet is a unique 32-bit number.
- IP addresses usually are written (and thought of) as four 8-bit numbers, separated by periods.
- The four parts are separately used by Internet-routing computers to decide where a message must go next to get to its destination.
- Although people nearly always type domain names into their browsers, the IP works just as well.
- For example, the IP for United Airlines (www.ual.com) is 209.87.113.93. So, if a browser is pointed at http://209.87.113.93, it will be connected to the United Airlines Web site.

1.1.4 Domain names

The IP addresses are numbers. Hence, it would be difficult for the users to remember IP address. To solve this problem, text based names were introduced. These are technically known as ***domain name system (DNS).***

These names begin with the names of the host machine, followed by progressively larger enclosing collection of machines, called ***domains.*** There may be two, three or more domain names. DNS is of the form **hostname.domainName.domainName** . Example:**atme.ac.in** The steps for conversion from DNS to IP:

- The DNS has to be converted to IP address before destination is reached.
- This conversion is needed because computer understands only numbers.
- The conversion is done with the help of *name server.*
- As soon as domain name is provided, it will be sent across the internet to contact name servers.
- This name server is responsible for converting domain name to IP
- If one of the *name servers* is not able to convert DNS to IP, it contacts other name server.
- This process continues until IP address is generated.

- Once the IP address is generated, the host can be accessed.
- The hostname and all domain names form what is known as FULLY QUALIFIED DOMAIN NAME.

This is as shown below:

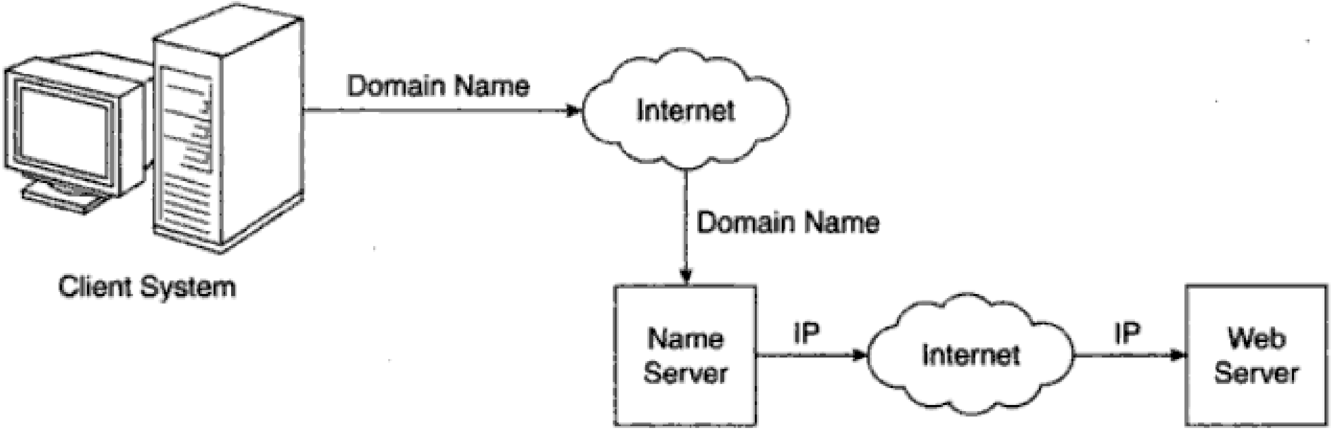


Figure 1.1 Domain name conversion

1.2 The World-Wide Web

1.2.1 Origins

- Tim Berners Lee and his group proposed a new protocol for the Internet whose intention was to allow scientists around the world to use the Internet to exchange documents describing their work.
- The proposed new system was designed to allow a user anywhere on the Internet to search for and retrieve documents from the databases on any number of different document-serving computers.
- The system used *hypertext*, which is text with embedded links to text in other documents to allow non-sequential browsing of textual material.
- The units of web are referred as pages, documents and resources.
- Web is merely a vast collection of documents, some of which are connected by links.
- These documents can be accessed by web browsers and are provided by web servers.

1.2.2 Web or Internet?

It is important to understand that the Internet and the Web is not the same thing.

- The **Internet** is a collection of computers and other devices connected by equipment that allows them to communicate with each other.
- The **Web** is a collection of software and protocols that has been installed on most, if not all, of the computers on the Internet.

1.3 Web Browsers

- Documents provided by servers on the Web are requested by **browsers**, which are programs running on client machines.

- They are called browsers because they allow the user to browse the resources available on servers.
- Mosaic was the first browser with a graphical user interface.
- A browser is a client on the Web because it initiates the communication with a server, which waits for a request from the client before doing anything.
- In the simplest case, a browser requests a static document from a server.
- The server locates the document among its servable documents and sends it to the browser, which displays it for the user.
- Sometimes a browser directly requests the execution of a program stored on the server. The output of the program is then returned to the browser.
- Examples: Internet Explorer, Mozilla Firefox, Netscape Navigator, Google Chrome, Opera etc.,

1.4 Web Servers

Web servers are programs that provide documents to requesting browsers. Example: Apache

1.4.1 Web Server Operation:

- All the communications between a web client and a web server use the HTTP
- When a web server begins execution, it informs the OS under which it is running & it runs as a background process
- A web client or browser, opens a network connection to a web server, sends information requests and possibly data to the server, receives information from the server and closes the connection.
- The primary task of web server is to monitor a communication port on host machine, accept HTTP commands through that port and perform the operations specified by the commands.
- When the URL is received, it is translated into either a filename or a program name.

1.4.2 General Server Characteristics:

- The file structure of a web server has two separate directories
- The root of one of these is called **document root** which stores web documents
- The root of the other directory is called the **server root** which stores server and its support softwares
- The files stored directly in the document root are those available to clients through top level URLs
- The secondary areas from which documents can be served are called **virtual document trees.**
- Many servers can support more than one site on a computer, potentially reducing the cost of each site and making their maintenance more convenient. Such secondary hosts are called **virtual hosts.**
- Some servers can serve documents that are in the document root of other machines on the web; in this case they are called as **proxy servers**

1.4.3 Apache

- Apache is the most widely used Web server.
- The primary reasons are as follows: Apache is an excellent server because it is both fast and reliable.

- Furthermore, it is open-source software, which means that it is free and is managed by a large team of volunteers, a process that efficiently and effectively maintains the system.
- Finally, it is one of the best available servers for Unix-based systems, which are the most popular for Web servers.
- Apache is capable of providing a long list of services beyond the basic process of serving documents to clients.
- When Apache begins execution, it reads its configuration information from a file and sets its parameters to operate accordingly.

1.4.4 IIS

- Microsoft IIS server is supplied as part of Windows—and because it is a reasonably good server—most Windows-based Web servers use IIS.
- With IIS, server behaviour is modified by changes made through a window-based management program, named the IIS snap-in, which controls both IIS and ftp.
- This program allows the site manager to set parameters for the server.
- Under Windows XP and Vista, the IIS snap-in is accessed by going to *Control Panel, Administrative Tools, and IIS Admin.*

1.5 Uniform Resource Locators

- Uniform Resource Locators (URLs) are used to identify different kinds of resources on Internet.
- If the web browser wants some document from web server, just giving domain name is not sufficient because domain name can only be used for locating the server.
- It does not have information about which document client needs. Therefore, URL should be provided.
- The general format of URL is: **scheme: object-address**
- Example: **http://www.vtu.ac.in/results.php**
- The scheme indicates protocols being used. (http, ftp, telnet...)
- In case of http, the full form of the object address of a URL is as follows:
- **//fully-qualified-domain-name/path-to-document**
- URLs can never have embedded spaces
- It cannot use special characters like semicolons, ampersands and colons
- The path to the document for http protocol is a sequence of directory names and a filename, all separated by whatever special character the OS uses. (forward or backward slashes)
- The path in a URL can differ from a path to a file because a URL need not include all directories on the path
- A path that includes all directories along the way is called a **complete path.**

- Example: http://www.atme.in/
- In most cases, the path to the document is relative to some base path that is specified in the configuration files of the server. Such paths are called **partial paths.**
- Example: http://www.atme.in/

1.6 Multipurpose Internet Mail Extensions

- MIME stands for Multipurpose Internet Mail Extension.
- The server system apart from sending the requested document, it will also send MIME information.
- The MIME information is used by web browser for rendering the document properly.
- The format of MIME is: type/subtype
- Example: text/html , text/doc , image/jpeg , video/mpeg
- When the type is either text or image, the browser renders the document without any problem
- However, if the type is video or audio, it cannot render the document
- It has to take the help of other software like media player, win amp etc.,
- These software's are called as helper applications or plugins
- These non-textual information are known as HYPER MEDIA
- Experimental document types are used when user wants to create a customized information & make it available in the internet
- The format of experimental document type is: type/x-subtype
- Example: database/x-xbase , video/x-msvideo
- Along with creating customized information, the user should also create helper applications.
- This helper application will be used for rendering the document by browser.
- The list of MIME specifications is stored in configuration file of web server.

1.7 The Hyper Text Transfer Protocol

1.7.1 The Request Phase

The general form of an HTTP request is as follows:

1. HTTP method Domain part of the URL HTTP version
2. Header fields
3. Blank line
4. Message body

The following is an example of the first line of an HTTP request: **GET /storefront.html HTTP/1.1**

Table 1.1 HTTP request methods

Method	Description
GET	Returns the contents of the specified document
HEAD	Returns the header information for the specified document
POST	Executes the specified document, using the enclosed data
PUT	Replaces the specified document with the enclosed data
DELETE	Deletes the specified document

The format of a header field is the field name followed by a colon and the value of the field. There are four categories of header fields:

1. ***General***: For general information, such as the date
2. ***Request***: Included in request headers
3. ***Response***: For response headers
4. ***Entity***: Used in both request and response headers

A wildcard character, the asterisk (*), can be used to specify that part of a MIME type can be anything.

Accept: text/plain

Accept: text/html → *Can be written as* → Accept: text/*

The Host: *host name* request field gives the name of the host. The Host field is required for HTTP 1.1. The If-Modified-Since: *date* request field specifies that the requested file should be sent only if it has been modified since the given date. If the request has a body, the length of that body must be given with a Content-length field. The header of a request must be followed by a blank line, which is used to separate the header from the body of the request.

1.7.2 The Response Phase:

The general form of an HTTP response is as follows:

1. Status line
2. Response header fields
3. Blank line
4. Response body

The status line includes the HTTP version used, a three-digit status code for the response, and a short textual explanation of the status code. For example, most responses begin with the following:

HTTP/1.1 200 OK

The status codes begin with 1, 2, 3, 4, or 5. The general meanings of the five categories specified by these first digits are shown in Table 1.2.

Table 1.2 First digits of HTTP status codes

First Digit	Category
1	Informational
2	Success
3	Redirection
4	Client error
5	Server error

One of the more common status codes is one user never want to see: 404 Not Found, which means the requested file could not be found.

1.8 Security

Security is one of the major concerns in the Internet. The server system can be accessed easily with basic hardware support, internet connection & web browser. The client can retrieve very important information from the server. Similarly, the server system can introduce virus on the client system. These viruses can destroy the hardware and software in client. While programming the web, following requirements should be considered:

- **Privacy:** it means message should be readable only to communicating parties and not to intruder.
- **Integrity:** it means message should not be modified during transmission.
- **Authentication:** it means communicating parties must be able to know each other's identity
- **Non-repudiation:** it means that it should be possible to prove that message was sent and received properly

Security can be provided using cryptographic algorithm. Ex: private key, public key Protection against viruses and worms is provided by antivirus software, which must be updated frequently so that it can detect and protect against the continuous stream of new viruses and worms.

1.9 The Web Programmer's Toolbox

- Web programmers use several languages to create the documents that servers can provide to browsers.
- The most basic of these is **XHTML**, the standard mark-up language for describing how Web documents should be presented by browsers. Tools that can be used without specific knowledge of XHTML are available to create XHTML documents.
- A **plug-in** is a program that can be integrated with a word processor to make it possible to use the word processor to create XHTML. A **filter** converts a document written in some other format to XHTML.

- **XML** is a meta-mark-up language that provides a standard way to define new mark-up languages.
- **JavaScript** is a client-side scripting language that can be embedded in XHTML to describe simple computations. JavaScript code is interpreted by the browser on the client machine; it provides access to the elements of an XHTML document, as well as the ability to change those elements dynamically.
- **Flash** is a framework for building animation into XHTML documents. A browser must have a Flash player plug-in to be able to display the movies created with the Flash framework.
- **Ajax** is an approach to building Web applications in which partial document requests are handled asynchronously. Ajax can significantly increase the speed of user interactions, so it is most useful for building systems that have frequent interactions.
- **PHP** is the server-side equivalent of JavaScript. It is an interpreted language whose code is embedded in XHTML documents. PHP is used primarily for form processing and database access from browsers.
- **Servlets** are server-side Java programs that are used for form processing, database access, or building dynamic documents. JSP documents, which are translated into servlets, are an alternative approach to building these applications. JSF is a development framework for specifying forms and their processing in JSP documents.
- **ASP.NET** is a Web development framework. The code used in ASP.NET documents, which is executed on the server, can be written in any .NET programming language.
- **Ruby** is a relatively recent object-oriented scripting language that is introduced here primarily because of its use in Rails, a Web applications framework.
- **Rails** provides a significant part of the code required to build Web applications that access databases, allowing the developer to spend his or her time on the specifics of the application without the drudgery of dealing with all of the housekeeping details.

1.10 Origins and Evolution of HTML and XHTML

HTML → Hyper Text Mark-up Language

XHTML → eXtensible Hyper Text Mark-up Language

1.10.1 HTML versus XHTML

HTML	XHTML
HTML is much easier to write	XHTML requires a level of discipline many of

	us naturally resist
huge number of HTML documents available on the Web, browsers will continue to support HTML as far as one can see into the future.	some older browsers have problems with some parts of XHTML.
HTML has few syntactic rules, and HTML processors (e.g., browsers) do not enforce the rules it does have. Therefore, HTML authors have a high degree of freedom to use their own syntactic preferences to create documents. Because of this freedom, HTML documents lack consistency, both in low-level syntax and in overall structure.	XHTML has strict syntactic rules that impose a consistent structure on all XHTML documents. Another significant reason for using XHTML is that when you create an XHTML document, its syntactic correctness can be checked, either by an XML browser or by a validation tool
Used for displaying the data	Used for describing the data

1.10.2 Basic Syntax

- The fundamental syntactic units of HTML are called ***tags***.
- In general, tags are used to specify categories of content.
- The syntax of a tag is the tag's name surrounded by *angle brackets* (< and >).
- Tag names must be written in all lowercase letters.
- Most tags appear in pairs: an *opening tag and a closing tag*.
- The name of a closing tag is the name of its corresponding opening tag with a slash attached to the beginning. For example, if the tag's name is p, the corresponding closing tag is named /p.
- Whatever appears between a tag and its closing tag is the ***content*** of the tag. Not all tags can have content.
- The opening tag and its closing tag together specify a container for the content they enclose.
- The container and its content together are called an ***element***.
- Example: **<p> This is ATME Web Programming Notes. </p>**
- The paragraph tag, <p>, marks the beginning of the content; the </p> tag marks the end of the content of the paragraph element.
- Attributes, which are used to specify alternative meanings of a tag, can appear between an opening tag's name and its right angle bracket.
- They are specified in keyword form, which means that the attribute's name is followed by an equal's sign and the attribute's value.
- Attribute names, like tag names, are written in lowercase letters.
- Attribute values must be delimited by double quotes.

- Comments in programs increase the readability of those programs. Comments in XHTML have the same purpose. They can appear in XHTML in the following form:
- **<!- - anything except two adjacent dashes - ->**
- Browsers ignore XHTML comments—they are for people only. Comments can be spread over as many lines as are needed. For example, you could have the following comment:
- **<!- - CopyRights.html**
  This notes is prepared by Kswamy of Computer Science Department ATME, Mysore - ->

1.10.3 Standard XHTML Document Structure

- Every XHTML document must begin with an xml declaration element that simply identifies the document as being one based on XML. This element includes an attribute that specifies the version number 1.0.
- The xml declaration usually includes a second attribute, encoding, which specifies the encoding used for the document [utf-8].
- Following is the xml declaration element, which should be the first line of every XHTML document:

<?xml version = "1.0" encoding = "utf-8"?>

- Note that this declaration must begin in the first character position of the document file.
- The xml declaration element is followed immediately by an SGML DOCTYPE command, which specifies the particular SGML document-type definition (DTD) with which the document complies, among other things.
- The following command states that the document in which it is included complies with the XHTML 1.0 Strict standard:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml1-strict.dtd">

- An XHTML document must include the four tags <html>, <head>, <title>, and <body>.
- The <html> tag identifies the root element of the document. So, XHTML documents always have an <html> tag immediately following the DOCTYPE command, and they always end with the closing html tag, </html>.
- The html element includes an attribute, xmlns, that specifies the XHTML namespace, as shown in the following element:

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

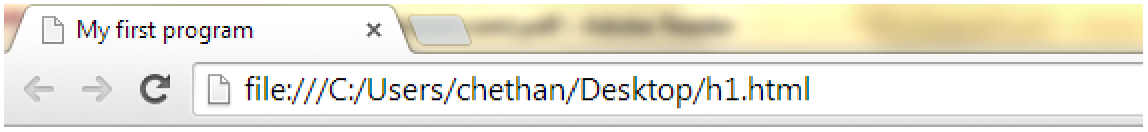
- Although the xmlns attribute's value looks like a URL, it does not specify a document. It is just a name that happens to have the form of a URL.
- An XHTML document consists of two parts, named the *head* and the *body*.
- The <head> element contains the head part of the document, which provides information about the document and does not provide the content of the document.

- The body of a document provides the content of the document.
- The content of the title element is displayed by the browser at the top of its display window, usually in the browser window's title bar.

1.11 Basic Text Markup

We will have a look at a complete XHTML document:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml1-strict.dtd">
<!- - complete.html
A document which must be followed throughout the notes - ->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>
My first program
</title>
</head>
<body>
<p>
My Dear ATME Friends, All The Best..!! Have a Happy Reading of my notes..!!
</p>
</body>
</html>
```



My Dear ATME Friends, All The Best..!! Have a Happy Reading of my notes..!!

PLEASE NOTE: From here onwards programming in XHTML will begin. Please add the following compulsory document structure to all programs in the first 4 lines and skip the simple <html> tag of first line because I have begun the coding part directly .

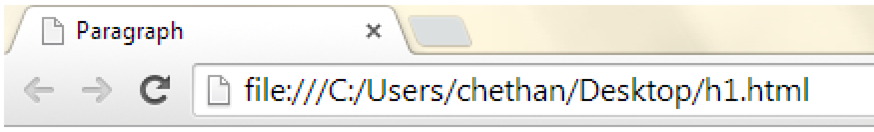
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml1-strict.dtd">

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

1.11.1 Paragraphs:

It begins with <p> and ends with </p>. Multiple paragraphs may appear in a single document.

```
<html>
 <head>
        <title> Paragraph </title>
</head>
<body>
         <p> Paragraph 1 </p>
        <p> Paragraph 2 </p>
        <p> Paragraph 3 </p>
</body>
 </html>
```



Paragraph 1

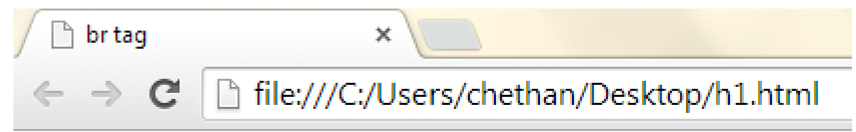
Paragraph 2

Paragraph 3

1.11.2 Line Breaks:

The break tag is specified as <br />. The slash indicates that the tag is both an opening and closing tag.

```
<html>
<head>
         <title> br tag </title>
</head>
<body>
        <p> My Name is Chethan <br/>
        I am from CSE Department <br/>
        ATME, Mysore </p>
 </body>
 </html>
```

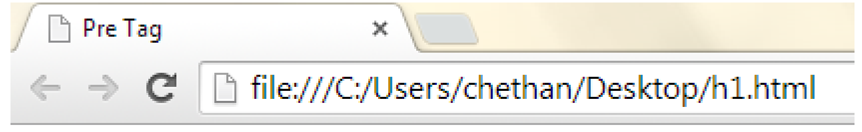


My Name is Chethan
I am from CSE Department
ATME, Mysore

1.11.3 Preserving White Space:

Sometimes it is desirable to preserve the white space in text—that is, to prevent the browser from eliminating multiple spaces and ignoring embedded line breaks. This can be specified with the **<pre>** tag.

```
<html>
 <head>
        <title> Pre Tag </title>
 </head>
<body>
        <p><pre> My Name is Chethan
                        I am from CSE Department
                                ATME, Mysore </pre></p>
</body>
</html>
```




```
My Name is Chethan
                I am from CSE Department
                                ATME, Mysore
```

1.11.4 Headings:

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

```
<html>
```

```
<head>
        <title> Headings </title>
</head>
<body>
        <h1> Heading 1 </h1>
        <h2> Heading 2 </h2>
        <h3> Heading 3 </h3>
        <h4> Heading 4 </h4>
        <h5> Heading 5 </h5>
        <h6> Heading 6 </h6>
</body>
</html>
```

Headings

file:///C:/Users/chethan/Desktop/h1.html

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

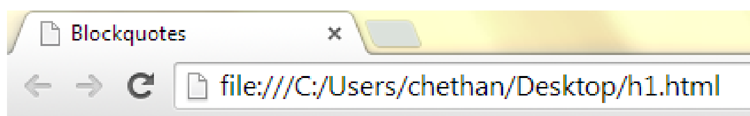
Heading 6

1.11.5 Block Quotations:

The <blockquote> tag is used to make the contents look different from the surrounding text.

```
<html>
<head>
        <title> Blockquotes </title>
</head>
<body>
                <p> Swami Vivekananda says </p>
        <blockquote>
                <p> "Arise..!! Awake..!!" </p>
        </blockquote>
```

```
            <p> He is my Role model </p>
</body>
</html>
```



Swami Vivekananda says

"Arise..!! Awake..!!"

He is my Role model

1.11.6 Font Styles and Sizes:

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

```
<html>
<head>
 <title> font styles and sizes </title>
</head>
<body>
       <p>
             <pre> Illustration of Font Styles
                    <b> This is Bold </b>
                    <i> This is Italics </i>
                    <u> This is Underline </u>
                    <em> This is Emphasis </em>
                    <strong> This is strong </strong>
                    <code> Total = Internals + Externals //this is code</code>
             </pre>
       </p>
       <p>
             <pre> Illustration of Font Sizes (subscripts and superscripts)
                    x<sub>2</sub><sup>3</sup> + y<sub>1</sub><sup>2</sup>
```

```
        </pre>
    </p>
</body>
</html>
```

font styles and sizes

file:///C:/Users/chethan/Desktop/h1.html

Illustration of Font Styles
This is Bold
This is Italics
This is Underline
This is Emphasis
This is strong
Total = Internals + Externals //this is code

Illustration of Font Sizes (subscripts and superscripts)
$x_2^3 + y_1^2$

1.11.7 Character Entities:

- XHTML provides a collection of special characters that are sometimes needed in a document but cannot be typed as themselves.
- In some cases, these characters are used in XHTML in some special way—for example, >, <, and &. In other cases, the characters do not appear on keyboards, such as the small raised circle that represents “degrees” in a reference to temperature.
- These special characters are defined as entities, which are codes for the characters. An entity in a document is replaced by its associated character by the browser.

```
<html>
 <head>
        <title> Character Entities </title>
 </head>
<body>
    <p>
        <pre> Illustration of character entities
            if you get &gt 70%, then you will get FCD
            if you get &lt 35%, then you are Fail
            &frac12 of my classmates get very good marks
            Now, the temperature in Bangalore is 30&deg C
        </pre>
```

```
        </p>
 </body>
</html>
```

Character Entities

file:///C:/Users/chethan/Desktop/h1.html

```
 Illustration of character entities
if you get > 70%, then you will get FCD
if you get < 35%, then you are Fail
½ of my classmates get very good marks
Now, the temperature in Bangalore is 30° C
```

1.11.8 Horizontal Rules:

- The parts of a document can be separated from each other, making the document easier to read, by placing horizontal lines between them. Such lines are called horizontal rules.
- The block tag that creates them is <hr />. The <hr /> tag causes a line break (ending the current line) and places a line across the screen. Note again the slash in the <hr /> tag, indicating that this tag has no content and no closing tag.

```
<html>
 <head>
        <title> Horizontal Rule </title>
 </head>
 <body>
        <p>
               The ATME Trust  was founded in the year 2007 <hr/>
               It was founded by our Chairman Mr. L Arun Kumar <hr/>
               Mr.K. Shiva Shankar  is our Member <hr/>
        </p>
</body>
</html>
```

Horizontal Rule

file:///C:/Users/chethan/Desktop/h1.html

The ATME Trust was founded in the year 2007

It was founded by our Chairman Mr. L Arun Kumar

Mr.K. Shiva Shankar is our Member

1.11.9 The meta Element:

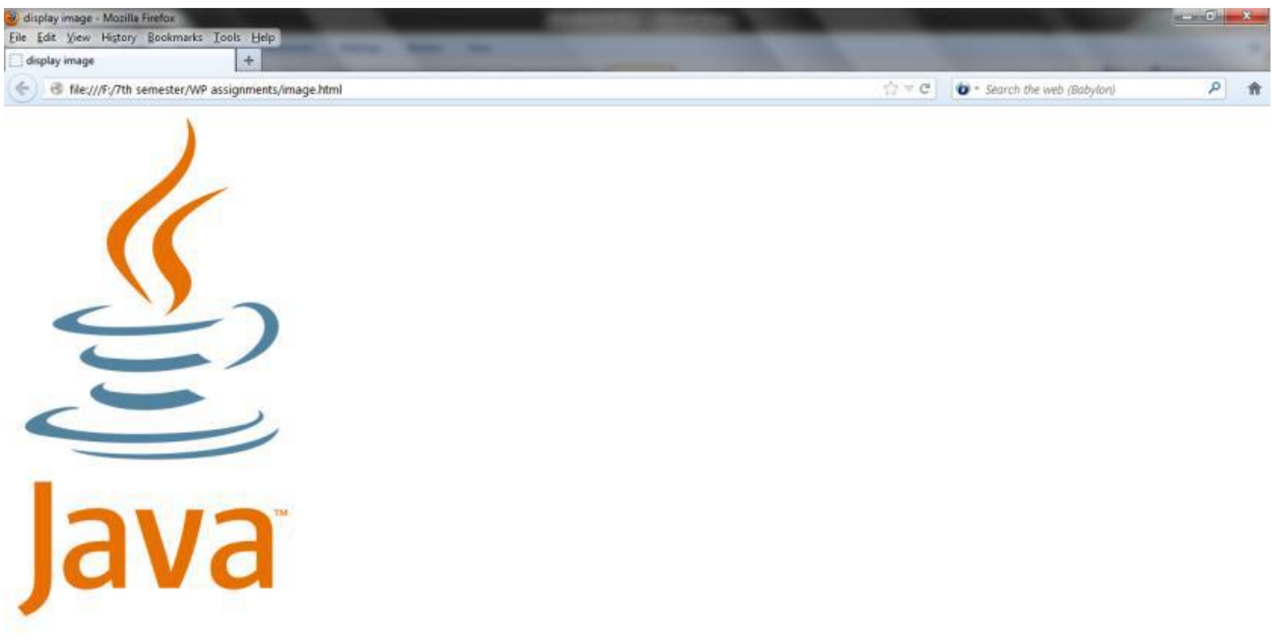
- The meta element is used to provide additional information about a document. The meta tag has no content; rather, all of the information provided is specified with attributes.
- The two attributes that are used to provide information are name and content. The user makes up a name as the value of the name attribute and specifies information through the content attribute.
- One commonly chosen name is keywords; the value of the content attribute associated with the keywords are those which the author of a document believes characterizes his or her document.
- An example is
    - **<meta name = "Title" content = "Programming the Web" />**
    - **<meta name = "Author" content = "Divya K" />**
- Web search engines use the information provided with the meta element to categorize Web documents in their indices.

1.12 IMAGES

- Image can be displayed on the web page using <img> tag.
- When the <img> tag is used, it should also be mentioned which image needs to be displayed. This is done using src attribute.
- Attribute means extra information given to the browser
- Whenever <img> tag is used, alt attribute is also used.
- Alt stands for alert.
- Some very old browsers would not be having the capacity to display the images.
- In this case, whatever is the message given to alt attribute, that would be displayed.
- Another use of alt is → when image display option has been disabled by user. The option is normally disabled when the size of the image is huge and takes time for downloading.

```
<html>
<head>
        <title>display image</title>
```

```
</head>
        <body>
                <img src="java.png" alt="cannot display"/>
        </body>
</html>
```



NOTE:

JPEG → Joint Photographic Experts Group

GIF → Graphic Interchange Format

PNG → Portable Network Graphics

XHTML Document Validation:

The W3C provides a convenient Web-based way to validate XHTML documents against its standards.

Step 1: The URL of the service is http://validator.w3.org/file-upload.html. Copy & paste this link.

Step 2: You will be driven to "Validate by File Upload" option automatically.

Step 3: Browse for a XHTML program file in your computer. (example: *F:/complete.html*)

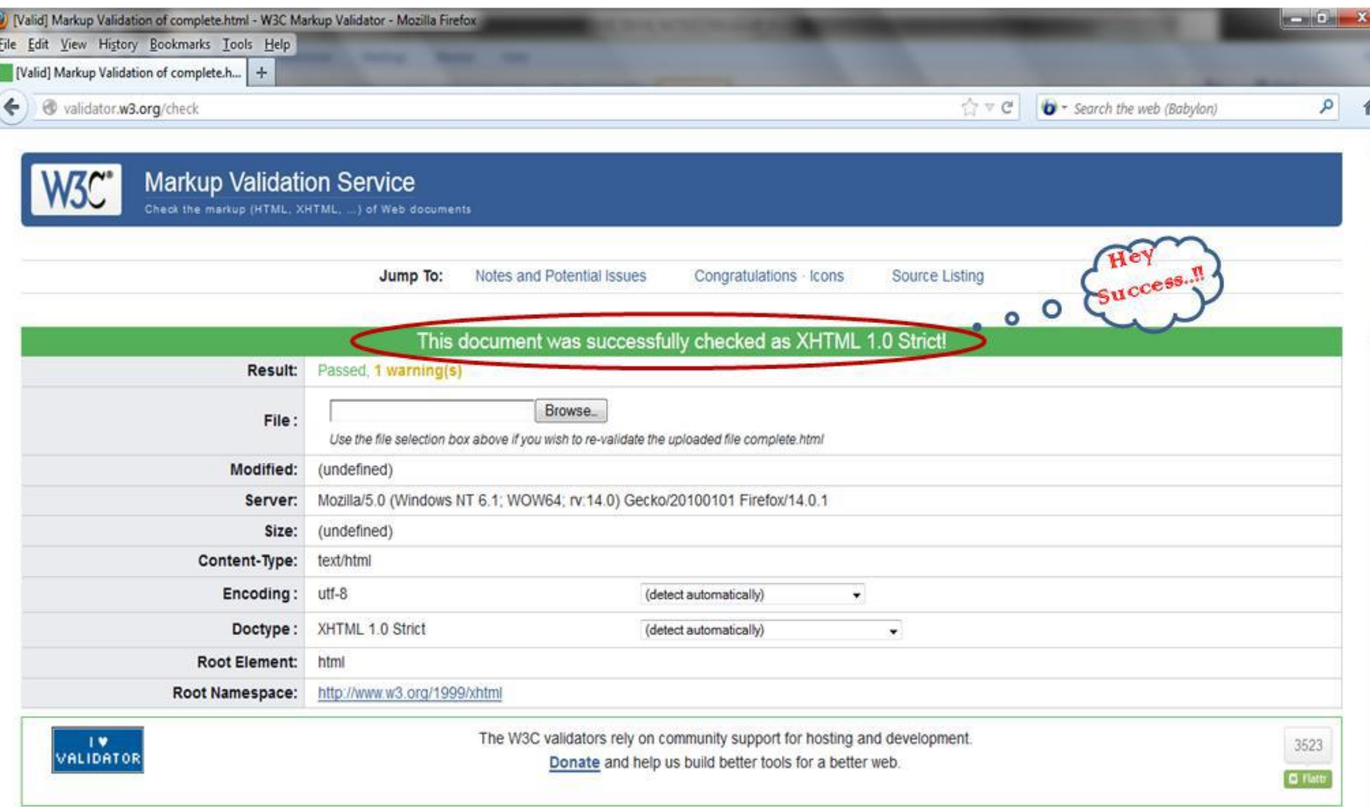
Step 4: Click on "More Options" and select your criteria like *show source*

Step 5: After all the settings, click on "Check" button

Now you will be navigated to another page which shows success or failure.

In our example, the file *complete.html* is a valid XHTML file. So the output shows success..!!

OUT PUT:



1.13 HYPERTEXT LINKS

1.13.1 Links:

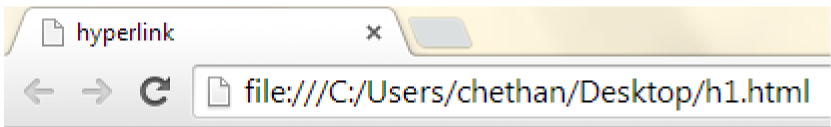
- Hyperlinks are the mechanism which allows the navigation from one page to another.
- The term “hyper” means beyond and “link” means connection
- Whichever text helps in navigation is called hypertext
- Hyperlinks cam be created using **<a>** (anchor tag)
- The attribute that should be used for <a> is **href**

Program: *hyper.html*

```
<html>
<head>
<title> hyperlink </title>
</head>
<a href = "link.html"> CLICK HERE
</a>
</html>
```

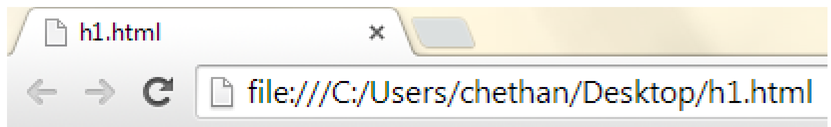
Program: *link.html*

```
<html>
<body> This is Web Programming
</body>
</html>
```



CLICK HERE

After clicking on the above text, we can navigate to another page “link.html” as shown below



This is Web Programming

1.13.2 Targets within Documents:

If the target of a link is not at the beginning of a document, it must be some element within the document, in which case there must be some means of specifying it.

The target element can include an id attribute, which can then be used to identify it in an href attribute. (observe the scroll bar in the outputs given)

```
<html>
<head>
        <title> target link</title>
</head>
 <body>
        <h2 id = “avionics”> Avionics </h2>
        <a href = “#avionics”> What about avionics? </a>
        <a href = “AIDAN1.html#avionics”> Avionics </a>
</body>
</html>
```

Question paper question

1) Give syntax and an example for each of the following tags. 1.<pre> 2.<a> 3.<img> 4.<sub> 5.<p> (10 M)
2) Explain with an example the following tags 1.select 2.frames 3.colspan 4.radio button 5.style class selector (10 M)
3) Give and explain syntax of following tags 1.<blockquote> 2.meta (03 M)
4) Explain the following tags with examples i. <img> ii. <a> (04 M)
5) Syntax and an example for each of the following tags. 1.<pre> 2.<p> 3.<sup> 4.<sub> 5.<blackquote> (10 m)
6) Give the standard structure of XHTML document. How line breaks, heading and fonts are handled in XHTML? (10 M)
7) Explain standard XHTML document structure (08 M)
8) Explain the different image formats, write XHTML document to illustrate use of <img> (with all its attributes) (08 M)
9) Give the syntactic difference between HTML and XHTML (08 M)
10) Discuss the following tags with syntax and example i) <pre> ii. <meta> (04 M)

Module – 2
HTML Tables and Forms:

2.1 Lists,

2.2 Tables,

2.3 Forms,

2.4 Frames

2.5 CSS: Introduction,

2.6 Levels of style sheets,

2.7 Style specification formats,

2.8 Selector forms,

2.9 Property value forms,

2.10 Font properties,

2.11 List properties,

2.12 Color,

2.13 Alignment of text,

2.14 The box model,

2.15 Background images,

2.16 The <span> and <div> tags,

2.17 Conflict resolution.

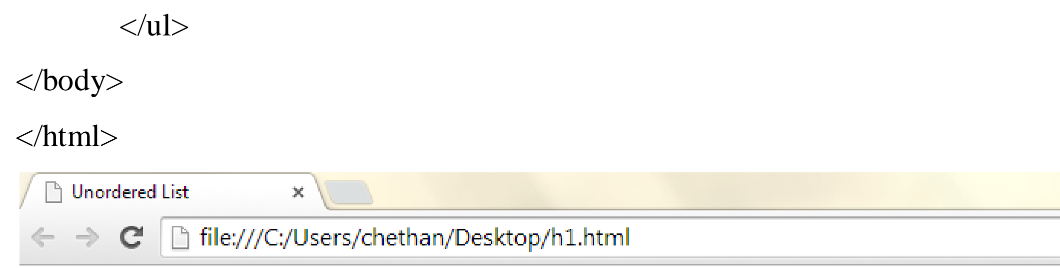
2.1 LISTS

2.1.1 Unordered Lists:

The <ul> tag, which is a block tag, creates an unordered list. Each item in a list is specified with an <li> tag (li is an acronym for *l*ist *i*tem). Any tags can appear in a list item, including nested lists. When displayed, each list item is implicitly preceded by a bullet.

```
<html>
 <head>
            <title> Unordered List </title>
 </head>
<body>
 <h1> Some Common Single-Engine Aircraft </h1>
          <ul>
                    <li> Cessna skyhawk</li>
                    <li> Beechcraft Bonaza</li>
                    <li> piper Cherokee</li>
```

```
        </ul>
</body>
</html>
```



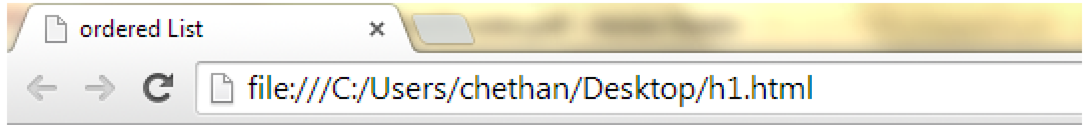
Some Common Single-Engine Aircraft

- Cessna skyhawk
- Beechcraft Bonaza
- piper Cherokee

2.1.2 Ordered Lists:

- Ordered lists are lists in which the order of items is important. This orderedness of a list is shown in the display of the list by the implicit attachment of a sequential value to the beginning of each item. The default sequential values are Arabic numerals, beginning with 1. An ordered list is created within the block tag <ol>.
- The items are specified and displayed just as are those in unordered lists, except that the items in an ordered list are preceded by sequential values instead of bullets.

```
<html>
 <head>
        <title> ordered List </title>
 </head>
<body>
<h3> Cessna 210 Engine Starting Instructions </h3>
        <ol>
                <li> Set mixture to rich </li>
                <li> Set propeller to high RPM </li>
                <li> Set ignition switch to "BOTH" </li>
                <li> Set auxiliary fuel pump switch to "LOW PRIME" </li>
                <li> When fuel pressure reaches 2 to 2.5 PSI, push starter button </li>
        </ol>
</body>
</html>
```



Cessna 210 Engine Starting Instructions

1. Set mixture to rich
2. Set propeller to high RPM
3. Set ignition switch to "BOTH"
4. Set auxiliary fuel pump switch to "LOW PRIME"
5. When fuel pressure reaches 2 to 2.5 PSI, push starter button

2.1.3 Nested Lists:

```
<html>
<head>
        <title> nested lists </title>
</head>
<ol>
  <li> Information Science </li>
    <ol>
        <li>OOMD</li>
        <li>Java & J2ee</li>
<ul>
        <li>classes and methods</li>
        <li>exceptions</li>
        <li>applets</li>
        <li>servelets</li>
</ul>
   <li>Computer Networks</li>
<ul>
        <li>Part 1</li>
        <li>Part 2</li>
</ul>
        <li>DBMS</li>
        <li>Operations Research</li>
  </ol>
        <li> Computer Science</li>
      <ol>
```

```
            <li>Compiler Design</li>
            <li>FLAT</li>
<ul>
            <li>NFA</li>
            <li>DFA</li>
            <li>CFG</li>
 </ul>
            <li>Computer Graphics</li>
            <li>Artificial Intelligence</li>
     </ol>
         </ol>
</html>
```

nested lists ×

file:///C:/Users/chethan/Desktop/h1.html

1. Information Science
    1. OOMD
    2. Java & J2ee
        - classes and methods
        - exceptions
        - applets
        - servelets
    3. Computer Networks
        - Part 1
        - Part 2
    4. DBMS
    5. Operations Research
2. Computer Science
    1. Compiler Design
    2. FLAT
        - NFA
        - DFA
        - CFG
    3. Computer Graphics
    4. Artificial Intelligence

2.1.4 Definition Lists:

- As the name implies, definition lists are used to specify lists of terms and their definitions, as in glossaries. A definition list is given as the content of a <dl> tag, which is a block tag.
- Each term to be defined in the definition list is given as the content of a <dt> tag. The definitions themselves are specified as the content of <dd> tags.
- The defined terms of a definition list are usually displayed in the left margin; the definitions are usually shown indented on the line or lines following the term.

```
<html>
<head>
    <title> Definition List </title>
</head>
<body>
    <h3> Single-Engine Cessna Airplanes </h3>
    <dl >
        <dt> 152 </dt>
            <dd> Two-place trainer </dd>
        <dt> 172 </dt>
            <dd> Smaller four-place airplane </dd>
        <dt> 182 </dt>
            <dd> Larger four-place airplane </dd>
        <dt> 210 </dt>
            <dd> Six-place airplane - high performance
        </dd>
    </dl>
</body>
 </html>
```

Definition List ×

file:///C:/Users/chethan/Desktop/h1.html

Single-Engine Cessna Airplanes

152
    Two-place trainer
172
    Smaller four-place airplane
182
    Larger four-place airplane
210
    Six-place airplane - high performance

2.2 TABLES

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

2.2.1 Basic Table Tags:

- A table is specified as the content of the block tag **<table>.**
- There are two kinds of lines in tables: the line around the outside of the whole table is called the *border*; the lines that separate the cells from each other are called *rules*.
- It can be obtained using **border** attribute. The possible values are “border” or any number.
- The table heading can be created using **<caption>** tag.
- The table row can be created using **<tr>** tag.
- The column can be created either by using **<th>** tag (stands for table header which is suitable for headings) or **<td>** tag (stands for table data which is suitable for other data).

```
<html>
 <head>
        <title> Table with text and image </title>
</head>
<body>
<table border = "border">
<caption>VTU Memo  </caption>
        <tr>
                <th> VTU </th>
                <th> Image </th>
        </tr>
        <tr>
                <td> Funny image </td>
                 <td> <img src = "img(13).jpg" alt = "cant display"/></td>
         </tr>
        <tr>
                <td> True Story </td>
                <td> <img src = "img(19).jpg" alt = "cant display"/></td>
        </tr>
 </table>
</body>
```

</html>

Table with text and image ×

file:///C:/Users/chethan/Desktop/web/h1.html

VTU Memo

VTU	Image
Funny image	Your Face While Answering VIVA...
True Story	THAT MOMENT WHEN YOU SEE THE LECTURER ENTERING THE CLASS ROOM WITH BLUEBOOKS

2.2.2 The rowspan and colspan Attributes:

Multiple-level labels can be specified with the rowspan and colspan attributes.

```
<html>
<head>
        <title>row-span and column-span</title>
</head>
```

```
<body>
        <p> Illustration of Row span</p>
 <table border="border">
            <tr>
                        <th rowspan="2"> ATME</th>
                         <th>ISE</th>
            </tr>
             <tr>
                        <th>CSE</th>
            </tr>
 </table>
        <p> Illustration of Column span</p>
<table border="border">
             <tr>
                        <th colspan="2"> ATME </th>
            </tr>
             <tr>
                        <th>ISE</th>
                         <th>CSE</th>
            </tr>
</table>
 </body>
 </html>
```

row-span and column-sp
file:///C:/Users/chethan/Desktop/web/h1.html

Illustration of Row span

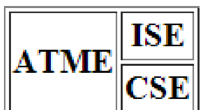
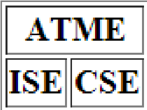


Illustration of Column span



2.2.3 The align and valign Attributes:

- The placement of the content within a table cell can be specified with the align and valign attributes in the <tr>, <th>, and <td> tags.

- The align attribute has the possible values left, right, and center, with the obvious meanings for horizontal placement of the content within a cell.
- The default alignment for th cells is center; for td cells, it is left. The valign attribute of the <th> and <td> tags has the possible values top and bottom.
- The default vertical alignment for both headings and data is center.

```
<html>
<head>
<title> Align and valign </title>
</head>
<body>
<p>Table having entries with different alignments</p>
<table border="border">
        <tr align = "center">
                <th>     </th>
                <th> Column Label </th>
                <th> Another One </th>
                 <th> Still Another  </th>
        </tr>
        <tr>
                <th> Align </th>
                <td align = "left"> Left</td>
                <td align = "center"> Center  </td>
                <td align = "right"> right </td>
         </tr>
        <tr>
                <th> <br/>Valign<br/><br/><br/></th>
                 <td> Deafult </td>
                 <td valign = "top"> Top</td>
                 <td valign = "bottom"> Bottom</td>
        </tr>
</table>
 </body>
</html>
```

Align and valign

file:///C:/Users/chethan/Desktop/web/h1.html

Table having entries with different alignments

	Column Label	**Another One**	**Still Another**
Align	Left	Center	right
Valign	Deafult	Top	Bottom

2.2.4 The cellpadding and cellspacing Attributes:

Cellspacing is the distance between cells.

Cellpadding is the distance between the edges of the cell to its content.

```
<html>
<head>
        <title> cell spacing and cell padding </title>
</head>
<body>
        <h3>Table with space = 10, pad = 50</h3>
 <table border = "7" cellspacing = "10" cellpadding = "50">
        <tr>
                <td> Kswamy</td>
               <td>Chethan </td>
        </tr>
</table>
        <h3>Table with space = 50, pad = 10</h3>
<table border = "7" cellspacing = "50" cellpadding = "10">
        <tr>
                <td> Divya </td>
                <td>Chethan </td>
         </tr>
 </table>
 </body>
</html>
```

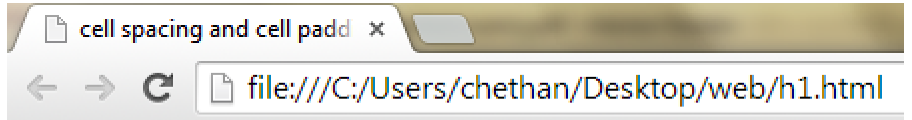


Table with space = 10, pad = 50

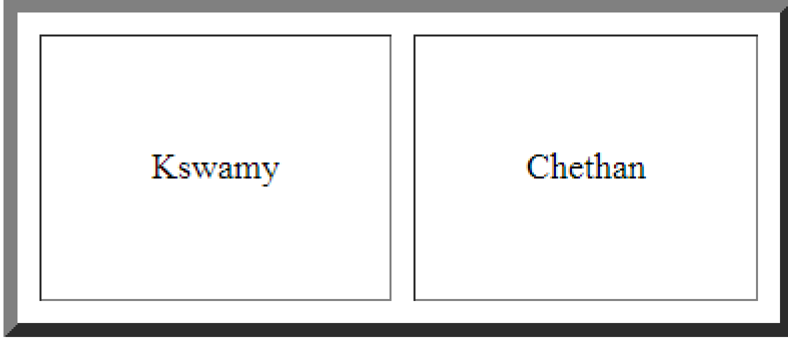
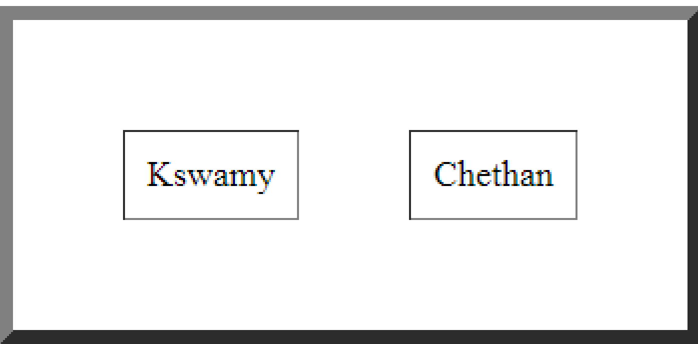


Table with space = 50, pad = 10



2.2.5 Table Sections:

- Tables naturally occur in two and sometimes three parts: header, body, and footer. (Not all tables have a natural footer.)
- These three parts can be respectively denoted in XHTML with the thead, tbody, and tfoot elements.
- The header includes the column labels, regardless of the number of levels in those labels.
- The body includes the data of the table, including the row labels.
- The footer, when it appears, sometimes has the column labels repeated after the body.
- In some tables, the footer contains totals for the columns of data above.
- A table can have multiple body sections, in which case the browser may delimit them with horizontal lines that are thicker than the rule lines within a body section.

2.3 FORMS

The most common way for a user to communicate information from a Web browser to the server is through a form. XHTML provides tags to generate the commonly used objects on a screen form. These objects are called *controls* or *widgets*. There are controls for single-line and multiple-line text collection, checkboxes, radio buttons, and menus, among others. All control tags are inline tags.

2.3.1 The <form> Tag:

All of the controls of a form appear in the content of a <form> tag. A block tag, <form>, can have several different attributes, only one of which, action, is required. The action attribute specifies the URL of the application on the Web server that is to be called when the user clicks the *Submit* button. Our examples of form elements will not have corresponding application programs, so the value of their action attributes will be the empty string ("").

2.3.2 The <input> Tag:

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

- **Text Box**
    - It is a type of input which takes the text.
    - Any type of input can be created using <input>
    - The *type* attribute indicates what type of input is needed for the text box, the value should be given as text.
    - For any type of input, a name has to be provided which is done using *name* attribute.
    - The size of the text can be controlled using *size* attribute.
    - Every browser has a limit on the number of characters it can collect. If this limit is exceeded, the extra characters are chopped off. To prevent this chopping, *maxlength* attribute can be used. When maxlength is used, users can enter only those many characters that is given as a value to the attribute.

```
<html>
<head>
        <title>Text Box</title>
</head>
<body>
<form action = " ">
        <p> <label>Enter your Name:
                <input type = "text" name = "myname" size = "20" maxlength = "20" />
        </label> </p>
```

```
</form>
</body>
 </html>
```

Text Box

file:///C:/Users/chethan/Desktop/web/h1.html

Enter your Name:

2.3.3 Password Box

- If the contents of a text box should not be displayed when they are entered by the user, a password control can be used.
- In this case, regardless of what characters are typed into the password control, only bullets or asterisks are displayed by the browser.

```
<html>
<head>
        <title>Password Box</title>
</head>
 <body>
<form action = " ">
        <p> <label>Enter the email id:
                <input type = "text" name = "myname" size = "24" maxlength = "25" /> </label> </p>
        <p> <label>Enter the password:
                <input type = "password" name = "mypass" size = "20" maxlength = "20" />
        </label> </p>
</form>
</body>
 </html>
```

Password Box

file:///C:/Users/chethan/Desktop/web/h1.html

Enter the email id: kswamy.chethan@gmail.com

Enter the password: •••••••••••••••

2.3.4 Radio Button

- Radio buttons are special type of buttons which allows the user to select only individual option
- Radio buttons are created using the input tag with the *type* attribute having the value **radio.**
- When radio buttons are created, values must be provided with the help of *value* attribute.
- All the radio buttons which are created would have same name. This is because the radio buttons are group elements.
- If one of the radio buttons has to be selected as soon as the web page is loaded, checked attribute should be used. The value also would be checked.

```
<html>
<head>
	<title>Radio Button</title>
</head>
<body>
	<h3>Age Category ?</h3>
<form action = " ">
<p>
	<label><input type="radio" name="age" value="under20" checked = “checked”/>0-19 </label>
	<label><input type="radio" name="age" value="20-35"/>20-35</label>
	<label><input type="radio" name="age" value="36-50"/>36-50 </label>
	<label><input type="radio" name="age" value=" over50"/>over50</label>
</p>
</form>
</body>
</html>
```

Age Category ?

0-19 20-35 36-50 over50

2.3.5 Check Box

- Check box is a type of input using which multiple options can be selected.
- Check box can also be created using the <input> tag with the *type* having the value “checkbox”.
- During the creation of check box, the value should be provided using the *value* attribute.
- All the checkbox which are created would have the same name because they are group elements.
- If one of the check box have to be selected as soon as the page is loaded, checked attribute should be used with the value checked.

```
<html>
<head>
        <title>Check Box</title>
</head>
<body>
        <h3>Grocery Checklist</h3>
<form action = " ">
   <p>
        <label><input type="checkbox" name="groceries" value="milk" checked=”checked”/>Milk</label>
        <label><input type="checkbox" name=" groceries" value="bread"/> Bread </label>
        <label><input type="checkbox" name=" groceries" value="eggs"/>Eggs</label>
    </p>
</form>
</body>
</html>
```

Check Box ×

file:///C:/Users/chethan/Desktop/web/h1.html

Grocery Checklist

2.3.6 The <select> Tag:

- Menu items is another type of input that can be created on the page.
- To create the menu item, <select> tag is used.
- To insert the item in the menu, <option> tag is used.

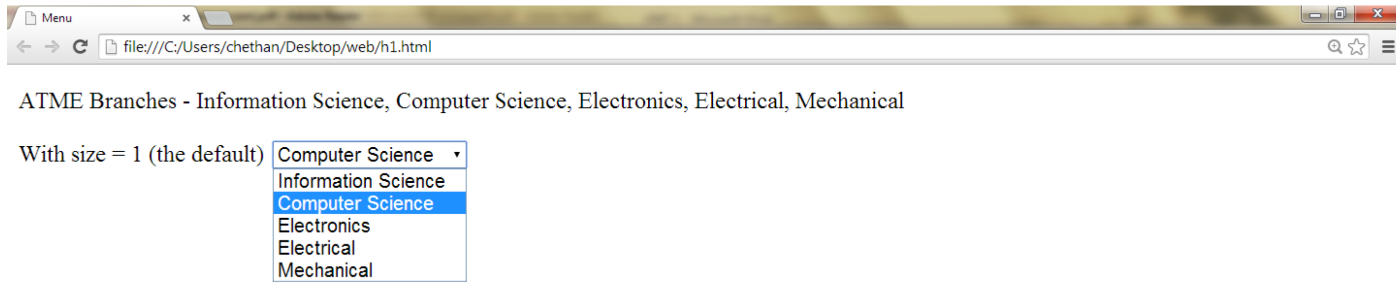
```
<html>
<head>
        <title> Menu </title>
</head>
<body>
<p> ATME Branches - Information Science, Computer Science, Electronics, Electrical, Mechanical </p>
<form action = "">
        <p> With size = 1 (the default)
                <select name = "branches">
                        <option> Information Science </option>
                        <option> Computer Science </option>
                        <option> Electronics </option>
                        <option> Electrical </option>
                         <option> Mechanical </option>
                </select>
        </p>
</form>
</body>
</html>
```

Menu

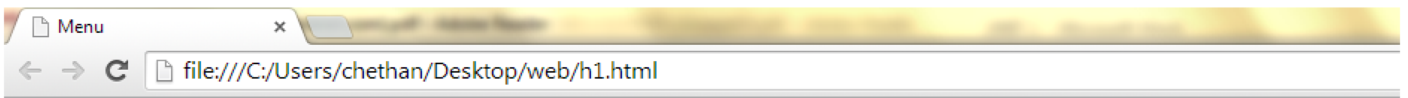
file:///C:/Users/chethan/Desktop/web/h1.html

ATME Branches - Information Science, Computer Science, Electronics, Electrical, Mechanical

With size = 1 (the default) Information Science



If you give <select name = "branches" size = “3”>, then you will get a scroll bar instead of drop down menu. It is as shown in the output given below:



ATME Branches - Information Science, Computer Science, Electronics, Electrical, Mechanical

With size = 1 (the default) Information Science
Computer Science
Electronics
Electrical

2.3.7 The <textarea> Tag:

- Text area is a type of input using which multiple statements can be entered.
- Text area is created using <textarea> tag.
- Text area should have the name.
- During the creation of text area, it should be mentioned how many sentences can be entered. This is done using *rows* attribute.
- Similarly, it should also be mentioned how many characters can be entered in a line. This is done using *cols* attribute.
- If the value given to rows is exceeded i.e. if users enter sentences more than specified, the *scroll bar* automatically appears.

```
<html>
<head>
        <title> text area </title>
</head>
<body>
<form action=" ">
        <h3> Enter your comments</h3>
        <p>
                <textarea name="feedback" rows="5" cols="100">
                (Be Brief and concise)
                </textarea>
         </p>
 </form>
</body>
 </html>
```

text area
file:///C:/Users/chethan/Desktop/web/h1.html

Enter your comments

(Be Brief and concise)

2.3.8 The Action Buttons:

The *Reset* button clears all of the controls in the form to their initial states. The *Submit* button has two actions: First, the form data is encoded and sent to the server; second, the server is requested to execute the server-resident program specified in the action attribute of the <form> tag.

The purpose of such a server-resident program is to process the form data and return some response to the user. Every form requires a *Submit* button.

The *Submit* and *Reset* buttons are created with the <input> tag.

```
<html>
<head>
        <title> action buttons </title>
</head>
<body>
<form action=" ">
        <p>
                <input type="SUBMIT" value="SUBMIT"/>
                <input type="RESET" value="RESET"/>
        </p>
</form>
 </body>
</html>
```

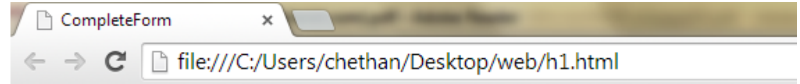
action buttons
file:///C:/Users/chethan/Desktop/web/h1.html

SUBMIT RESET

NOTE: A *plain* button has the type button. *Plain* buttons are used to choose an action.

2.3.9 Example of a Complete Form:

```
<html>
<head>
        <title> CompleteForm</title>
</head> <body>
<h1>Registration Form</h1>
<form action=" ">
<p>  <label>Enter your email id:
        <input type = "text" name = "myname" size = "24" maxlength = "25" />
</label> </p>
<p> <label>Enter the password:
        <input type = "password" name = "mypass" size = "20" maxlength = "20" />
</label> </p>
<p>Sex</p>
<p>
        <label><input type="radio" name="act" value="one"/>Male</label>
        <label><input type="radio" name="act" value="two"/>Female</label>
</p>
<p>Which of the following Accounts do you have?</p>
<p>
        <label><input type="checkbox" name="act" value="one"/>Gmail</label>
        <label><input type="checkbox" name="act" value="two"/>Facebook</label>
        <label><input type="checkbox" name="act" value="three"/>Twitter</label>
        <label><input type="checkbox" name="act" value="four"/>Google+</label>
</p>
<p> Any Suggestions?</p>
        <p> <textarea name="feedback" rows="5" cols="100"> </textarea> </p>
        <p>Click on Submit if you want to register</p>
        <p> <input type="SUBMIT" value="SUBMIT"/>
        <input type="RESET" value="RESET"/>
</p>
</form>
</body>
</html>
```



Registration Form

Enter your email id: kswamy.chethan@gmail.com

Enter the password: •••••••••••••••

Sex

◉ Male ○ Female

Which of the following Accounts do you have?

☑ Gmail ☑ Facebook ☐ Twitter ☐ Google+

Any Suggestions?

Click on Submit if you want to register

SUBMIT RESET

2.4 FRAMES

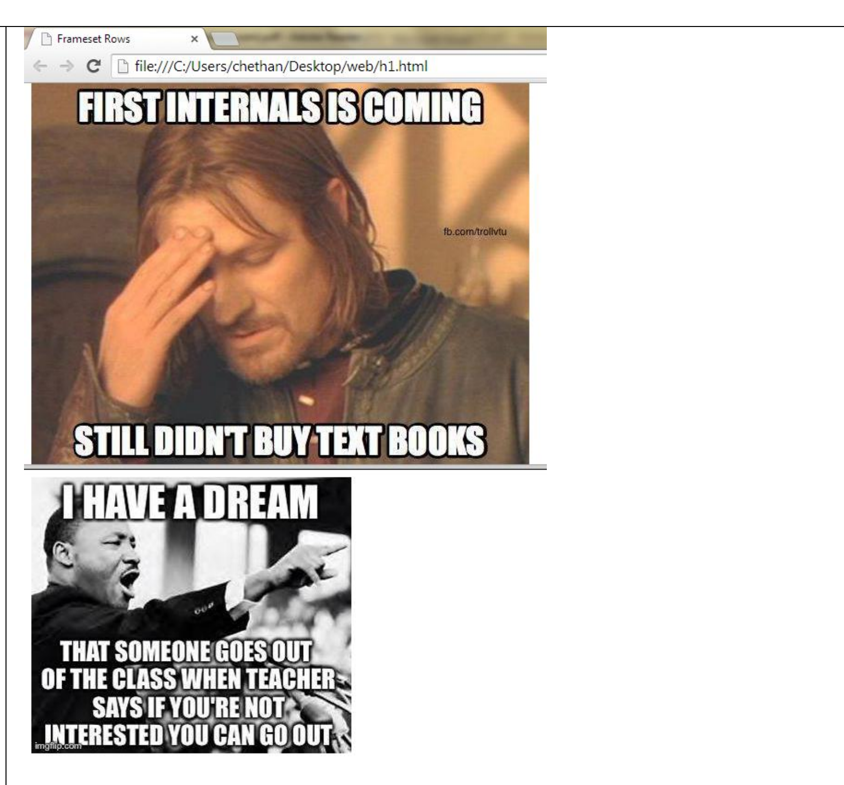
The browser window can be used to display more than one document at a time. The window can be divided into rectangular areas, each of which is a *frame*. Each frame is capable of displaying its own document.

2.4.1 Framesets:

- The number of frames and their layout in the browser window are specified with the <frameset> tag.
- A frameset element takes the place of the body element in a document. A document has either a body or a frameset but cannot have both.
- The <frameset> tag must have either a *rows* or a *cols* attribute. (or both)
- To create horizontal frames, *rows* attribute is used.
- To create vertical frames, *cols* attribute is used.
- The values for these attributes can be numbers, percentages and asterisks.
- Two or more values are separated by commas & given in quoted string.

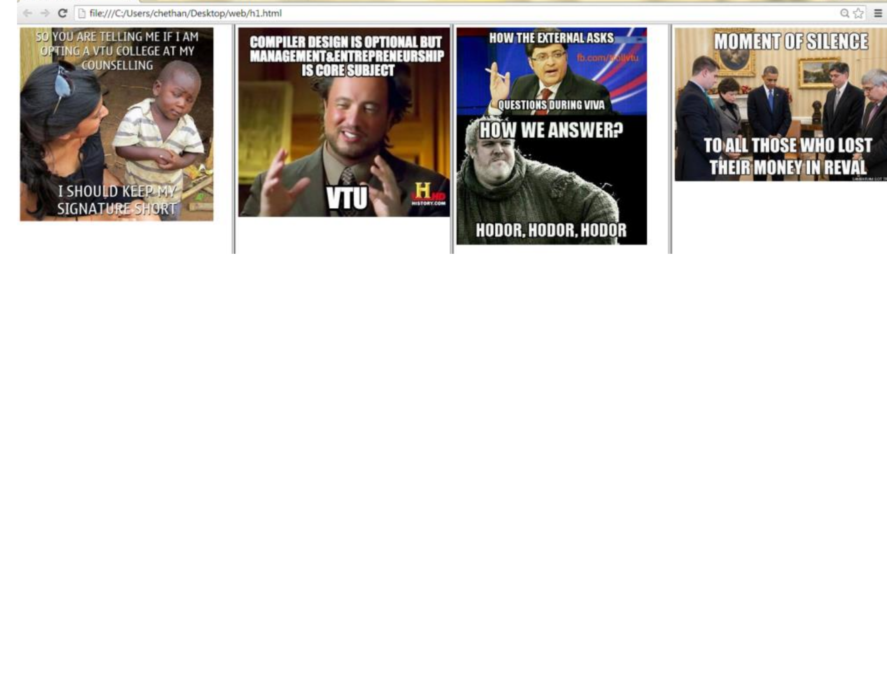
To Demonstrate Horizontal Frames using *rows* Attribute

```
<html>
<head>
<title>Frameset Rows</title>
</head>
<frameset rows = "*,*">
<frame src = "Framerow1.html"/>
<frame src = ""Framerow2.html"/>
</frameset>
</html>
```



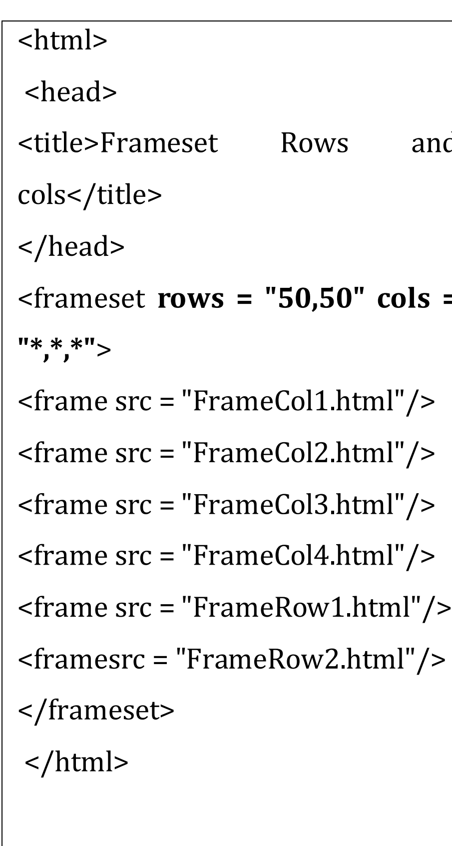
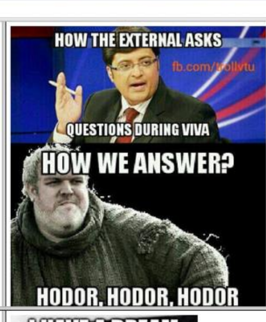
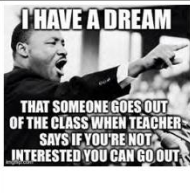
To Demonstrate Vertical Frames using *cols* Attribute

```
<html>
<head>
<title>Frameset Cols</title>
</head>
<frameset cols
="25%,25%,25%,25%">
<frame src = "FrameCol1.html"/>
<frame src = "FrameCol2.html"/>
<frame src = "FrameCol3.html"/>
<frame src = "FrameCol4.html"/>
</frameset>
</html>
```



Note: Here, the programs FrameRow1.html, FrameRow2.html, FrameCol1.html, FrameCol2.html, FrameCol3.html, FrameCol4.html are programs to display images. They must be coded separately.

<html> <head> <title>frame row 1</title> </head> <body> </body> </html>	<html> <head> <title>frame col 1</title> </head> <body> </body> </html>	<html> <head> <title>frame col 3</title> </head> <body> </body> </html>
<html> <head> <title>frame row 2</title> </head> <body> </body> </html>	<html> <head> <title>frame col 3</title> </head> <body> </body> </html>	<html> <head> <title>frame col 4</title> </head> <body> </body> </html>

<html> <head> <title>Frameset Rows and cols</title> </head> <frameset **rows = "50,50" cols = "*,*,*"**> <frame src = "FrameCol1.html"/> <frame src = "FrameCol2.html"/> <frame src = "FrameCol3.html"/> <frame src = "FrameCol4.html"/> <frame src = "FrameRow1.html"/> <framesrc = "FrameRow2.html"/> </frameset> </html>	  

2.4.2SYNTACTIC DIFFERENCES BETWEEN HTML AND XHTML

PARAMETERS	HTML	XHTML
Case Sensitivity	Tags and attributes names are case insensitive	Tags and attributes names must be in lowercase
Closing tags	Closing tags may be omitted	All elements must have closing tag
Quoted attribute values	Special characters are quoted. Numeric values are rarely quoted.	All attribute values must be quoted including numbers
Explicit attribute values	Some attribute values are implicit. For example: <table border>. A default value for border is assumed	All attribute values must be explicitly stated
***id* and *name* attributes**	Both *id* and *name* attributes are encouraged	Use of *id* is encouraged and use of *name* is discouraged
Element nesting	Rules against improper nesting of elements (for example: a form element cannot contain another form element) are not enforced.	All nesting rules are strictly enforced

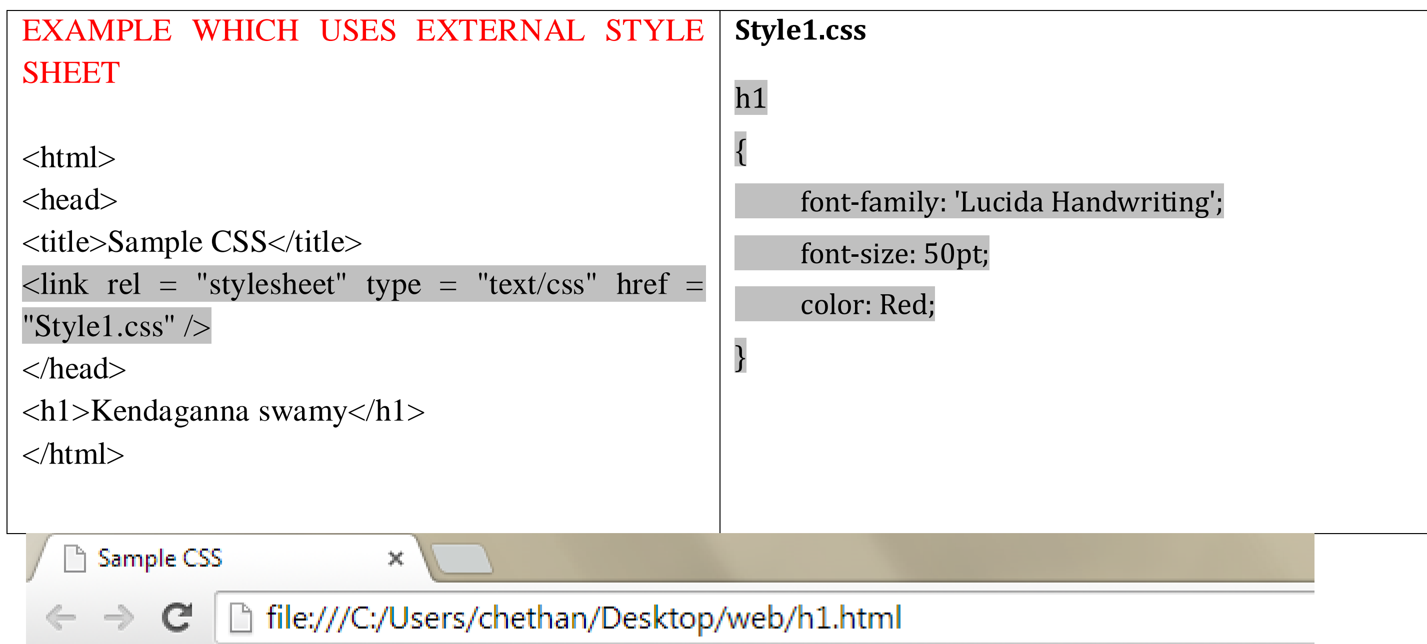
2.5 CSS: Introduction:

XHTML style sheets are called *cascading* style sheets because they can be defined at three different levels to specify the style of a document. Lower level style sheets can override higher level style sheets, so the style of the content of a tag is determined, in effect, through a cascade of style-sheet applications.

2.6 Levels of style sheets:

- The three levels of style sheets, in order from lowest level to highest level, are inline, document level, and external.
- **Inline style sheets** apply to the content of a single XHTML element.
- **Document-level style sheets** apply to the whole body of a document.
- **External style sheets** can apply to the bodies of any number of documents.
- Inline style sheets have precedence over document style sheets, which have precedence over external style sheets.
- Inline style specifications appear within the opening tag and apply only to the content of that tag.
- Document-level style specifications appear in the document head section and apply to the entire body of the document.
- External style sheets stored separately and are referenced in all documents that use them.
- External style sheets are written as text files with the MIME type text/css.
- They can be stored on any computer on the Web. The browser fetches external style sheets just as it fetches documents.

- The <link> tag is used to specify external style sheets. Within <link>, the rel attribute is used to specify the relationship of the linked-to document to the document in which the link appears. The href attribute of <link> is used to specify the URL of the style sheet document.



EXAMPLE WHICH USES EXTERNAL STYLE SHEET	**Style1.css**

<html>
<head>
<title>Sample CSS</title>
<link rel = "stylesheet" type = "text/css" href = "Style1.css" />
</head>
 Kendaganna swamy
</html>
	h1
	{
	font-family: 'Lucida Handwriting';
	font-size: 50pt;
	color: Red;
	}

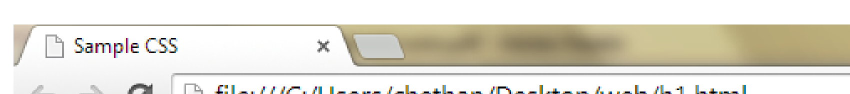
EXAMPLE WHICH USES DOCUMENT LEVEL STYLE SHEET

```
<html>
<head>
        <title>Sample CSS</title>
<style type = "text/css">
h1
{
        font-family: 'Lucida Handwriting';
        font-size: 50pt;
        color: Red;
}
</style>
</head>
        <h1>Kendaganna swamy</h1>
 </html>
```

Kendaganna swamy

EXAMPLE WHICH USES INLINE STYLE SHEET

```
<html>
<head>
<title>Sample CSS</title>
</head>
        <h1 style ="font-family: 'Lucida Handwriting';
        font-size: 50pt;
         color: Red;"> Chethan Kswamy</h1>
</html>
```



Chethan Kswamy

2.7 STYLE SPECIFICATION FORMATS

Inline Style Specification:

*Style = "Property1 : Value1; Property2 : Value2; Property3 : Value3; .... Property_**n** : Value_**n**;"*

Document Style Specification:

<style type = "text/css"> *Rule list* </style> Each style rule in a rule list has two parts: a selector, which indicates the tag or tags affected by the rule, and a list of property–value pairs. The list has the same form as the quoted list for inline style sheets, except that it is delimited by braces rather than double quotes. So, the form of a style rule is as follows:

Selector { *Property1 : Value1; Property2 : Value2; Property3 : Value3; .... Property_**n** : Value_**n**;*
}

[For examples on all three levels of style sheets along with specifications, Please refer the previous examples] .

2.8 SELECTOR FORMS

Simple Selector Forms:

In case of simple selector, a tag is used. If the properties of the tag are changed, then it reflects at all the places when used in the program.

The selector can be any tag. If the new properties for a tag are not mentioned within the rule list, then the browser uses default behaviour of a tag.

```
<html>
<head>
          <title>Sample CSS</title>
<style type = "text/css">
p { font-family: 'Lucida Handwriting'; font-size: 50pt; color: Red; }
</style>
</head>
           <body>
                     <p>Kendaganna Swamy </p>
                    <p>Sunil </p>
                     <p>Siddiq shariff</p>
          </body>
 </html>
```

Sample CSS

file:///C:/Users/chethan/Desktop/web/h1.html

Kendaganna Swamy

Sunil

Siddiq shariff

Class Selectors:

In class selector, it is possible to give different properties for different elements

```
<html>
<head>
 <title>Sample CSS</title>
<style type = "text/css">
          p.one { font-family: 'Lucida Handwriting'; font-size: 25pt; color: Red; }
```

```
        p.two { font-family: 'Monotype Corsiva'; font-size: 50pt; color: green; }
</style>
 </head>
        <body>
                <p class = "one">Kendaganna Swamy</p>
                <p class = "two">Sunil</p>
        </body>
</html>
```

Generic Selectors:

In case of generic selector, when the class is created, it would not be associated to any particular tag. In other words, it is generic in nature.

```
<html>
<head>
        <title>Sample CSS</title>
<style type = "text/css">
        .one { font-family: 'Monotype Corsiva'; color: green; }
</style>
</head>
        <body>
                <p class = "one">KSwamy</p>
                <h1 class = "one">Sunil</h1>
                <h6 class = "one"> Siddiq </h6>
        </body>
</html>
```

Sample CSS

file:///C:/Users/chethan/Desktop/web/h1.html

KSwamy

Sunil

Siddiq

id Selectors:

An id selector allows the application of a style to one specific element.

```
<html>
<head>
<title>Sample CSS</title>
<style type = "text/css">
        #one { font-family: 'lucida calligraphy'; color: purple; }
        #two { font-family: 'comic sans ms'; color: orange; }
</style>
</head>
        <body>
                <p id = "two">Kswamy</p>
                <h1 id = "one">Sunil</h1>
        </body>
</html>
```

Sample CSS

file:///C:/Users/chethan/Desktop/web/h1.html

Kswamy

Sunil

Universal Selectors:

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

```
<html>
<head>
<title>Sample CSS</title>
<style type = "text/css">
        * { font-family: 'lucida calligraphy'; color: purple; }
</style>
</head>
        <body>
                <h1>Kswamy</h1>
                <h2>Sunil</h2>
                <h3>Siddiq</h3>
                <p>Gagana</p>
        </body>
</html>
```

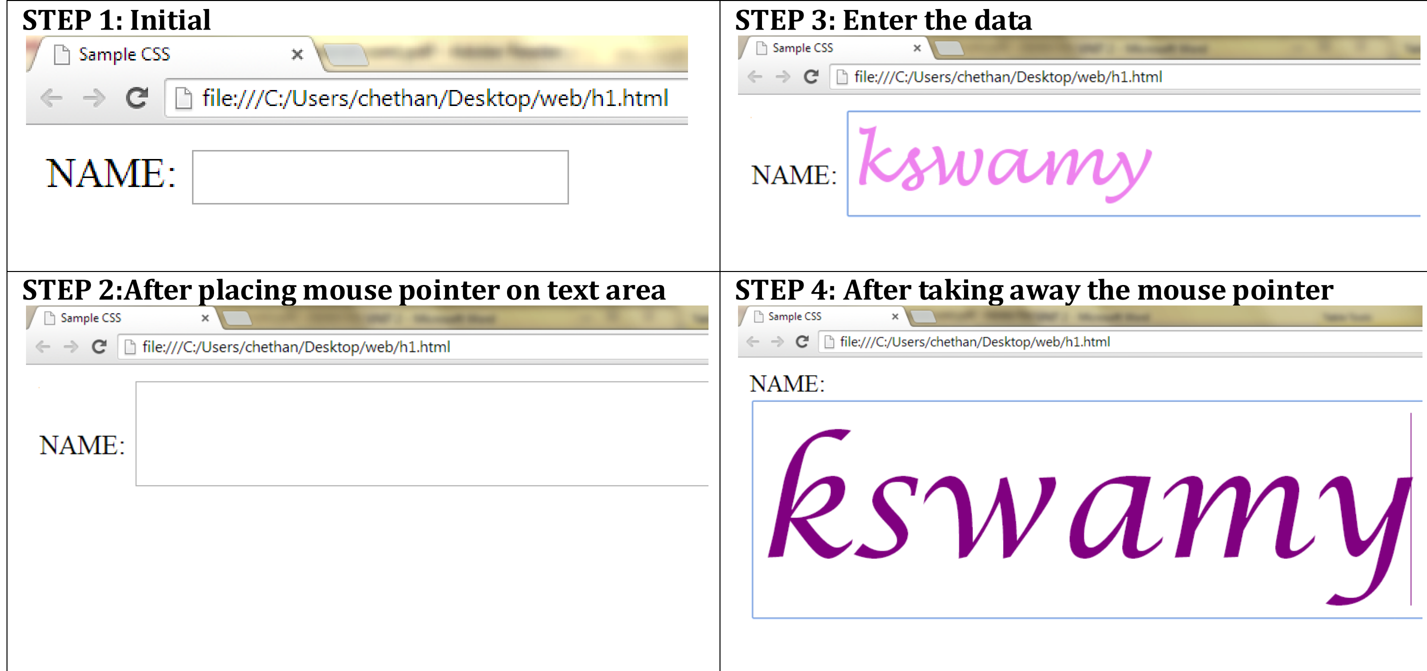
Pseudo Classes:

Pseudo class selectors are used if the properties are to be changed dynamically.

For example: when mouse movement happens, in other words, hover happens or focus happens.

```
<html>
<head>
        <title>Sample CSS</title>
<style type = "text/css">
```

```
        input:focus {  font-family: 'lucida calligraphy'; color: purple; font-size:100; }
        input:hover {  font-family: 'lucida handwriting'; color: violet; font-size:40; }
</style>
</head>
 <body>
        <form action = " ">
                <p>
                        <label> NAME: <input type = "text" />
                         </label>
                 </p>
         </form>
 </body>
</html>
```



2.9 PROPERTY VALUE FORMS

CSS includes 60 different properties in seven categories: fonts, lists, alignment of text, margins, colours, backgrounds, and borders. Property values can appear in a variety of forms.

- Keyword property values are used when there are only a few possible values and they are predefined.
- A number value can be either an integer or a sequence of digits with a decimal point and can be preceded by a sign (+ or -).
- Length values are specified as number values that are followed immediately by a two-character abbreviation of a unit name. The possible unit names are px, for pixels; in, for inches; cm, for centimeters; mm, for millimeters; pt, for points.

- Percentage values are used to provide a measure that is relative to the previously used measure for a property value. Percentage values are numbers that are followed immediately by a percent sign (%).Percentage values can be signed. If preceded by a plus sign, the percentage is added to the previous value; if negative, the percentage is subtracted.
- There can be no space between url and the left parenthesis.
- Color property values can be specified as color names, as six-digit hexadecimal numbers, or in RGB form. RGB form is just the word rgb followed by a parenthesized list of three numbers that specify the levels of red, green, and blue, respectively. The RGB values can be given either as decimal numbers between 0 and 255 or as percentages. Hexadecimal numbers must be preceded with pound signs (#), as in #43AF00.

2.10 FONT PROPERTIES

Font Families:

The font-family property is used to specify a list of font names. The browser uses the first font in the list that it supports. For example, the property:

font-family: Arial, Helvetica, Futura

tells the browser to use Arial if it supports that font. If not, it will use Helvetica if it supports it. If the browser supports neither Arial nor Helvetica, it will use Futura if it can. If the browser does not support any of the specified fonts, it will use an alternative of its choosing. If a font name has more than one word, the whole name should be delimited by single quotes, as in the following example:

font-family: 'Times New Roman'

Font Sizes:

The font-size property does what its name implies. For example, the following property specification sets the font size for text to 10 points:

font-size: 10pt

Many relative font-size values are defined, including xx-small, x-small, small, medium, large, x-large, and xx-large. In addition, smaller or larger can be specified. Furthermore, the value can be a percentage relative to the current font size.

Font Variants:

The default value of the font-variant property is normal, which specifies the usual character font. This property can be set to small-caps to specify small capital characters. These characters are all uppercase, but the letters that are normally uppercase are somewhat larger than those that are normally lowercase.

Font Styles:

The font-style property is most commonly used to specify italic, as in

font-style: italic

Font Weights:

The font-weight property is used to specify the degree of boldness, as in

font-weight: bold

Besides bold, the values normal, bolder, and lighter can be specified. Specific numbers also can be given in multiples of 100 from 100 to 900, where 400 is the same as normal and 700 is the same as bold.

Font Shorthands:

If more than one font property must be specified, the values can be stated in a list as the value of the font property. The order in which the property values are given in a font value list is important. The order must be as follows: The font names must be last, the font size must be second to last, and the font style, font variant, and font weight, when they are included, can be in any order but must precede the font size and font names.

font: bold 14pt 'Times New Roman'

```
<html>
<head>
	<title>Font Properties</title>
<style type = "text/css">
p.one
{
	 font-family: 'lucida calligraphy';
	font-weight:bold;
	font-size:75pt;
	color: purple;
}
h1.two
{
	font-family: 'cambria';
	color: violet;
	font-style:italics;
}
```

```
p.three
{
        font: small-caps italic bold 50pt 'times new roman'
}
</style>
</head>
        <body>
                <p class = "one">Kswamy Chethan</p>
                <h1 class = "two">Sunil Kumar </h1>
                <p class = "three">Siddiq Shariff</p>
        </body>
</html>
```

Font Properties

file:///C:/Users/chethan/Desktop/web/h1.html

Kswamy Chethan

Sunil Kumar

SIDDIQ SHARIFF

Text Decoration:

The text-decoration property is used to specify some special features of text.

The available values are line-through, overline, underline, and none, which is the default.

```
<html>
<head>
        <title>Text Decoration</title>
<style type = "text/css">
h1.one {text-decoration: line-through;}
h1.two {text-decoration: overline;}
h1.three {text-decoration: underline;}
</style>
</head>
```

```
    <body>
        <h1 class = "one">Kswamy Chethan </h1>
    <p>[This is line-through]</p><br/>
        <h1 class = "two"> Sunil Kumar </h1>
    <p>[This is overline]</p><br/>
        <h1 class = "three"> Siddiq Shariff </h1>
    <p>[This is underline]</p><br/>
    </body>
</html>
```

Text Decoration

file:///C:/Users/chethan/Desktop/web/h1.html

~~Kswamy Chethan~~

[This is line-through]

Sunil Kumar

[This is overline]

Siddiq Shariff

[This is underline]

2.11 LIST PROPERTIES

Two presentation details of lists can be specified in XHTML documents: the shape of the bullets that precede the items in an unordered list and the sequencing values that precede the items in an ordered list. The list-style-type property is used to specify both of these. The **list-style-type** property of an unordered list can be set to disc, circle, square, or none.

```
<html>
<head>
    <title>CSS Bullets</title>
<style type = "text/css">
    li.one {list-style-type:disc}
    li.two{list-style-type:square}
    li.three{list-style-type:circle}
</style>
</head>
```

```
<body>
<h3>Crazy Boy's </h3>
    <ul>
        <li class = "one"> Kendeganna Swamy</li>
        <li class = "two"> Sunil Kumar</li>
        <li class = "three"> Siddiq Shariff</li>
    </ul>
</body>
</html>
```

CSS Bullets

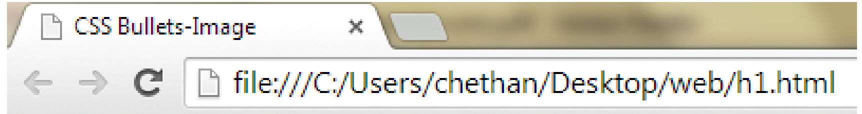
file:///C:/Users/chethan/Desktop/web/h1.html

Crazy Boy's

- Kendeganna Swamy
- Sunil Kumar
- Siddiq Shariff

Bullets in unordered lists are not limited to discs, squares, and circles. Any image can be used in a list item bullet. Such a bullet is specified with the list-style-image property, whose value is specified with the url form.

```
<html>
<head>
    <title>CSS Bullets-Image</title>
<style type = "text/css">
li.image {list-style-image: url(bullet.png); font-size:25pt;}
</style>
</head>
<body>
<h1>Crazy Boy's </h3>
    <ul>
        <li class = "image"> Kendeganna Swamy</li>
        <li class = "image "> Sunil Kumar</li>
        <li class = "image"> Siddiq Shariff</li>
    </ul>
</body>
</html>
```



Crazy Boy's

Kendeganna Swamy

Sunil Kumar

Siddiq Shariff

The following example illustrates the use of different sequence value types in nested lists:

```
<html>
<head>
<title> CSS nested lists </title>
<style type = "text/css">
        ol {list-style-type:upper-roman;}
        ol ol {list-style-type:upper-alpha;}
        ol ol ol {list-style-type:decimal;}
</style>
</head>
        <ol>
                <li> Information Science </li>
        <ol>
<li>OOMD</li>
<li>Java & J2ee</li>
        <ol>
                <li>classes and methods</li>
                <li>exceptions</li>
                <li>applets</li>
                <li>servelets</li>
        </ol>
<li>Computer Networks</li>
        <ol>
                <li>Part 1</li>
                <li>Part 2</li>
        </ol>
 <li>DBMS</li>
```

```
<li>Operations Research</li>
</ol>
<li> Computer Science</li>
            <ol>
                        <li>Compiler Design</li>
                        <li>FLAT</li>
            <ol>
<li>NFA</li>
<li>DFA</li>
 <li>CFG</li>
</ol>
             <li>Computer Graphics</li>
            <li>Artificial Intelligence</li>
 </ol>
</ol>
 </html>
```

CSS nested lists

file:///C:/Users/chethan/Desktop/web/h1.html

I. Information Science
    A. OOMD
    B. Java & J2ee
        1. classes and methods
        2. exceptions
        3. applets
        4. servelets
    C. Computer Networks
        1. Part 1
        2. Part 2
    D. DBMS
    E. Operations Research
II. Computer Science
    A. Compiler Design
    B. FLAT
        1. NFA
        2. DFA
        3. CFG
    C. Computer Graphics
    D. Artificial Intelligence

2.12 COLOR

Color Groups:

Three levels of collections of colours might be used by an XHTML document. The smallest useful set of colours includes only those that have standard names and are guaranteed to be correctly displayable by all browsers on all color monitors. This collection of 17 colours is called the *named colours*.

Name	Hexadecimal Code	Name	Hexadecimal Code
aqua	00FFFF	olive	808000
black	000000	orange	FFA500
blue	0000FF	purple	800080
fuchsia	FF00FF	red	FF0000
gray	808080	silver	C0C0C0
green	008000	teal	008080
lime	00FF00	white	FFFFFF
maroon	800000	yellow	FFFF00
navy	000080		

Larger set of colors, called the Web palette, consists of 216 colors. The colors of the Web palette can be viewed at **http://www.web-source.net/216_color_chart.htm**

Color Properties:

The color property is used to specify the foreground color of XHTML elements.

```
<html>
<head>
        <title>Colours</title>
<style type = "text/css">
        p.one {color: pink; }
        p.two {color: # 9900FF; }
        p.three {background-color:#99FF00;}
</style>
</head>
        <body>
                <p class = "one">Kendaganna Swamy</p>
                <p class = "two">Sunil Kumar </p>
                <p class = "three">Siddiq Shariff</p>
        </body>
</html>
```

Colours

file:///C:/Users/chethan/Desktop/web/h1.html

Kendaganna Swamy

Sunil Kumar

Siddiq Shariff

2.13 ALIGNMENT OF TEXT

The text-indent property can be used to indent the first line of a paragraph. This property takes either a length or a percentage value. The text-align property, for which the possible keyword values are left, center, right, and justify, is used to arrange text horizontally.

The float property is used to specify that text should flow around some element, often an image or a table. The possible values for float are left, right, and none, which is the default.

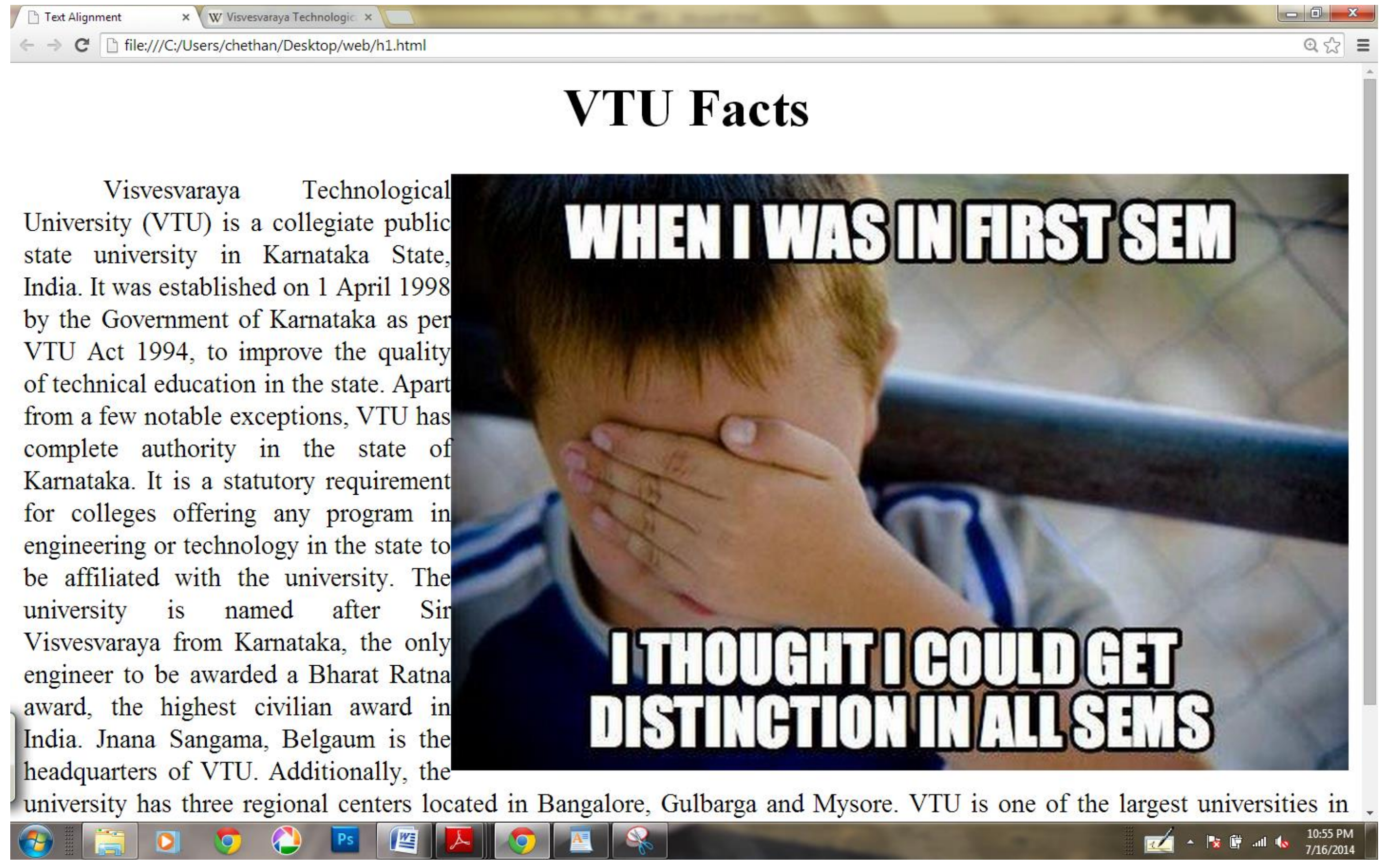
```
<html>
<head>
<title>Text Alignment</title>
<style type = "text/css">
h1.one {text-align: center}
p.two {text-indent: 0.5in; text-align: justify;}
img{float:right}
</style>
</head>
<body>
<h1 class = "one">VTU Facts</h1>
<p>
<img src = "img19.jpg" alt="error"/>
 </p>
<p class = "two">Visvesvaraya Technological University (VTU) is a collegiate public state university in
Karnataka State, India. It was established on 1 April 1998 by the Government of Karnataka as per VTU Act
1994, to improve the quality of technical education in the state. Apart from a few notable exceptions, VTU
has complete authority in the state of Karnataka. It is a statutory requirement for colleges offering any
program in engineering or technology in the state to be affiliated with the university.
```

The university is named after Sir Visvesvaraya from Karnataka, the only engineer to be awarded a Bharat Ratna award, the highest civilian award in India. Jnana Sangama, Belgaum is the headquarters of VTU. Additionally, the university has three regional centers located in Bangalore, Gulbarga and Mysore.

VTU is one of the largest universities in India with 208 colleges affiliated to it with an intake capacity of over 67100 undergraduate students and 12666 postgraduate students. The university encompasses various technical & management fields which offers a total of 30 undergraduate and 71 postgraduate courses. The university has around 1800 PhD candidates.</p>

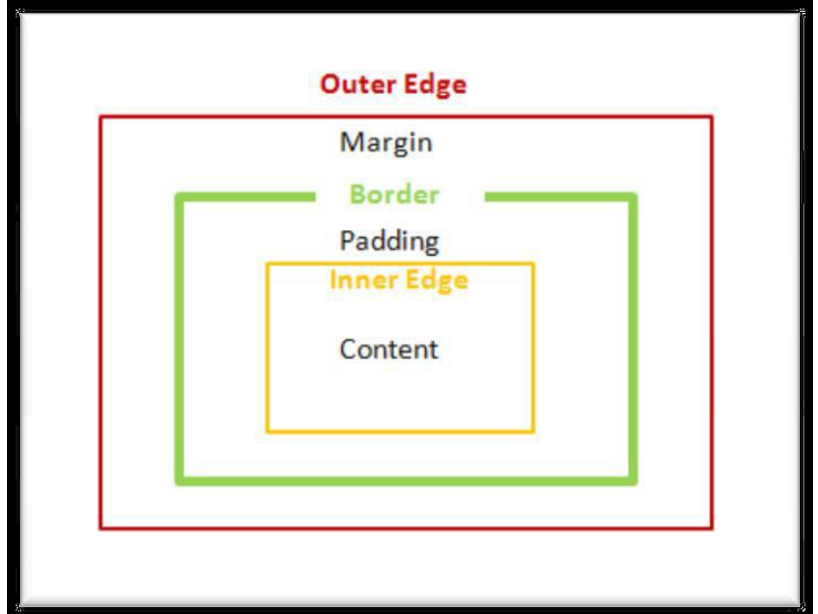
</body>

</html>



2.14 THE BOX MODEL

- On a given web page or a document, all the elements can have borders.
- The borders have various styles, color and width.
- The amount of space between the content of the element and its border is known as *padding*.
- The space between border and adjacent element is known as *margin*.



Borders:

Border-style

- It can be dotted, dashed, double
- Border-top-style
- Border-bottom-style
- Border-left-style
- Border-right-style

Border-width

- It can be thin, medium, thick or any length value
- Border-top-width
- Border-bottom-width
- Border-left-width
- Border-right-width

Border-color

- Border-top-color
- Border-bottom-color
- Border-left-color
- Border-right-color

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

```
<title> Table with border effects </title>
<style type = "text/css">
table
{
        border-width:thick;
        border-top-color:red;
        border-left-color:orange;
        border-bottom-color:violet;
        border-right-color:green;
        border-top-style:dashed;
        border-bottom-style:double;
        border-right-style:dotted;
}
</style>
</head>
<body>
        <table border = "border">
<caption>VTU  </caption>
        <tr>
                <td> VTU Memo </td>
                <td> <img src = "img9.jpg" alt = "cant display"/></td>
        </tr>
</table>
</body>
</html>
```

Margins and Padding:

The margin properties are named margin, which applies to all four sides of an element: margin-left, margin-right, margin-top, and margin-bottom.

The padding properties are named padding, which applies to all four sides: padding-left, padding-right, padding-top, and padding-bottom.

```
<html>
<head>
        <title> Margins and Padding </title>
<style type = "text/css">
p.one
{
        margin:0.1in;
        padding:0.5in;
        background-color:#FF33FF;
        border-style:dotted;
}
        p.two
        {
                margin:0.5in;
                padding:0.1in;
                background-color:#00FF33;
                border-style:dashed;
        }
 p.three
{
        margin:0.3in;
        background-color:#FFFF00;
}
        p.four
        {
                padding:0.3in;
                background-color:#FF9900;
        }
</style>
</head>
<body>
```

```
<p class = "one"> 3MONTHS OF LECTURE! 3WEEKS OF INTERNAL TESTS!<br/>
[margin=0.1in, padding=0.5in]</p>
<p class = "two"> 3DAYS OF STUDY!<br/>
[margin=0.5in, padding=0.1in]</p>
<p class = "three"> 3HRS OF EXAMS!<br/>
[margin=0.3in, no padding, no border]</p>
<p class = "four"> 3MINS OF CORRECTION IS WAT WE CALL<br/>
[no margin, padding=0.3in, no border]</p>
</body>
</html>
```

Margins and Padding

file:///C:/Users/chethan/Desktop/web/h1.html

3MONTHS OF LECTURE! 3WEEKS OF INTERNAL TESTS!
[margin=0.1in, padding=0.5in]

3DAYS OF STUDY!
[margin=0.5in, padding=0.1in]

3HRS OF EXAMS!
[margin=0.3in, no padding, no border]

3MINS OF CORRECTION IS WAT WE CALL
[no margin, padding=0.3in, no border]

2.15 BACKGROUND IMAGES

The background-image property is used to place an image in the background of an element.

```
<html>
<head>
<title>Background Image</title>
<style type = "text/css">
body {background-image:url(bk.jpg);}
p {text-align: justify; color:white;font-size:25pt;}
</style>
</head>
<body>
<p class = "two">Visvesvaraya Technological University (VTU) is a collegiate public state university in Karnataka State, India. It was established on 1 April 1998 by the Government of Karnataka as per VTU Act 1994, to improve the quality of technical education in the state. Apart from a few notable exceptions, VTU
```

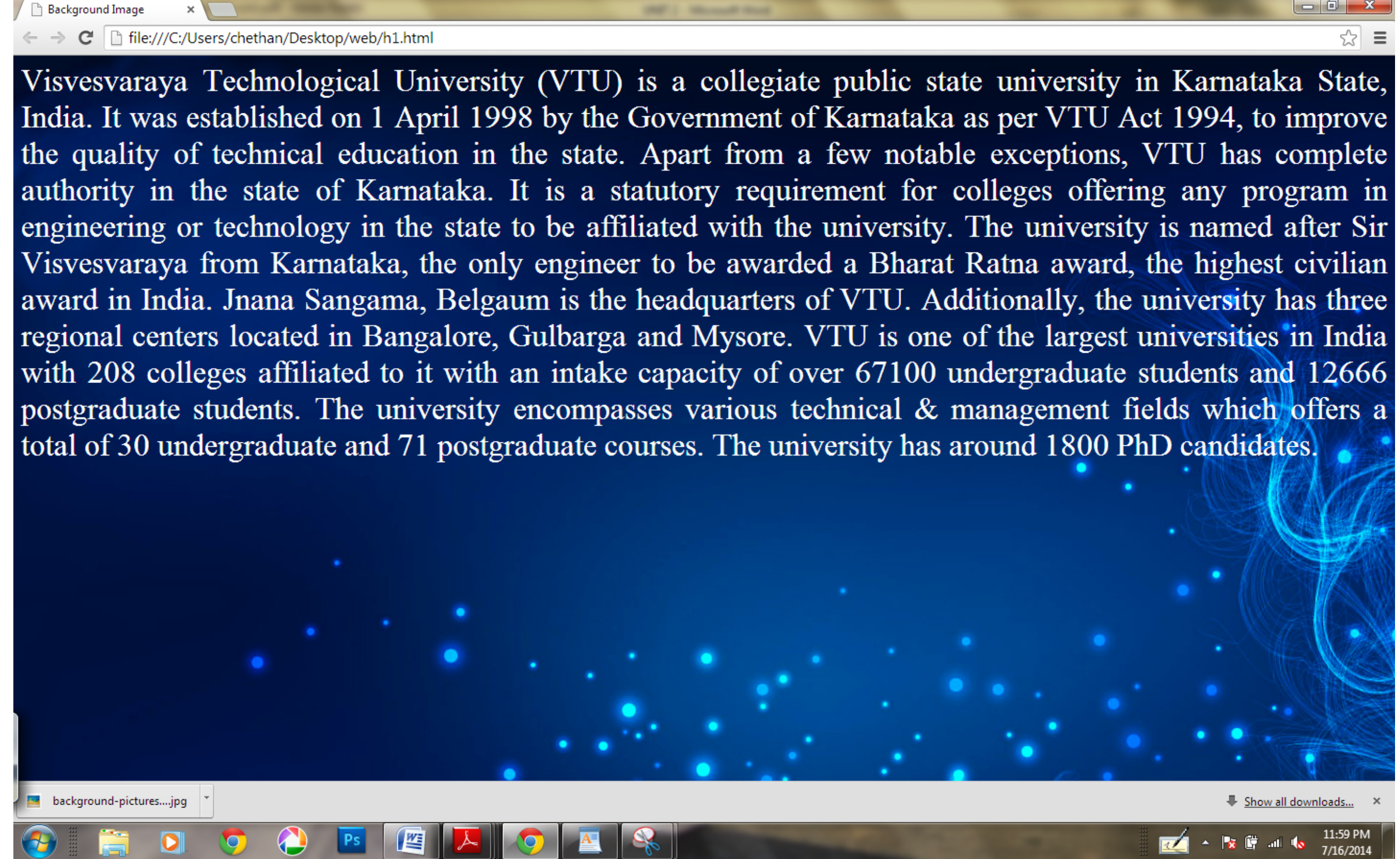
has complete authority in the state of Karnataka. It is a statutory requirement for colleges offering any program in engineering or technology in the state to be affiliated with the university.

The university is named after Sir Visvesvaraya from Karnataka, the only engineer to be awarded a Bharat Ratna award, the highest civilian award in India. Jnana Sangama, Belgaum is the headquarters of VTU. Additionally, the university has three regional centers located in Bangalore, Gulbarga and Mysore.

VTU is one of the largest universities in India with 208 colleges affiliated to it with an intake capacity of over 67100 undergraduate students and 12666 postgraduate students. The university encompasses various technical & management fields which offers a total of 30 undergraduate and 71 postgraduate courses. The university has around 1800 PhD candidates.</p>

</body>

</html>



In some time, the background image is replicated as necessary to fill the area of the element. This replication is called *tiling*. Tiling can be controlled with the background-repeat property, which can take the value repeat (the default), no-repeat, repeat-x, or repeat-y. The no-repeat value specifies that just one copy of the image is to be displayed. The repeat-x value means that the image is to be repeated horizontally; repeat-y means that the image is to be repeated vertically. In addition, the position of a non-repeated background image can be specified with the background-position property, which can take a large number of different values. The keyword values are top, center, bottom, left, and right, all of which can be used in many different combinations.

2.16 THE <span> AND <div> TAGS

In many situations, we want to apply special font properties to less than a whole paragraph of text. The <span> tag is designed for just this purpose.

```
<html>
<head>
        <title>span</title>
<style type = "text/css">
        .spanviolet {font-size:25pt;font-family:'lucida calligraphy';color:violet;}
</style>
</head>
<body>
<p > The university is named after
        <span class = "spanviolet"> Sir Visvesvaraya </span>
, from Karnataka, the only engineer to be awarded a Bharat Ratna award. </p>
</body>
 </html>
```

span

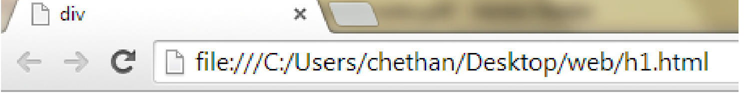
file:///C:/Users/chethan/Desktop/web/h1.html

The university is named after *Sir Visvesvaraya* , from Karnataka, the only engineer to be awarded a Bharat Ratna award.

It is more convenient, however, to be able to apply a style to a section of a document rather than to each paragraph. This can be done with the <div> tag. As with <span>, there is no implied layout for the content of the <div> tag, so its primary use is to specify presentation details for a section or division of a document.

```
<html>
<head>
        <title>div</title>
<style type = "text/css">
        .one {font-size:20pt;font-family:'lucida calligraphy';color:violet;}
        .two {font-size:25pt;font-family:'comic sans ms';color:green;}
</style>
</head>
<body>
        <div class = "one">
                <p>Paragragh 1 under division 1</p>
                <p>Paragragh 2 under division 1</p>
                <p>Paragragh 3 under division 1</p>
```

```
        </div>
        <div class = "two">
                <p>Paragragh 1 under division 2</p>
                <p>Paragragh 2 under division 2</p>
                <p>Paragragh 3 under division 2</p>
        </div>
</body>
</html>
```



Paragragh 1 under division 1

Paragragh 2 under division 1

Paragragh 3 under division 1

Paragragh 1 under division 2

Paragragh 2 under division 2

Paragragh 3 under division 2

2.17 CONFLICT RESOLUTION

- Sometimes on a web page, there can be two different values for the same property on the same element leading to conflict.
- h3 {color: blue;}
  body h3 {color: red;}
- The browser has to resolve this conflict.
- There can be one or more type of conflict: i.e. when style sheets at 2 or more levels specify different value for same property on some element.
- This conflict is resolved by providing priority to the different levels of style sheets.
- The inline level gets the highest priority over the document level.
- The document level gets the higher priority over the external level
- But the browser must be able to resolve the conflict in the first example using same technique.

- There can be several different origins of the specification of property values.
- One of the value may come from a style sheet created by the author or it can be specified by the user using the options provided by the browser.
- The property values with different origin have different precedence.
- The precedence can also be set for a property by marking it as important.
- p.special {font-style: italic !important; font-size: 14}
- This means that font-style:italic is important [this is known as weight of specification]
- The process of conflict resolution is a multi-stage sorting process.
- The first step is to gather information about levels of style sheet.
- Next, all the origins and weights are sorted. The following rules are considered:
  1. Important declarations with user origin
  2. Important declarations with author origin
  3. Normal declarations with author origin
  4. Normal declarations with user origin
  5. Any declarations with browser (or other user agent) origin
- If there are other conflicts even after sorting, the next step is sorting by specificity. Rules are:
  1. id selectors
  2. Class and pseudo class selectors
  3. Contextual selectors (more element type names means that they are more specific)
  4. Universal selectors
- If there still conflicts, they are resolved by giving precedence to most recently seen specification.

Question paper questions :

a) Write an XHTML document to describe an ordered list of your five favorite movies. Each element of the list must have a nested list of at least two actors in your favorite movies. (05 M)

b) With examples, explain a style class selector. (05 M)

c) Write an XHTML document that has six short paragraphs of text. Define three different paragraph styles p1, p2 and p3. The p1 style must use left and right margins of 20 pixels, a background colour of yellow, and a foreground color of blue . The p2 style must use font size of 18 points, font name 'Arial' and a font style in italic form. The p3 style must use a text indent of 1 centimeter , a background color of green, and a foreground color of white. The 1st and the 4th paragraph must use p1, the 2nd and 5th must use p2 and the 3rd and 6th must use p3. (10 M)

d) Explain the following with respect to table creation in XHTML documents: Align and valign attributes tr, th and td attributes Rowspan and Colspan attributes Cell padding and Cell spacing attributes (10 M)

e) Create XHTML document to describe a table with the following contents: The columns of the table must have the headings pine, maple, Oak and fir. The rown must have the labels average height, average width, typical lifespan, and leaf type. Fill the data cells with some values. (10 M)

f) Explain the syntactic differences between HTML and XHTML (05 M)

g) What tag and attribute are used to describe a link? Discuss about it. (04 M)

h) Explain all controls that are created with the <input> tag with examples, which are used for text collection. (08 M)

i) Explain the XHTML tags used for lists in documents (08 M)

j) Write an XHTML program to create a link within a document (04 M)

k) Create XHTML document that defines a table with 5 rows and 5 columns. The first row should contain country name, gold, silver, bronze (all three indicating the type of medals) and total in each column respectively. Fill in the information details in the table with appropriate values. After filling the details, set red color to the background for the first row, blue for the second, yellow for the third, purple for the fourth and green for the fifth row. Use of align and valign attributes for this table has to be made at the appropriate places (10 M)

l) How lists are handled in XHTML? Design an XHTML code to illustrate nested lists (10 M)

m) Design an XHTML code to construct a simple class time table to illustrate table handling (10 M)

n) Write a XHTML program to create a table with 2 levels of column label an overall label, meals and 3 secondary labels of much breakfast, lunch and dinner. There must be 2 level of row labels: an overall label, food and 4 secondary labels, bread, main cause, vegetables and dessert. The cell of table must contain a number if grams for each category of the food.(12 M)

o) Explain the different levels of style sheets available in CSS (04 M)

p) Create XHTML document that contain student information via name, USN, subject 1, subject 2 and subject 3. Insert values for each student in five rows. Also row background of each student should be in the different color (08 M)

q) Explain the following tags

i. <select> ii. <Frame> ii. <text area> (08 M)

r) What are selector forms? Explain the different levels of selector forms available in CSS with example (08 M)

s) Write document level style sheet to illustrate pseudo classes. Discuss the conflict resolution in CSS(08 M)

t) Create an XHTML document that includes atleast 2 images and enough text to precede the images, flow around them (one on left and one on right) and continue after the last image (Note : Use CSS tags) (04 M)

u) Write an XHTML document to describe an ordered list of four states. Each element of the list must have an unordered list of at least two cities in the state. (05 M)

v) Explain the following , with respect to table creation in XHTML documents.

i. <table> ii. tr, th and td attributes iii. rowspan and colspan attributes

iv. align and valign attributes v. cell padding and cell spacing (10 M)

w) Create XHTML document that has two frames. The left frame displays contents.html and the right frame displays cars.html where the second frame is a target of link from the first frame. [Note: contents.html is a list of links to the cars description.] (05 M)

x) Create , test and validate a XHTML document that has a form with

i. Three text boxes to collect user name and address.

ii. Tables with the headings product name , price and quantity and the values are

100—watts light bulb, $2.39 , 4

200—watts light bulb, $4.29 , 8

100—watts long life light bulbs, $3.95 , 4

200—watts long life light bulbs, $7.49 , 8

iii. A collection of 4 radio buttons that are labeled as Visa Master card Discover Check

iv. A submit and a reset button (10 M)

Course notes

1 What is a Programming Language? 5
Key Points 5

2 Server-side vs. Client-side 7
Key Points 7

3 About JavaScript 10
Key Points 10

4 A Tour of JavaScript 13
Key Points 13
Project 13

5 Objects, Properties and Methods 18
Key Points 18

6 Assigning Values to Properties 21
Key Points 21
Project 22

7 About Comments 25
Key Points 25
Project 26

8 Hiding Scripts from Older Browsers 28
Key Points 28
Project 29

9 Automatically Redirecting the User 31
Key Points 31
Project 31

10 Alert, Prompt and Confirm 33
Key Points 33
Project 34

11 Variables and Operators 35
Key Points 35
Project 38

12 Comparisons 40
Key Points 40
Project 41

13 Conditionals 42
Key Points 42
Project 45
Project 2 46

14 Looping ... 48
Key Points ... 48
Project ... 50

15 Arrays ... 53
Key points ... 53
Project ... 55

16 Associative & Objective Arrays ... 57
Key Points ... 57
Project ... 58

17 Two Dimensional Arrays ... 59
Key Points ... 59
Project ... 60

18 String Manipulation ... 61
Key Points ... 61
Project ... 65

19 Using Functions ... 66
Key Points ... 66
Project ... 69

20 Logical Operators ... 71
Key Points ... 71
Project ... 74

21 Using Event Handlers ... 75
Key Points ... 75
Project ... 77

22 Working with Images ... 79
Key Points ... 79
Project ... 80

23 Simple Image Rollovers ... 81
Key Points ... 81
Project ... 83

24 Object Instantiation and Better Rollovers ... 85
Key Points ... 85
Project ... 86

25 Working with Browser Windows ... 88
Key Points ... 88
Project ... 90

26 Positioning Browser Windows ... 91
Key Points ... 91
Project ... 92

27 Focus and Blur....93
Key Points 93
Project.... 94

28 Dynamically Created Content....95
Key Points 95
Project.... 95

29 Working with Multiple Windows....97
Key Points 97
Project.... 98

30 Using an External Script File....99
Key Points 99
Project.... 100

31 Javascript and Forms....101
Key Points 101
Project.... 103

32 Form Methods and Event Handlers....105
Key Points 105
Project.... 106

33 JavaScript and Maths....108
Key Points 108
Project.... 109

34 Object Variables – A Refresher....111
Key Points 111
Project.... 112

35 Actions From Menu Items....113
Key Points 113
Project.... 114

36 Requiring Form Values or Selections....116
Key Points 116
Project.... 118

37 Working with Dates....121
Key Points 121
Project.... 122

38 Retrieving Information from Date Objects....123
Key Points 123
Project.... 124

39 Creating a JavaScript Clock....126
Key Points 126
Project.... 128

1 What is a Programming Language?

Key Points

- A programming language is a set of codes that we can use to give a computer instructions to follow.

- Popular and well-known programming languages include Java, C++, COBOL, BASIC, LISP and more. Most popular programming languages consist of words and phrases that are similar in form to the English language.

- A well-written program will be easily readable by anyone with a little programming experience, regardless of whether they have any direct experience of the language in question. This is because modern programming languages share a large number of common concepts. In particular, they all have a notion of **variables**, **arrays**, **loops**, **conditionals**, and **functions**. We will meet these concepts again in more depth later in the course.

- Traditionally, programming languages have been used to write (for the most part) "stand-alone" applications. Things like Microsoft Word, Mozilla Firefox and Lotus Notes are all examples of such applications. Once installed on a PC, these applications run without necessarily requiring any other software to be installed alongside them.

- Web Applications differ from these traditional applications in many respects, but the most striking is that they all run **inside your web browser**. Examples of popular web applications are things like Google, Hotmail, Flickr, GMail and any of the vast array of "weblogging" systems.

- These applications are also written using programming languages, but as a rule they are built using multiple, interdependent technologies. These technologies are easily (though not completely) broken down into two categories: **server-side** and **client-side**.

2 Server-side vs. Client-side

Key Points

- The World Wide Web is built on a number of different technologies.

- For most users, the web starts and ends with their choice of **web browser**. The browser is said to define the **client-side** of the web, with the browser, the computer it is running on, and the user surfing the web being collectively referred to as **the client**.

- Consider a client who has decided to visit the web site at **www.google.com**. The first thing that happens is that the client will make a request to Google's **web server** for the default page of that web site.

- The **web server** is an application running on a computer owned by Google. Like the client, the server application and the computer on which it runs define the **server-side** of the web, and are collectively referred to as **the server**.

- When the server receives the request from the client for a particular page, its job is to retrieve the page from the computer's files and **serve** it back to the client. In many cases, this operation is a very simple procedure involving little or no work on the part of the server.

- However, using a programming language like PHP, Perl or Java, we can cause the server to either modify the page it finds before it passes it back to the client, or even to generate the page entirely from scratch. This is referred to as a **server-side** application. The page passed back to the client looks (to the client) exactly the same as any other page that has not been modified.

- An example of a **server-side** application might be to insert the current date and time into a page. This would mean that each time the page was requested (say, by using the browser's refresh button), a new time value would be added to the page.
- Once the client has received the page from the server, it displays the page and waits for the user to request another page. As soon as the page reaches this state, it has moved beyond the control of the server. No **server-side** application can now alter the contents of the page without the client having to make another trip back to the server to get a new (and possibly updated) copy of the page.
- However, all modern browsers allow for the running of **client-side** applications. These are small applications which are **embedded** within the HTML code of the page itself.
- **Server-side** applications ignore any **client-side** applications that they find while modifying pages to send to the client, so in general the two types of application cannot easily "talk" to each other.
- However, once the client has received a **client-side** application, it can begin to modify the page **dynamically**, without the need to go back to the server.
- An example of a **client-side** application might be a clock on a web page that updated every second.
- An unfortunate side effect of **client-side** applications is that all the code must be sent to the client for running, which means that the application's inner workings are available for anyone to see. This makes it impractical for checking passwords, or doing anything else that could cause confidential information to be released into the wild.
- In addition, all modern web browsers afford the user the opportunity to switch off **client-side** applications altogether. On top of this, the way the same **client-side** application is run will vary from browser type to browser type.
- Despite these drawbacks, **client-side** applications (or **scripts**, as they are better known due to their general brevity) remain the best way to provide web users with a rich environment when developing web applications.

- In short, the two technologies each have their strengths and weaknesses:
- **Client-side** scripts allow the developer to alter pages dynamically, and to respond to user actions immediately rather than having to wait for the server to create a new version of the page. However, there are security and **cross-browser compatibility** issues to be aware of, and these are often non-trivial.
- **Server-side** applications allow the developer to keep her code secure and secret, thus allowing for more powerful applications to be created. In addition, since the server running the code is always a known quantity, applications that run successfully in one browser will run successfully in all browsers. However, despite all this power, there is no direct way for a **server-side** application to alter a page without having to force the **client-side** to load another page. This makes it completely impractical for things like drop-down menus, pre-submission form checking, timers, warning alerts and so forth.

3 About JavaScript

Key Points

- **JavaScript** is an interpreted, client-side, event-based, object-oriented scripting language that you can use to add dynamic interactivity to your web pages.

- **JavaScript** scripts are written in plain text, like HTML, XML, Java, PHP and just about any other modern computer code. In this code, we will use **Windows NotePad** to create and edit our **JavaScript** code, but there are a large number of alternatives available. **NotePad** is chosen to demonstrate **JavaScript's** immediacy and simplicity.

- You can use **JavaScript** to achieve any of the following:

  - Create special effects with images that give the impression that a button is either highlighted or depressed whenever the mouse pointer is hovered over it.

  - Validate information that users enter into your web forms

  - Open pages in new windows, and customise the appearance of those new windows.

  - Detect the capabilities of the user's browser and alter your page's content appropriately.

  - Create custom pages "on the fly" without the need for a server-side language like PHP.

  - And much more…

- **JavaScript** is *not* **Java**, though if you come from a Java background, you will notice that both languages look similar when written. **Java** is a full featured and comprehensive programming language similar to C or C++, and although **JavaScript** can interact with **Java** web applications, the two should not be confused.

- Different **web browsers** will run your **JavaScript** in different, sometimes incompatible ways. In order to work around this, it is often necessary to use **JavaScript** itself to detect the capabilities of the browser in which it finds itself, and alter its operation depending on the result.

- To revisit the original definition in this chapter, note the following points:

    - **Interpreted** refers to the fact that **JavaScript** code is executed (acted on) as it is loaded into the browser. This is a change of pace from **compiled** languages like Java, which check your program thoroughly before running a single line of code, and can have many implications that can catch you out if you are from a non-interpreted programming background.

    - **Client-side** has been defined already in the previous chapter.

    - **Event-based** refers to **JavaScript's** ability to run certain bits of code only when a specified **event** occurs. An event could be the page being loaded, a form being submitted, a link being clicked, or an image being pointed at by a mouse pointer.

    - **Object-oriented** signals that **JavaScript's** power to exert control over an HTML page is based on manipulating **objects** within that page. If you are familiar with **object-oriented programming**, you will be aware of some of the power that this can bring to the coding environment.

- One final note: While **JavaScript** is a programming language, HTML (the language of the World Wide Web) is *not*. HTML is a **Markup Language**, which means that it can be used to mark areas of a document as having special characteristics like headers, paragraphs, images, forms and so on, but it cannot perform any logical processing on its own. So while **JavaScript** is often written alongside HTML, the rules of one do not necessarily have any bearing on the other.

4 A Tour of JavaScript

Key Points

- Let's start with a quick tour of the major features of **JavaScript**. This chapter is intended to be a showcase of what **JavaScript** can do, not an in depth investigation into the deeper concepts, so don't worry too much if you get lost or don't understand the code you're typing in!

Project

- Our **JavaScript** is all going to be written using **NotePad**. Open **NotePad** and save the resulting empty document in your user drive as **`chapter_4.html`**.

- Begin by creating a basic HTML page in your blank document. It doesn't have to be anything fancy – the following will be more than sufficient:

```
<html>
<head>
    <title>Chapter 4: A Tour of ↲
  JavaScript</title>
</head>


<body>

<h1>A Tour of JavaScript</h1>

</body>
</html>
```

- As a convention, when the notes intend that you should enter code all on one line, they will use an arrow as above ↲ *to indicate that you should not take a new line at that point. With HTML, this is rarely important, but with **JavaScript**, a new line in the wrong place can stop your code from working.*

- Save your new webpage, and view it in your web browser. For the moment, use **Internet Explorer** to view this page. To do this, find your saved file on your user drive, and double-click on it. This will open the file in **Internet Explorer** by default, and let you see the header you've just created.

- So far, we haven't done anything beyond the scope of HTML. Let's add some **JavaScript** to the page.

- There are (generally speaking) **three** places in a web page where we can add **JavaScript**. The first of these is between a new set of HTML tags. These **`script`** tags take the following form:

```
<script language="JavaScript" ↲
  type="text/JavaScript">

… code …

</script>
```

- The **`script`** element above can be placed virtually anywhere you could place any element in an HTML page – in other words, in either the **`head`** element or the **`body`** element. It is most commonly placed in the former, though this is usually so that all your code can be easily found on the page.

- Note too that there is no arbitrary limit on the number of **`script`** elements that can be contained in an HTML page. There is nothing stopping you from having a hundred of these dotted around your pages, except perhaps prudence.

- Let's add our opening and closing script tags to the head element of the page, like so:

```
<html>
<head>
    <title> … </title>
    <script language="JavaScript" ↲
  type="text/JavaScript">

    </script>
</head>

…
```

- Save the file, and then try refreshing your page in the browser window. Note that nothing has happened. This is what we expected – all we have done so far is to set up an area of the page to *hold* our **JavaScript**.

- Go back to **NotePad** and enter the following text between the opening and closing `script` tags:

```
window.alert("Hello world!");
```

- Save your changes, and again refresh your page in the browser window. Welcome to the world of **JavaScript**!

- Go back to notepad and remove the `window.alert` line you just added. Now add the following, slightly more complex code:

```
if  ( confirm("Go to Google?") )  {
    document.location  =  ↲
"http://www.google.com/";
}
```

- Again, save your changes and refresh the page. For those with an eye to future chapters, this is an example of a **conditional** statement, where we ask **JavaScript** to check the **condition** of something (in this case, our response to a question) and then to alter its behaviour based on what it finds.

- Now, both of these bits of **JavaScript** have run uncontrollably when the page has loaded into the browser. In most cases, we will want to have more control over when our **JavaScript** does what we ask it to.

- This control is the domain of **events**. In a browser, every element of an HTML document has associated with it a number of **events** that can happen to it. Links can be `click`ed, forms can be `submit`ted, pages can be `load`ed and so on.

- Modify the previous lines of JavaScript in your `script` element to match the following:

```
function  go_to_google() {
  if  ( confirm("Go to Google?") )  {
    document.location  =  ↲
"http://www.google.com/";
  }
}
```

- Be careful with your brackets here!

- Save and refresh, and note that nothing happens this time. This is because we have enclosed the previous action (popping up a question and acting on the response) within a **function**. A **function** is a block of code that is given a name – in this case, the name is **`go_to_google()`** – and is only run when that name is "called". It can be useful to think of **functions** as magic spells that can be invoked when their name is said.

- To invoke this spell, we need to choose an element on the page to trigger it. A natural candidate is a link element, so add the following HTML to the **`body`** section of your page:

```
<p>A quick <a href="#">test</a>.</p>
```

- The # link is a common HTML trick that allows us to create a "link to nowhere".

- Save and refresh, and check that the link appears on the page, and that it goes nowhere when clicked.

- Now, we want to have our page ask us if we want to "Go to Google?" when we click on that link. Here's how

- Take the link element, and modify it as follows:

```
<a href="#" onclick="go_to_google();">test</a>
```

- Save and refresh, and then click on the link. This is an example of an **event handler**. When the link is clicked (**`onclick`**), our browser says the "magic words" **`go_to_google()`**, and our **function** is invoked.

- For our final trick, add the following code to the **`body`** section of the page, after the paragraph containing the link:

```
<body>
…
<script language="JavaScript" ↵
  type="text/JavaScript">

document.write("<h2>Here's another ↵
  header!</h2>");

</script>
```

- Note that the line of code should be all on one line!

- Save the page and refresh the browser. Note that we now have a new line of text on the page – another header! We've used **JavaScript** to create HTML and tell the browser to display it appropriately. In this example, **JavaScript** has done nothing that we couldn't have done with a line of HTML, but in future chapters we will see how we can use this to write the current date and more.

5 Objects, Properties and Methods

Key Points

- Generally speaking, **objects** are “things”. For example, a piano is an **object**.

- **Properties** are terms that can describe and define a particular **object**. Our piano, for example, has a colour, a weight, a height, pedals, a keyboard and a lid.

- Note from the above that an object’s properties *can be properties themselves*. So we have the case where a piano lid is a property of the piano, but is also an object in its own right, with its own set of properties – for example, the lid has a colour, a length, and even a *state* of either open or closed.

- If **objects** are the nouns of a programming language and **properties** are the adjectives, then **methods** are the verbs. **Methods** are actions that can be performed on (or by) a particular object. To continue our piano example, you could play a piano, open its lid, or press the sustain pedal.

- Many programming languages have similar ways of referring to objects and their properties or methods. In general, they are *hierarchical*, and an object’s relationship with its properties and methods, as well as with other objects, can often be easily seen from the programming notation.

- In JavaScript, we use a “dot notation” to represent objects and their properties and methods. For example, we would refer to our piano’s colour in the following way:

```
piano.colour;
```

- If we wanted to instruct JavaScript to play the piano, we could write something as simple as:

  ```
  piano.play();
  ```

- A clear example of object hierarchy could be seen if we decided to open the lid of the piano:

  ```
  piano.lid.open();
  ```

- Or even more so if we wanted to press the sustain pedal of the piano:

  ```
  piano.pedals.sustain.press();
  ```

- Note that in some of the examples above, we have brackets () after each set of words, and in some we don't. This has to do with making sure that JavaScript can understand what we say.

- JavaScript works with objects throughout its existence in a web browser. All HTML elements on a page can be described as objects, properties or methods. We have already seen a few of these objects in our previous introductory chapter:

  ```
  document.write(…);
  document.location;
  ```

- In these examples, **`document`** is an object, while **`write`** is a method and **`location`** is a property.

- In these lines, we see a clue about the use of brackets in these statements. We use brackets to signify to JavaScript that we are talking about an object's **method**, and not a property of the same name.

- Brackets also allow us to pass certain extra information to an object's method. In the above example, to write the text "Hello world!" to a web page document, we would write the following JavaScript:

  ```
  document.write("Hello World");
  ```

- Each method can do different things depending on what is put in the brackets (or "passed to the method as an argument", to use the technical term). Indeed, many methods can take multiple "arguments" to modify its behaviour. Multiple arguments are separated by a comma (,).

- A JavaScript instruction like those shown here is referred to as a JavaScript **statement**. All statements should end in a single semi-colon (;). JavaScript will often ignore missed semi-colons at the end of lines, and insert the semi-colon for you. However, this can cause some unexpected results. Consider the following:

```
document.write("<h1>
                   Hello World!
                </h1>");
```

- In many other languages, this would be acceptable. However, JavaScript will often interpret something like this as the following:

```
document.write("<h1>;
Hello World!;
</h1>");
```

- This interpretation will generate an error, as JavaScript will complain if you end a statement without ensuring that any terms between quotes have matching pairs of quotes. In this example, the first line's "statement" is cut short, and JavaScript will fall over.

- For this reason, it is recommended that all your statements should end with semi-colons.

6 Assigning Values to Properties

Key Points

- While objects and methods allow us to **do** things on a page, such as alter the content or pop up dialogue boxes to interact with the user, in many cases we will want to alter the value of one of an object's properties directly. These cases are akin to painting our piano green.

- Given our discussion on methods so far, we might expect to be able to alter our object's properties by using a method – for example, the following would seem logical:

  ```
  piano.paint("green");
  ```

- In many cases, that is exactly what we will do. However, there are two drawbacks here. The first is that, within this course, the majority of objects that we discover are built into and defined by our browser. If we rely on using a method to alter an object's property, we are also relying on the fact that the method exists in the first place.

- A much more direct way to solve this problem is to access the object's properties directly. For example:

  ```
  piano.colour  =  "green";
  ```

- Here we are no longer using a method to perform an action, we are using what is known as an **operator**. In this case, the operator has the symbol "=", and is known as the **assignment operator**.

- Within JavaScript, we can use this operator to great effectiveness. For example, we could alter the title element of a document (the text that is displayed in the top bar of the browser's window) dynamically. This could be done when a user clicked on a part of the page using an event handler (more later on this), or could be set to automatically update each minute to show the current time in the page title. The code we would use for this task is simple:

```
document.title  =  "a new title";
```

- There are many assignment operators in JavaScript. Some of the more common are shown in the table below:

Assignment	Function
x = y	Sets the value of x to y
x += y	Sets the value of x to x+y
x -= y	Sets the value of x to x-y
x *=y	Sets the value of x to x times y
x /=y	Sets the value of x to x divided by y

- Not all assignment operators work with all types of values. But the addition assignment operator works with both numbers and text. When dealing with numbers, the result will be the sum of the two numbers. When dealing with text (technically called **strings**), the result will be the **concatenation** of the two strings:

```
document.title += "!";
```

  will cause the symbol "!" to be appended to the end of the current document title.

Project

- Open your previous project file, and save it under the name **`chapter_6.html`**.

- Remove any existing JavaScript from your script tags, but leave the tags in place ready for some new JavaScript.

- Use your text editor to change the value of the title element of the page as follows, then load your page into a browser and view the result:

```
<title>With a little help from</title>
```

- Now, add a statement to our script element to add the following text to the end of the current title:

```
"JavaScript for Beginners!";
```

- Reload the page in your browser and note the title bar of the window.

- If the display looks odd, consider your use of spaces…

- All we have so far is an example that does nothing more than HTML could manage. Let's introduce a new method of the **window** object to help us to add a little more dynamism and interaction to the script. Change the value of the title tag as follows:

```
<title>Chapter 6: Assigning Values to
Properties</title>
```

- Now, remove your previous JavaScript statement and insert the following:

```
document.title = ↲
   window.prompt("Your title?", "");
```

- Reload your page and consider the result.

- We have come across the **window** object before. Our demonstration of the **alert** method in chapter 4 could have been more properly written as:

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

  In many cases, we can omit certain parts of our object/property/method hierarchy when writing our code. We will discuss this again later.

- To understand what is going on with our **prompt** method, we can write down a method **prototype**. This is a way of describing a method's arguments in such a way that their effect on the method is more self explanatory. A prototype for the prompt method of the window object might look like the following:

  ```
  window.prompt( message, default_response );
  ```

- So, we can see that the first argument defines the text that appears as the question in the prompt dialogue box. The second argument is a little less clear. Try your code with different values and see what difference your changes make.

- Finally, we note that this prompt method somehow takes the information typed into the box and passes it to our JavaScript assignment. Say someone typed "Hello World" into the box. It would have been as if our assignment had actually been:

  ```
  document.title = "Hello World";
  ```

- When this sort of passing of values occurs, it is said that the method has **returned** the value passed. In this case, we would say that "the prompt method has returned the value 'Hello World'", or that "the return value of the prompt method was 'Hello World'".

- Return values will become very important when we deal with event handlers later on.

7 About Comments

Key Points

- Repeat after me : Comments are important. Comments are **important**. **Comments are important**.
- Adding comments to your code is always good practice. As the complexity of your scripts grows, comments help you (and others) understand their structure when you come to view the code at a later date.
- A lot of code created quickly is said to be “write only” code, as it suffers from an inherent lack of structure or commenting. Debugging such code, or reusing it months later, becomes maddeningly impossible as you try to remember what a certain line was supposed to do, or why using certain values seems to stop your code from working.
- Comments are completely ignored by JavaScript and have no effect on the speed at which your scripts run, provided they are properly formed.
- Comments *can* slow the loading of your page, however – many coders keep a “development” copy of their code fully commented for editing, and remove all comments from their code when they finally publish it.
- There are two types of comment in JavaScript – single line comments, and multi-line comments.
- Single line comments begin with two forward-slash characters (//), and end with a new line:

```
// this is a comment

alert(“hello”); // so is this
```

- Single line comments in JavaScript can also use the HTML comment format that you may be familiar with:

```
<!-- this is a comment

alert("hello");
```

- Note two things: firstly, this use of the HTML comment format **does not** require a closing **-->** tag. Secondly, this is only a one line comment, unlike its use in HTML, which comments all lines until the closing comment tag.

- You can add multiple-line comments by enclosing the comment block between /* and */. For example:

```
/* all of this text is going to be
ignored by JavaScript. This allows us to
write larger comments without worrying about
having to individually "comment out" each
line */

alert("Hello World");

/* a one line, "mult-line" comment */
```

- As well as adding narrative to your script, you can use comments to remove code from your pages without having to delete the code. For example:

```
// this was the old message
// alert("Hello World");
// and this is the new message
alert("Hello everyone!");
```

- This can be very useful if you are trying to track down an error in your code – you can "comment out" each suspect line in turn until you manage to get your code working again.

Project

- Open your previous project file, and save it under the name **chapter_7.html**.

- Add the single line comment

```
This is my first comment
```

to the beginning of your script.

- Add a multi-line comment to your script, replacing your previous single line comment. The multi-line comment should describe what your script does at present.

8 Hiding Scripts from Older Browsers

Key Points

- Very old browsers don't understand JavaScript. There are very few such browsers in use today, but two factors force us to continue to consider environments that may not be able to cope with our JavaScript code.

- Firstly, all modern browsers allow users to control whether JavaScript code will be run. In many cases, users will not have any say over their company policy, and may not even know that their work machine has had JavaScript disabled.

- Secondly, not all of your visitors will be using browsers that can make any use of JavaScript. Braille displays, screen readers and other non-visual browsers have little use for many JavaScript tricks. In addition, search engines like Google will ignore any JavaScript you use on your pages, potentially hiding important content and causing your pages to remain un-indexed.

- Browsers that don't support JavaScript are supposed to ignore anything between the opening and closing script tags. However, many break this rule and will attempt to render your code as HTML, with potentially embarrassing consequences.

- However, we can use the fact that **<!--** denotes a single line comment in JavaScript but a multi-line comment in HTML to ensure that our code is seen by a JavaScript-savvy browser, but ignored as commented-out HTML by anything else:

```
<script>
<!-- hide  from older browsers

… your code

// stop hiding code -->
</script>
```

- This prevents older browsers from displaying the code, but what if we want to replace this with some comment. For example, let's say we had a bit of code that displayed the time of day and greeted our user by name. Without JavaScript and using the method above, there would simply be a blank on the page where the greeting should have been.

- We can use the <noscript> tag to cause our code to "fail gracefully" where JavaScript has been disabled or is unavailable. The contents of this element will be ignored where JavaScript is understood, and displayed anywhere else. For example:

```
<noscript>
    <h1>Welcome to our site!</h1>
</noscript>

<script>
<!-- hide  from older browsers

… code to customise header

// stop hiding code -->
</script>
```

Project

- Open your previous project file, and save it under the name **chapter_8.html**.

- Add two lines to your code to ensure that it will not confuse older browsers or browsers where the user has disabled JavaScript.

- Add a noscript element to explain what your JavaScript does. It is generally considered “bad form” to instruct your user to “upgrade to a better browser”, as this can insult many people who use assistive devices – consider this form of advice to be similar to the advice that tells a blind person “to get some glasses”.

- Instead where possible you should use the noscript element to provide content that adequately replaces the scripted content with a suitable replacement. For example, if you use your JavaScript to build a navigation panel on your page, the noscript element should contain a plain HTML list that does the same job.

9 Automatically Redirecting the User

Key Points

- We have already briefly seen the use of browser redirection in chapter 4.

- To formulate the idea more completely, in order to redirect the user to a different page, you set the **location** property of the **document** objects.

- As we saw in chapter 6, we can use the assignment operator here. For example:

```
document.location = "http://www.bbc.co.uk/";
document.location = "chapter_4.html";
```

Project

- Open your previous project file, and save it under the name **`chapter_9_redirect.html`**.

- Save another copy of the file, this time called **`chapter_9.html`**.

- Make sure both files are saved in the same folder, and that you have **`chapter_9.html`** open in your editor.

- Remove all script from between the script tags, except for your browser hiding lines. Make sure that the script tags are still in the head section of the page.

- Now, add a single statement to this script that will automatically redirect the user to the page **`chapter_9_redirect.html`** as soon as the page is loaded into a browser.

- Finally, add a header tag to the body section of the page containing the text “You can’t see me!”.
- Close this page, don’t check it in a browser yet, and open the page **`chapter_9_redirect.html`** in your editor.
- Remove all JavaScript from this page (including your script tags) and ensure that only HTML remains on the page.
- Add a header tag to the body section of the page containing the text “But you can see ME!”.
- Save this page, and load the page **`chapter_9.html`** into your browser.
- Experiment with various positions for the script tags on **`chapter_9.html`** to see if you can make the header appear before the redirection.

10 Alert, Prompt and Confirm

Key Points

- So far, we have seen brief examples of **alert**, **prompt** and **confirm** dialogue boxes to request a response from the user, and to pause all code in the background until the request is satisfied.

- All of these boxes are the result of methods of the **window** object. This object is the highest level object that JavaScript can deal with in a browser. As such, all other objects on a page (with a few exceptions) are actually properties of the window object.

- Because of this ubiquity, its presence is assumed even if it is omitted. Thus, where we technically *should* write:

  ```
  window.document.write("…");
  ```

  it is equally valid to write:

  ```
  document.write("…");
  ```

  as we have been doing.

- Similarly, instead of writing:

  ```
  window.alert("…");
  ```

  we can happily write:

  ```
  alert("…");
  ```

- The prototypes of the three methods are:

```
window.alert( message );
window.confirm( message );
window.prompt( message, default_response );
```

- Alert will always return a value of “true” when it is cleared by clicking “ok”.
- Confirm will return either “true” or “false” depending on the response chosen to clear the box.
- Prompt will return either the value typed in, “null” if nothing is typed in, and “false” if the box is cancelled.

Project

- Open your previous project file, and save it under the name **chapter_10.html**.
- Clear the previous redirection code, and ensure that the script tags have been returned to the head section of the document.
- Add a new statement to the script on the page that will display the following message before the rest of the page is shown:

  Welcome to my website! Click OK to continue.

- Check your page in your browser.
- We will use **alert**, **confirm**, and **prompt** throughout this course. Take a moment to try each of them in turn on this page, each time stopping to review your changes.
- Use the write method of the **document** object to check the return values of each method. For example:

```
document.write(alert(“hello world”));
```

  Make sure that you place this particular snippet of code in script tags within the body area of the page, as we are generating text output to be rendered by the browser. Also, note the use (or not) of quotes here. More next chapter!

11 Variables and Operators

Key Points

- We have been introduced to the concepts of **objects** and their various **properties** and **methods**. These inter-related concepts allow any web page to be broken down into little snippets of information or **data**, which can then be accessed by JavaScript and, in many cases, changed.

- However, what if we want to create our own storage space for information that doesn't necessarily have a page-based counterpart? For example, what if we wanted to store the previous value of a document's title property before changing it, so it could be retrieved later, or if we wished to store the date time that the page was loaded into the browser for reproduction in several places on the page, and didn't want to have to recalculate the time on each occasion?

- **Variables** are named containers for **values** within JavaScript. They are similar to object properties in many ways, but differ importantly:

- In a practical sense, variables have no "parent" object with which they are associated.

- Variables can be created ("declared") by you as a developer, and can be given any arbitrary name (within certain rules) – object properties, however, are restricted by the definition of the parent object. It would make no sense, for example, for our piano object in the previous chapters to have a propeller property!

- Variable name rules are straightforward – no spaces, names must start with a letter. Examples of valid variable names are:

  BrowserName
  page_name
  Message1
  MESSAGE1

- In many browsers, JavaScript is **case-sensitive**, which means that the last two variables in the example above are **distinct variables**. It is a good idea to pick a particular naming style for your variables, and to stick to it within your projects.

- At the simplest level, variables can store three different types of value:

- **Numbers**
  e.g. 1.3324, 3.14159, 100000, -8 etc.

- **Strings**
  e.g. “JavaScript for Beginners, week 3”, “Hello World” etc.

- **Boolean Values**
  e.g. true, false

- Note that strings can contain numbers, but the following variable values are **not** equivalent:

  1.234 and “1.234”

  The latter is a **string value**. This becomes important. Consider:

  1+2 = 3
  “a” + “b” = “ab”
  “1” + “2” = “12”

- Some developers use their own naming convention with variable names to denote the type of value expected to be contained in a given variable. This can often be helpful, but is in no way required by JavaScript (c.f. JavaScript comments)

- For example, **strMessage** might indicate a string variable, where **numPageHits** might indicate a numerical value in the variable.

- **Variable assignment** is accomplished in the same way as object property assignment. When a variable is assigned a value for the first time, it is automatically created. This is different from other programming languages, where a variable must be created explicitly first, before it can be loaded with a value.

- Some examples of variable assignment follow:

```
numBrowserVersion = 5.5;

numTotal += 33;

Message = "Hello!";
Message = "Goodbye";
Message = 3;
```

- Note that the last three examples show that variable values can be altered after their initial assignment, and also that the type of value stored in a variable can be altered in a similar manner.

- Once a variable has been created and a value stored within, we will want to be able to access it and perhaps manipulate it. In a similar manner to object properties, we access our variables simply by **calling them:**

```
Message = "Hello World!";
alert(Message);
```

- Note that we do not use quote marks around our variable names. The above code is different from:

```
alert("Message");
```

  for hopefully obvious reasons.

- As well as using variables for storage and access, we can combine and manipulate them using **operators**. For example:

```
a = 12;
b = 13;
c = a + b; // c is now 25
c += a; // c is now 37
c = b + " Hello!"; // c is now "13 Hello!"
```

- Our last example may have been unexpected – we added a number to a string and got a string as a result. JavaScript is smart enough to realise that a number cannot be “added” to a string in a numerical sense, so it converts the number temporarily to a string and performs a **concatenation** of the two strings. Note that **b** remains **13**, not **“13”**.

- A table of operators:

Operator	**Function**
x + y	Adds x to y if both are numerical – otherwise performs concatenation
x – y	Subtracts x from y if both are numerical
x * y	Multiplies x and y
x / y	Divides x by y
x % y	Divides x by y, and returns the remainder
-x	Reverses the sign of x
x++	Adds 1 to x AFTER any associated assignment
++x	Adds 1 to x BEFORE any associated assignment
x--	Subtracts 1 from x AFTER any associated assignment
--x	Subtracts 1 from x BEFORE any associated assignment

Project

- Open your previous project file, and save it under the name **`chapter_11.html`**.

- Clear the previous JavaScript code, and ensure that the script tags are contained in the body section of the document.

- Assign the message

  `“Welcome to my web site”`

  to a variable called **`greeting`**.

- Use this variable to create an alert box containing the message, and also to produce a header on the page without having to retype the message.
- Test this page in your browser.
- Now, modify your code to create two variables, **`var_1`** and **`var_2`**.
- Assign the value “Welcome to” to **`var_1`**, and the value “my web site” to **`var_2`**.
- Create a third variable **`var_3`**, and assign to it the value of **`var_1 + var_2`**. Then use an alert box to check the resultant value of **`var_3`**.
- Test this page in your browser.
- If the text in the alert box does not appear as expected, consider the use of spaces in the variable assignments, and correct the error.
- Now, modify your code to produce the same result but without requiring a third variable.
- Clear all statements from the current script tags.
- Add two statements to the script which assign the **numbers `100`** to one variable and **`5.5`** to another.
- Use **`document.write`** to show the effects of each of the operators given in the table on page 34 on the two numerical values.
- Substitute one of the numerical values for a text string and repeat the procedure. Note the differences.

12 Comparisons

Key Points

- Comparison operators compare two values with each other. Most commonly, they are used to compare the contents of two variables – for example we might want to check if the value of **`var_1`** was numerically greater than that of **`var_2`**.

- When you use a comparison operator, the value that is **returned** from the comparison is invariably a **Boolean** value of either **`true`** or **`false`**. For example, consider the following statements:

```
var_1 = 4;
var_2 = 10;

var_3 = var_1 > var_2;
```

  In this case, the value of **`var_3`** is false. Note that the **Boolean** value of false is not the same as the text string **`"false"`**:

```
var_4 = false;  // Boolean value
var_5 = "false";  // Text string
```

- Common comparison operators are given below:

Comparison	**Function**
X == y	Returns true if x and y are equivalent, false otherwise
X != y	Returns true if x and y are **not** equivalent, false otherwise
X > y	Returns true if x is numerically greater than y, false otherwise

X >= y	Returns true if x is numerically greater than or equal to y, false otherwise
X < y	Returns true if y is numerically greater than x, false otherwise
X <= y	Returns true if y is numerically greater than or equal to x, false otherwise

- To reverse the value returned from a comparison, we generally modify the comparison operator with a **!** (a "bang"). Note that in many cases this is not necessary, but can aid comprehension:

```
var_1 !> var_2;
var_1 <= var_2;
```

  both of these are equivalent, but one may make more semantic sense in a given context than the other.

Project

- Open your previous project file, and save it under the name **chapter_12.html**.
- Ensure that your two variables both have numerical values in them and not strings.
- Use an alert box to display the result of a comparison of your two variables for each of the comparison operators listed above.
- Substitute one of the numerical values for a text string and repeat the procedure. Note the differences.

13 Conditionals

Key Points

- Up until now, our JavaScript projects have been unable to alter their behaviour spontaneously. When a page loads with our JavaScript embedded within, it is unable to do anything other than what we expect, time and again.

- The only difference we have seen is in the use of a prompt box to alter what is shown on a page. However, the page essentially does the same thing with the text provided, regardless of what text is typed in.

- What would be really handy would be to give JavaScript a mechanism to make decisions. For example, if we provided a prompt box asking the visitor for their name, it might be nice to have a list of "known names" that could be greeted differently from any other visitors to the site.

- **Conditional statements** give us that ability, and are key to working with JavaScript.

- A conditional statement consists of three parts:

- A test (often with a comparison operator, or **comparator**) to see **if** a given condition is **true** or **false**.

- A block of code that is performed if and only if the condition is **true**.

- An optional block of code that is performed if and only if the condition is **false**.

- These three parts are represented in JavaScript as follows:

```
if ( conditional_test )
   {
   JavaScript statement;
   JavaScript statement;
   JavaScript statement;
   …
   }
else
   {
   JavaScript statement;
   JavaScript statement;
   JavaScript statement;
   …
   }
```

- Everything from the first closing curly bracket (or **brace**) is optional, so the following is also a valid conditional prototype:

```
if ( conditional_test )
   {
   JavaScript statement;
   JavaScript statement;
   JavaScript statement;
   …
   }
```

- In this case, if the `conditional_test` does not return **true**, nothing happens.

- An example of complete conditional statement is as follows:

```
if ( var_1 > var_2 )
   {
   alert("Variable 1 is greater");
   }
else
   {
   alert("Variable 2 is greater");
   }
```

- Note that the above condition is not necessarily always correct. Consider the case where **var_1** is equal to **var_2**. In that case, the above code will produce the message that "Variable 2 is greater", since **var_1 > var_2** returns **false**. In this case, we want to add an additional condition to the **else** branch of code:

```
if ( var_1 > var_2 )
   {
   alert("Variable 1 is greater");
   }
else
if ( var_1 < var_2 )
   {
   alert("Variable 2 is greater");
   }
```

- In this case, equality will produce no output, as neither of the conditions will return true. For completeness, we could add a final else branch to the statement:

```
if ( var_1 > var_2 )
   {
   alert("Variable 1 is greater");
   }
else
if ( var_1 < var_2 )
   {
   alert("Variable 2 is greater");
   }
else
   {
   alert("The variables are equal");
   }
```

- Note that in this case, we don't have to check for equality in the final branch, as if **var_1** is neither greater than nor less than **var_2**, then – numerically at least – the two must be equal.

- We can continue adding as many else if statements as required to this stack.

- If you only have one statement following your conditional test, the braces may be omitted:

```
if ( var_1 > var_2 )
   alert("Variable 2 is greater");
```

However, if you later want to add further statements to this conditional branch, you will have to add braces around the block, and this can lead to confusion. It is recommended that you use braces to enclose all blocks of conditional code.

- Consider the following block of code:

```
if ( var_1 > 4 )
   {
   var_2 = var_1;
   }
else
   {
   var_2 = 4;
   }
```

- This code is rather long, but achieves comparatively little – **`var_2`** is equal to **`var_1`** or **`4`**, whichever is greater.

- A more compact way of writing this could be:

```
var_2 = 4;
if ( var_1 > var_2 )
   {
   var_2 = var_1;
   }
```

- However, an even more compact way of writing this could be to use the **ternary operator**:

```
var_2 = (var_1 > 4) ? var_1 : 4;
```

- In the above statement, the conditional is evaluated and, if true, the value returned is the value between **`?`** and **`:`** symbols, or if false, it is the value between the **`:`** and **`;`** symbols.

Project

- Open your previous project file, and save it under the name **`chapter_13.html`**.

- Clear all JavaScript code from your script tags.

- Create two variables and assign numerical values to them.

- Use a conditional statement to show alert boxes declaring which variable is the greater of the two.
- Consider the following code:

```
var_3 = (var_1 > var_2);
```

- Use this code in your script to simplify your conditional checking code.
- Now, use a prompt box to ask your visitor their name. Assign this name to **var_3**.
- Check to see if the name typed in is your own name. If it is, use **document.write** to display a personalised greeting on the page. Otherwise, display a generic greeting.
- Use multiple else if branches to check the typed name against the names of some of your friends. Create personalised messages for all of them.
- There may be a way to simplify your conditional greeting code to use only one **document.write** statement. See if you can figure out how. Hint – how might you use a variable called **greeting**?

Project 2

- In many cases, the brevity of your conditional statements will rely on your ability to formulate the right "questions" to consider when performing your tests. Try to make your solution to the following problem as concise as possible.
- Clear all of your current code from the script tags.
- Ensure that your script tags are currently situated in the body section of the page.
- Create a variable called **exam_result** and store a numerical value of between **0** and **100** in it.

- Use an **`if`** statement and multiple **`else if`** statements to check the value of this variable against the following exam grading scheme, and print out the appropriate message to the page:

Exam Result	**Result Message**
90 or more	Excellent. Pass with Distinction.
Between 70 and 89	Well Done. Pass with Honours
Between 55 and 69	Just passed.
54 or below	Failed. Do better next time.

- Test your result in your browser. Vary the value of **`exam_result`** and check the value shown in the browser. For extra practise, try to use a prompt box to make changes to your **`exam_result`** variable as easy to achieve as possible.

14 Looping

Key Points

- The letters **i**, **j** and **k** are traditionally used by programmers to name variables that are used as counters. For example, at different stages of the program, **i** may contain the numbers 1, 2, 3 etc.

- In order to achieve a "counting" effect, you will need to **increment** or **decrement** the value of your counting variable by a set value. Here are some examples:

```
i = i + 1;

i = i - 1;

i = i + 35;

incr = 10
i = i + incr;
```

- To keep things concise, we can use the following shortcuts:

```
i++;  // equivalent to i = i + 1;
i--; // equivalent to i = i + 1;
```

- Counting in JavaScript, like many other programming languages, **often begins at zero**.

- In many cases, this makes a lot of sense, as we will see. However, it can often cause confusion. Consider starting at 0 and counting up to 10. In that case, we may have actually counted 11 items:

  0 (1)
  1 (2)
  2 (3)
  3 (4)
  4 (5)
  5 (6)
  6 (7)
  7 (8)
  8 (9)
  10 (11!)

- If you wanted to give an instruction to someone to perform a repetitive action, you might say that you wanted them to continue the action for a certain number of times. If someone were performing an action 300 times, for example, they might do something like the following to ensure that their count was accurate:

- Write the number 1 on a bit of paper.

- After each action, erase the number on the bit of paper and increment it by 1.

- Before each action, check the number on the bit of paper. If it is less than **or equal to** 300, perform the action.

- Alternatively, they might decide to start counting at 0. In this case, the procedure would be identical, but the check before each action would be to make sure that the number was **strictly less than** 300.

- In JavaScript, we say almost the same thing. The following code will display the numbers 1 to 100 on the page:

```
for     ( i = 1; i <= 100; i++ )
   {
   document.write(“<p>” + i “ </p>”);
   }
```

- The **for** statement tells the browser that we are about to perform a **loop**. The layout here is very similar to a conditional statement, but in this case we have much more information in the brackets. Where our conditional had one JavaScript statement to describe its action, a **for loop** has three:

- An initialiser – this sets up the initial counting condition, in this case `i = 1`.

- A conditional – this is identical to our conditional statements earlier, and must return **true** or **false**. If it returns **true**, the loop continues, otherwise it exits.

- An incrementer – this defines the action to be performed at the end of each loop. In this case, i is incremented by a value of 1.

- The key difference between a conditional and a for loop is that the condition is constantly being changed and re-evaluated. It is possible to create an infinite loop by making the conditional non-reliant on the count value – for example:

```
for     ( i=0; 5 > 4; i++ )
```

  will always perform the script in the braces, and will probably cause errors in the browser.

- Note too that it is very common to start counting at zero in JavaScript. The reason for this is that it is often desirable to count **how many times** an operation has been performed. Consider the following:

```
for     ( i=1; 1 < 2; i++ )
```

- In this case, the loop will run once, but the value of `i` will be 2, as after the first run, `i` will be incremented to 2, and will then fail the test and so the loop will exit. If we use the following:

```
for     ( i=0; 1 < 1; i++ )
```

  Then the loop will run once, and the value of `i` afterwards will be 1, as we might hope.

Project

- Open your previous project file, and save it under the name **`chapter_14.html`**.

- Clear all JavaScript code from your script tags.

- Write a series of statements to produce a multiplication table as follows:

The 12x Multiplication Table

1 × 12 = 12

2 × 12 = 24

3 × 12 = 36

4 × 12 = 48

5 × 12 = 60

6 × 12 = 72

7 × 12 = 84

8 × 12 = 96

9 × 12 = 108

10 × 12 = 120

11 × 12 = 132

12 × 12 = 144

- The following exercise is more of an HTML example, but demonstrates an important facet of using JavaScript (or, indeed, any programming language) to produce well-formatted text.

- Modify your previous code to make your page's content appear in the centre of the page. Put your multiplication table in an HTML table to make sure that the equals signs, multiplication signs and so forth line up in neat columns:

The 12x Multiplication Table

1	×	12	=	12
2	×	12	=	24
3	×	12	=	36
4	×	12	=	48
5	×	12	=	60
6	×	12	=	72
7	×	12	=	84
8	×	12	=	96
9	×	12	=	108
10	×	12	=	120
11	×	12	=	132
12	×	12	=	144

- As a hint, here is a look at the table cells involved:

The 12x Multiplication Table

1	× 12 =	12
2	× 12 =	24
3	× 12 =	36
4	× 12 =	48
5	× 12 =	60
6	× 12 =	72
7	× 12 =	84
8	× 12 =	96
9	× 12 =	108
10	× 12 =	120
11	× 12 =	132
12	× 12 =	144

15 Arrays

Key points

- In many cases, variables will completely satisfy our data storage needs in JavaScript. However, in a large number of cases, we may wish to “group” variables into a collection of related items.

- Take, for example, days of the week. In each day we perform a number of tasks, so we could want to record each task as a separate item under a group called, say, Monday’s Tasks.

- In JavaScript, to achieve this we would store each task in a separate variable, and then group those variables together into an **array**.

- An **array** is a special type of JavaScript object that can store multiple data values – unlike a variable, which can only store one data value at a time.

- It could be helpful to think of an array as a row of mail boxes in an office, just as you might think of a variable as a single, solitary mail box.

- The boxes in an array are numbered upwards, starting at box number 0 – note that counting begins at 0 here, just as we discussed in the previous chapter. The number assigned to each box is known as its **index**.

- In order to use an array in JavaScript, you must first create it. There are a number of ways to create arrays in JavaScript. The simplest follows:

  ```
  arrDays = new Array();
  ```

  This statement creates a new, empty array called arrDays. We can call arrays just like we can variables, but with a few minor adjustments.

- If you already know how many elements a given array will have, you can declare this explicitly:

  ```
  arrDays = new Array(7);
  ```

  This modification creates an array with 7 empty boxes. However, arrays will expand and contract to the required size in JavaScript, so the cases where you will need to state the size of the array are rare.

- More useful, however, is the ability to "fill" the boxes of an array when you create it. For example:

  ```
  arrDays = new Array("Monday","Tuesday");
  ```

  We now have an array with two elements. The first (element 0) has a value of "Monday", while the second (element 1) has a value of "Tuesday". We need not restrict ourselves to string values in arrays – Boolean, numerical and string values are allowed, as in arrays. It is even possible to assign other arrays to array elements – more on this later.

- The most often-used way of creating an array is to use "square bracket" notation. Square brackets play a large role in the use of arrays, so this is often the easiest method to remember:

  ```
  arrDays = ["Monday","Tuesday"];
  ```

  This is identical to the previous example.

- To access an array's elements, we first call the array's name, and then specify the number of the element in square brackets, like so:

  ```
  alert(arrDays[0]);
  ```

  Note the lack of quotes around the 0 here. This line of code is equivalent to:

  ```
  alert("Monday");
  ```

  assuming the array is defined as in the previous examples.

- Not only can we access the value of an array element using this notation, but we can also **assign** the value as well:

  ```
  arrDays[2] = "Tuesday";
  arrDays[3] = "Wednesday";
  ```

- If you wish to add an element to an array **without** knowing the index of the last element, you can use the following code:

  ```
  arrDays[] = "some other day";
  ```

- As we will see, arrays are actually just special JavaScript objects, and have properties and methods associated with them. The most important property that every array has is its length property:

  how_many_days = arrDays.length;

- As well as properties, arrays have very useful methods. If you wished to sort your array alphanumerically, you could use the array's sort method thus:

  arrDays.sort();

  Note that the sort method works on the actual array itself, over-writing the current values. So, if you had an array with each day of the week as an element, calling its sort method would mean that **`arrDays[0]`** was then equal to "Friday", not "Monday".

Project

- Open your previous project file, and save it under the name **`chapter_15.html`**.

- Clear all JavaScript code from your script tags.
- Write a few JavaScript statements that will present the months of the year on the page in alphabetical order. You should use the following technique to achieve this:
- Store the names of the months of the year in an array.
- Use an array method to sort the array elements alphanumerically.
- Use a **for** loop to *iterate* through each array element in turn, and print the value of the element to the screen (hint, consider the use of **`array[i]`**, where **`i`** is the for loop's counter).
- The above method (the use of a **for** loop to iterate through a series of array elements) is one of the first common programming techniques we have discussed in this course. Its usefulness cannot be overstated, as it allows us to perform repetitive tasks on a series of related elements *without necessarily knowing what those elements might be when we wrote the code*. It can be applied to form elements, cookies, page elements, pages, windows, and just about any other collection of object that you might wish to manipulate with JavaScript.
- To reinforce this generalism, if you have not used the **`array.length`** value in your loop, consider its use now. To prove that you have created a more generic loop, try the code with an array of days instead of an array of months, and see if you have to change any of the looping code.

16 Associative & Objective Arrays

Key Points

- We have already seen that we can access array elements by their index:

```
arrDays = ["Monday", "Tuesday"];

// print "Monday" to the page
document.write(arrDays[0]);
```

- However, it might be more useful to be able to **name** our array elements. By default, an array will be created as a **numeric** array. We can also create an **associative** array:

```
arrDays = new Array();

arrDays["Monday"] = "Go to the dentist";
arrDays["Tuesday"] = "Attend JavaScript
class";
arrDays["Wednesday"] = "JavaScript homework";

// remind you of Wednesday's task
alert(arrDays["Wednesday"]);
```

- This looks a lot like our previous discussion of objects and properties. In fact, since an array *is* actually an object, we can access its elements as though they were properties:

```
// remind you of Wednesday's task
alert(arrDays.Wedensday);
```

- Note a subtle difference here – in our previous, numeric array examples, the names of the week days were the **values** of our array elements. Here, the names of the week days are the **indexes** of our elements. Avoid the following common error:

```
arrDays = ["Monday","Tuesday"];
arrDays["Monday"] = "Go to work";

// this is actually equivalent to
arrDays = new Array();

arrDays[0] = "Monday";
arrDays[1] = "Tuesday";
arrDays["Monday"] = "Go to work";

// and arrDays.length is now 3, not 2
```

Project

- Open your previous project file, and save it under the name `chapter_16.html`.
- Clear all JavaScript code from your script tags.
- Write a new script which creates a new, seven element associative array called Week:
- Use the days of the week as indexes for each element of the array.
- Assign a task to each day of the week as each associative element is created.
- Use a for loop to display a calendar on the page, as follows:

  Monday: task
  Tuesday: task

  etc…

- Modify your code to use a prompt box to ask the visitor to choose a day, and display on the page the task allotted to that day.

17 Two Dimensional Arrays

Key Points

- Referring back to our mailbox analogy, where our array could be pictured as a row of mailboxes, each with its own contents and label, a two dimensional array can be thought of as a series of these rows of mailboxes, stacked on top of each other.

- In reality, a **two dimensional array** is simply an array in which each element is itself an array. Each "sub array" of a two dimensional array can be of a different length – in other words, the two dimensional array doesn't have to be "square".

- You can access the contents of each sub array by using two pairs of square brackets instead of just one. An example will illustrate this best:

```
array_1 = ["element", "element 2"];
array_2 = ["another element", 2, 98, true];

array_3 = [array_1, array_2];

alert(array_3[1][3]);  // displays "98"
```

- While you can't mix numerical and string indexing systems in a single array (i.e. an array cannot be both numerical *and* associative), you can have both associative and numerical arrays in two dimensional arrays. For example, consider the above recast as follows:

```
array_3 = new Array();
array_3["firstArray"] = array_1;
array_3["secondArray"] = array_2;

alert(array_3["secondArray"][3]);
   //displays "98" again
```

- Similarly, we can happily use our "objective" notation for associative arrays:

  ```
  alert(array_3.secondArray[3]);
     //displays "98" yet again
  ```

Project

- Open your previous project file, and save it under the name **`chapter_17.html`**.

- Building on your previous project, create a number of new, seven element associative arrays to represent 4 separate weeks.

- Combine these 4 weeks into a four element array to represent a month.

- Modify your previous code to take a week number and print out all that week's activities to the page.

- Modify one of your week arrays to consist not of single elements, but of arrays of hours from 8am to 5pm. This then represents a *three dimensional array*. We can extend arrays to be ***n-dimensional***, where ***n*** is more or less arbitrary.

- Finally, alter your code to prompt the user for **three** values – a week, a day and an hour. Store these values in three separate variables, and use those variables to display the requested task, or else to display an error message if a task cannot be found.

18 String Manipulation

Key Points

- Throughout your use of JavaScript in a production environment, you will often use it to read values from variables, and alter a behaviour based on what it finds.

- We have already seen some basic string reading in the section on comparisons where we test for equality. However, this all-or-nothing approach is often not subtle enough for our purposes.

- Take the case where we want to check a user's name against a list of known users. If the user enters their name as "Karen", for example, that will be fine if **and only if** they spell the name precisely as we have it recorded, including capitalisation etc. If the user decides to type in her full name, say "Karen Aaronofsky", the code will not recognise her.

- In this case, we want to see if the text "Karen" appears at all in the string. We call this **substring searching**, and it can be incredibly useful.

- One of the simplest substring searches is done by using the **`indexOf`** method. Every string-type variable has this method associated with it. Consider this code:

  ```
  var_1 = "Karen Aaronofsky";
  var_2 = var_1.indexOf("Karen");
  ```

  In this case, the value of **`var_2`** will be 0 – remembering that JavaScript begins counting at 0!

- If we were to search for a surname here:

  ```
  var_1 = "Karen Aaronofsky";
  var_2 = var_1.indexOf("Aaronofsky");
  ```

  var_2 will have a value of 6.

- Finally, if the search were to "fail", so say we searched for the name "Anisa" as a substring, the value of **var_2** would then be -1.

- Note that this is more flexible, but still presents an issue if the user forgets to capitalise any of the substrings that we are searching for – in JavaScript, "Karen" does not equal "karen".

- In order to get around this, we might want to ensure that capitalisation is not taken into account. A simple way to achieve this is to force strings into lowercase before the check is performed:

  ```
  real_name = "Karen";
  name = prompt("Your name?","");

  real_name = real_name.toLowerCase();
  try_name = try_name.toLowerCase();

  if ( try_name.indexOf(real_name) > -1 )
  {
    alert("Hello Karen!");
  }
  else
  {
   // note we use the original,
   // non-lower-cased name here
   alert("Welcome " + name);
  }
  ```

- There are a number of string methods we can use to perform "value checks" on strings. A few are printed in the following table:

Method	**Behaviour**
String.indexOf("str")	Returns the numerical position of the first character of the substring "str" in the String
String.charAt(x)	Returns the character at position x in the string – the opposite of indexOf

String.toLowerCase()	Returns a copy of the string with all capital letters replaced by their lowercase counterparts
String.toUpperCase()	Returns a copy of the string with all lowercase letters replaced by their capital counterparts
String.match(/exp/)	Returns true or false based on a **regular expression** search of the string

- The final method here deserves some comment. What is a **regular expression**?

- A **regular expression** is a standard way of writing down a "pattern" of characters that is easily recognisable. For example, consider a typical email address:

  jonathan@relativesanity.com

- An email address follows a "pattern" which makes it instantly recognisable as an email address to a human. Wouldn't it be handy if we could define that pattern in JavaScript for a browser to use? Regular expressions allow us to do just that.

- Let's look at our email address again. We can break it down to a "prototype" email address as follows:

  [some letters]@[some more letters].[a few more letters]

- Of course, it's slightly more complex than that – there are some characters which aren't allowed to be in certain parts of the email address (no spaces, for example), but this lets you see the idea of breaking this string up into required "chunks".

- Now, to convert this to a regular expression. Just as we use quote marks to denote (or "delimit") string values in JavaScript, to signify a regular expression, we use forward slashes: /. Our email address regular expression might look like this:

  ```
  /^.+@.+\..+$/
  ```

- This warrants some discussion. Let's look at this a character at a time:

- / denotes the start of the regular expression

- ^ denotes that we want this regular expression to be found at the very beginning of the string we are searching.
- **.+** the dot symbol is used to stand in for **any** character. The plus signifies we want to find *at least one* of those. So this is equivalent to our plain-English phrase [some letters].
- **@** this is simply a character – it has no special meaning other than to say we want to find an @ character after at least one character from the beginning of the string.
- **.+** the same as before – at least one more character after the @.
- **\.** This is interesting. We know that the dot symbol has a special meaning in a regular expression – it means "match any character". However, here we want to find an **actual dot**. Unlike @, which has no special meaning, we have to tell JavaScript to ignore the dots special meaning. We do this by preceding it with a backslash, which tells JavaScript to treat the character immediately following it as though it has no special meaning. This is a convention you will come across many times while programming. The net result here is that we want to match a dot after a series of characters.
- **.+** and again, at least one more character after the dot.
- **$** this is the mirror of the ^ at the beginning – this matches the end of the tested string.
- / tells JavaScript we are at the end of the regular expression.
- Phew! Lots to consider here. Regular expressions are an arcane art at the best of times, so don't worry too much if the above code is indecipherable. The important thing to realise at the moment is that we can perform some quite sophisticated pattern recognition in JavaScript without having to resort to checking each individual character of a string multiple times.

- The following code checks a variable to see if it looks like an email address:

```
var_1 = prompt("Your email?", "");

if ( var_1.match(/^.+@.+\..+$/) )
{
  alert("valid email address");
}
else
{
  alert("are you sure?");
}
```

- There are a few problems with this code at the moment – for example, it will pass the string "-@-.-" quite happily, which is clearly wrong. We will look at ways around this later on in the course.

Project

- Open your previous project file, and save it under the name **`chapter_18.html`**.
- Clear all JavaScript code from your script tags.
- Use a prompt box to capture some user input to a variable called **`check_string`**.
- Use a document.write statement to output the results of each of the various string methods when applied to the user input.
- Check the user input to see if it's an email address, and alter your output accordingly to either "That's an email address" or "That doesn't look like an email address to me!"
- In the latter case, output the failed string as well so that the user can see their input and modify it next time, if appropriate.

19 Using Functions

Key Points

- A **function** is a named set of JavaScript statements that perform a task and appear inside the standard **<script>** tags. The task can be simple or complex, and the name of the function is up to you, within similar constraints to the naming of variables.

- JavaScript **functions** are declared before they are used. The declaration looks like this:

```
function name_of_function( )
{
  …your code here…
}
```

- Unlike all the JavaScript instructions we have looked at so far, the code inside a function will not be run until specifically requested. Such a request is called a function **call**.

- Functions can be called from anywhere on the page that JavaScript can live, but must be called *after* the function has been declared. For example, if you declare a function in the body of a document and then call it from the head of the document, you may find that the browser returns an error message. For this reason, most JavaScript programmers define any functions they are going to use between **<script>** tags in the head section of their pages to ensure that they are all defined before they are used.

- Functions are to object methods as variables are to object properties – they are also called in a similar manner. To run the code contained in the **name_of_function** function above, we would use the following line of code:

```
name_of_function();
```

- Note the parentheses after the function name. This lets JavaScript know that we are dealing with a function and not a variable. The parentheses also allow us to "pass" extra information to the function, which can alter its behaviour. Consider the following function:

```
function greet_user( username )
{
  message = "Hello " + username;
  alert(message);
}
```

- Whenever the function is called, it will greet the user named. How can we pass this information through? Consider:

```
greet_user("Anisa");
```

or

```
var_1 = prompt("Name?", "");
greet_user(var_1);
```

- We should use functions in our code as often as possible. Whenever we perform an action that isn't accomplished by the use of a method or an assignment operator, we should try to build a function that can accomplish the task.

- For example, we can build a function to check email addresses:

```
function check_email( address )
{
  var_1 = false;
  if ( address.match(/^.+@.+\..+$/) )
  {
    var_1 = true;
  }
}
```

- The above function will take a string that is passed to it (often called the function's **argument**), and will alter the value of `var_1` depending on what it finds. However, the function is lacking an important ability – the ability to communicate its findings back out to the rest of the script.

- We have mentioned **return values** a few times in the notes. Now we see a situation that requires a function to **return** its findings to the rest of the code. Ideally, we'd like to be able to use the above function as follows:

```
if ( check_email(address) )
{
  …do some email things…
}
```

- In order for this to work, the return value from check_email would have to be a Boolean value. We can arrange this quite simply:

```
function check_email( address )
{
  var_1 = false;
  if ( address.match(/^.+@.+\..+$/) )
  {
    var_1 = true;
  }
  return var_1;
}
```

- Since var_1 is either true or false, the returned value will be Boolean. We can even skip the use of the variable here and be more direct:

```
function check_email( address )
{
  if ( address.match(/^.+@.+\..+$/) )
  {
    return true;
  }
  return false;
}
```

or even better, since address.match() will return a Boolean value of its own:

```
function check_email( address )
{
  return address.match(/^.+@.+\..+$);
}
```

- The above function may not seem like a great saving. After all, we are using four lines of code to define a function that performs only one line of code. Compare:

```
function check_email( address )
{
  return address.match(/^.+@.+\..+$);
}

if ( check_email(address) )
{
  …do some email things…
}
```

  with:

```
if ( address.match(/^.+@.+\..+$/) )
{
  …do some email things…
}
```

- While the benefits here are not obvious, consider the case where, at some point in the future, you discover a better method of checking for email addresses. In the second case abov, you will have to search your code for each and every instance of that method, and replace it with your new method, which may not be one line of code. In the first case, you will just have to change the underlying function definition, and the "upgrade" will be effective throughout your code without you having to update each occurrence.

Project

- Open your previous project file, and save it under the name **`chapter_19.html`**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and one in the body area.
- In the head area, define a function called **`show_message`**. This function should take one argument, called **`message`**, and should use an alert box to display the contents of the argument.
- In the body area, call the function with various messages as arguments.

- Now use a variable in the body area to store the return value of a prompt asking the user for a message. Use this variable as the argument to a single instance of **`show_message`**.

- Define a new function in the head area called **`get_message`**. It should take no argument, but should replicate the function of your prompt in the body area and ask the user for a message via a prompt box.

- Make sure that **`get_message return`**s a sensible value. We are aiming to replace our prompt statement in the body area with the following code:

  ```
  message = get_message();
  ```

  so consider what you will have to return to enable this to work.

- Once you are happy with your **`get_message`** definition, try replacing your prompt code in the body area with the statement above.

- To demonstrate the power of functions, change the action of **`show_message`** to write the message to the page without changing any code in the body area of the page.

20 Logical Operators

Key Points

- In our discussion of conditionals, we saw how to check the veracity of a single condition via a comparator:

```
if ( x > some_value )
{
  …expressions…
}
```

- We have also seen the limitations of such an approach. Let us say we wanted to discover if **x** lay *between* two values, say **val_1** and **val_2**. There are a number of ways we could achieve this. In our example on student grades, we learned that we could use an **if…else** pair to achieve this effect:

```
if ( x > val_1 )
{
  …do something…
}
else
if ( x > val_2 )
{
  …do something else…
}
```

- The above code achieves what we want – for the second branch, x must lie between **val_2** and **val_1** (assuming **val_1** is greater than **val_2**, of course). However, it's rather unwieldy, and does not scale elegantly to checking three conditions (say we wanted to check if **x** was an even number as well), or in fact to ten conditions.

- Enter **Logical Operators**. These operators are used to join together conditional checks and return true or false depending on whether **all** or **any** of the checks are true.

- In English, we refer to these operators by using the words "AND" and "OR".

- For example, say we wanted to do something each Tuesday at 8pm. We would want to check whether the current day was Tuesday, **and** whether the time was 8pm.

- Another example: Let's say we wanted to do something on the first Tuesday of each month, and also on the 3rd of the month as well. We would have to check whether the current day was the first Tuesday of the month, **or** whether it was the 3rd day of the month.

- Note in the last example, if **both** conditions were true, then we would be on Tuesday the 3rd and would perform the action. In other words, an **or** condition allows for either one, or the other, *or both* conditions to be true.

- In JavaScript, we use the following syntax to check multiple conditions:

  ```
  ( 100 > 10 && 5 < 8 )
  ```

  translates as “if 100 is greater than 10 and 5 is less than 8”. In this case, the result is **true**.

  ```
  ( 100 > 200 && 4 < 9 )
  ```

  in this case, the result is **false**. Note here that only the first condition is actually checked. Since **and** requires both comparisons to be true, as soon as it finds a false one it stops checking. This can be useful.

  ```
  ( 100 > 10 || 9 < 8 )
  ```

  translates as “if 100 is greater than 10 or 9 is less than 8”. In this case, the result is **true**, since at least one of the conditions is met.

  ```
  ( 100 > 200 || 4 > 9 )
  ```

  in this case, the result is **false** since neither of the comparisons are true. Finally:

  ```
  ( 100 > 200 || 5 < 2 || 3 > 2 )
  ```

  in this case, the result is **true**. Any one of the three being true will provide this result.

- As we can see from the last example, this method of checking *scales* to any number of conditions. We can also mix and match the operators. For example:

  ((100 > 200 && 100 > 300) || 100 > 2)

  in this case, the **and** condition evaluates to **false**, but since either that **or** the last condition has to be true to return **true**, the overall condition returns **true** as 100 is indeed greater than 2.

- This sort of complex logic can take a while to comprehend, and will not form a set part of the course. However, it is useful to be aware of it.

Project

- Open your previous project file, and save it under the name **`chapter_20.html`**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and one in the body area.
- Copy the file **`available_plugins.js`** from the network drive (your tutor will demonstrate this), and open it using NotePad's File > Open command.
- Copy and paste the entire contents of **`available_plugins.js`** into your current project file, into the script element in the head area of your page.
- Have a read through the code. Note that it defines a large, two dimensional array. The array has a list of various components that can be present in web browsers (such as Flash or Quicktime)
- Add a function to the head area script element, called **`flash_exists()`**. This function should use a **for loop** to check each of the elements of the **`available_plugins`** array and establish if Flash is present.
- Add a further function to the head area script element, called **`quicktime_exists()`**. This function should also use **a for loop** to check each element of the array, this time returning true if Quicktime is present.
- Finally, add a function to the head area script element called **`both_quicktime_and_flash_exist()`**. This function should call both of the previous functions, store their results in a variable, and produce an alert box containing the message:
- "Both Quicktime and Flash" if both functions returned true; or:
- "One of Quicktime or Flash is missing" if either of the functions return false.
- Call the final function from the body area script element.
- Check your results in your browser.

21 Using Event Handlers

Key Points

- So far, our scripts have run as soon as the browser page has loaded. Even when we have used functions to "package" our code, those functions have run as soon as they have been called in the page, or not at all if no call was made. In other words, the only event our scripts have so far responded to has been the event of our page loading into the browser window.

- Most of the time, however, you will want your code to respond specifically to user activities. You will want to define functions, and have them spring into action only when the user does something. Enter **event handlers**.

- Every time a user interacts with the page, the browser tells JavaScript about an "event" happening. That event could be a mouse click, a mouse pointer moving over or out of a given element (such as an image or a paragraph), a user tabbing to a new part of an HTML form, the user leaving the page, the user submitting a form and so on.

- An **event handler** is a bit of JavaScript code that allows us to capture each event as it happens, and respond to it by running some JavaScript code.

- In general, we attach event handlers to specific HTML tags, so a mouse click on one element of the page might be captured by an event handler, but clicking somewhere else might do something completely different, or indeed nothing at all.

- Some common event handlers are in the table below:

Event Handler	Occurs When…
onload	An element is loaded on the page

onunload	An element is not loaded, for example when a user leaves a page
onmouseover	When the mouse pointer enters the area defined by an HTML element
onmouseout	When the mouse pointer leaves the area defined by an HTML element
onclick	When the left mouse button is clicked within the area defined by an HTML element
onmousedown	When the left mouse button is depressed within the area defined by an HTML element
onmouseup	When the left mouse button is released within the area defined by an HTML element

- The last three are related, but there are subtle differences – onclick is defined as being when **both** mousedown and mouseup events happen in the given element's area. For example, if you click on an area of the page, that registers the area's mousedown event. If you then hold the mouse down and move to another area before releasing, it will register the other area's mouseup event. The browser's click event, however, will remain unregistered.

- In theory, we can add most of these event handlers to just about any HTML tag we want. In practise, many browsers restrict what we can interact with.

- We will mostly be attaching event handlers to **<img>**, **<a>** and **<body>** tags.

- To attach an event handler to a tag, we use the following method:

```
<a href="…" onclick="a_function();">link</a>
```

- We can use this method to attach any event handler listed above to the elements of the page. In addition to calling functions (with any optional arguments, of course), we can write JavaScript directly into our event handlers:

```
<a href="…" onclick="alert('hello');">link</a>
```

- Note the potential issue with quote marks here – if you use double quotes around your event handler, you need to use single quotes within and vice versa.

Project

- Open your previous project file, and save it under the name **chapter_21.html**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and none in the body area.
- Within the head area script element, define the following function:

```
function set_status( msg )
{
  window.status = msg;
}
```

- When called, this function will set the text displayed in the browser's status bar (the part of the window below the page content) to whatever is passed as an argument. For example, to set the status bar to display "Welcome", we would call:

```
set_status("Welcome");
```

- Now define the following function immediately below the last:

```
function clear_status( )
{
  set_status("");
}
```

- When called, this function will clear the status bar. Notice that we are using our previous function within the new one. This is a common programming technique that allows us to define functions of specific cases using more general functions.
- Now, add the following HTML to the body area of your page. Remember, we're adding HTML here, not JavaScript, so do not be tempted to use script tags for this part of the project:

```
<a href="#" ↲
   onmouseover="set_status('hello');" ↲
   onmouseout="clear_status();">testing</a>
```

- Load the page in your browser and observe what happens when you move your mouse over the link.
- The # value for the href attribute of the link allows us to define a “dead” link on the page. Clicking on the link will take you nowhere – try it.
- Now, alter the code to have the link point at a real website that you know of.
- Clicking on the link now will take you away from the page. Let’s say we want to suppress that behaviour.
- When an event handler intercepts an event, it pauses the normal action of the event. For example, if you used an onclick handler on a link to pop up an alert box, the link would only be followed **after** the alert box had been dismissed. We can use event handlers to cancel the action if required by using their return values.
- Add a new function to the head area script element:

```
function confirm_link( )
{
  check = confirm(“This will take you ↲
    away from our site. Are you sure?”);
  return check;
}
```

- The value of **`check`** will be **true** or **false**.
- Now, modify your link to contain the following event handler:

```
onclick=”return confirm_link();”
```

- By using the word **`return`** in our event handler, the response of the function will be used to decide whether the rest of the normal action is run. In this case, if **`confirm_link()`** returns **false**, our link action will be cancelled.
- Load your page in your browser and view the result.

22 Working with Images

Key Points

- In HTML, we can identify specific elements on the page using an id attribute. For example, to “name” an image, we can use the following code:

  ```
  <img src="logo.gif" alt="" id="theLogo" />
  ```

- To refer to this element in JavaScript, we can now get to it directly by its id value:

  ```
  document.getElementById("theLogo")
  ```

- This method will return an object that refers to the given element on the page. If no such element can be found, the method will return false.

- For easier use, we can assign the object found to a variable. For example, to create an object called our_logo in our scripts, we can use the following line of code:

  ```
  our_logo = document.getElementById("theLogo");
  ```

- The resultant object has a number of properties. Since, in this case, our object represents an image element, its properties include:

  ```
  our_logo.height
  our_logo.width
  our_logo.src
  ```

- We can use JavaScript to change any of these properties, so if we wanted to change the image displayed, we could do so as follows:

  ```
  our_logo.src = "new_logo.gif";
  ```

Project

- Copy the folder called **`images`** from the network drive to your project folder.
- Open your previous project file, and save it under the name **`chapter_22.html`**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and none in the body area.
- Within the body area of your page, create an image element that loads an image from the **`images`** folder. Give the element an appropriate **`id`** attribute.
- In the head area script element, define a function called describe_image() that will pop up an alert box containing the following information about your image:

  the image file used
  the image width
  the image height

- To have each bit of text appear on a separate line, you can add the following character to your alert text:

  ```
  \n
  ```

  for example

  ```
  alert("line one\nLine two");
  ```

- Load your page in the browser and view the results.

23 Simple Image Rollovers

Key Points

- A simple image rollover is an effect which happens when the mouse appears over an image (usually a linked button) on the page. The original image is replaced by another of equal size, usually giving the effect that the button is highlighted in some way.

- With what we have learned so far, we already have the ability to create a simple image rollover effect. All that remains is to clarify the particulars of the process:

- The page is loaded and the original image appears on the page as specified by the **`<img>`** tag's **`src`** attribute.

- The mouse moves offer the image and the alternative image is loaded into place.

- The mouse leaves the image and the original image is loaded back.

- As you may have realised, we are going to use JavaScript to alter the **`src`** attribute of the image tag. The best way to think of this is to picture the **`<img>`** tag as simply a space on the page into which an image file can be loaded. The **`src`** attribute tells the browser *which* image to load into the space, and so if we change that value, the image will be changed.

- In other words, the **`id`** attribute of the **`<img>`** tag is naming the "space", not the image.

- Now, in order to alter the src attribute with JavaScript, we need to tell JavaScript which image “space” we want to alter. We use the id attribute along with the getElementById() method from the last chapter to do this:

```
button_img = ↲
    document.getElementById(“button”);

button_img.src = “new_image.jpg”;
```

- We can directly insert this code into the image’s event handler:

```
<img src=″old.jpg″ id=″button″ ↲
    onmouseover=↲
    ″document.getElementById(‘button’).src ↲
     = ′new.jpg′;″>
```

- Note that this code is suddenly very convoluted. There are two immediate potential solutions. The first is to define a function:

```
function swap_image( id, new_image )
{
  img = document.getElementById(id);
  img.src = new_image;
}

…

<img src=″old.jpg″ id=″button″ ↲
    onmouseover= ↲
    ″swap_image(‘button′, ‘new.jpg′);″>
```

- This is a much cleaner solution, and more importantly we can use this for any images on the page, simply by changing the arguments of our function call. We can also use the function to achieve the “swap back” functionality:

```
…

<img src=″old.jpg″ id=″button″ ↲
    onmouseover= ↲
    ″swap_image(‘button′, ‘new.jpg′);″ ↲
    onmouseout= ↲
    ″swap_image(‘button′, ‘old.jpg′);″>
```

- We can go even further in "cleaning up" our code, though. Because the event handler is being used to alter the object which is experiencing the event (ie, the mouse is moving over the image tag that we are trying to change), we can use the "automagic" JavaScript object **this** to perform the operation:

```
function swap_image( img, new_image )
{
  img.src = new_image;
}

…

<img src="old.jpg" id="button" ↲
    onmouseover= ↲
    "swap_image(this, 'new.jpg');">
```

- Note a couple of things. Firstly, **this** has no quotes around it – we are using it like a variable name. Secondly, our function now uses the first argument directly, instead of using it to get the relevant object from the page. We can do that because **this** is actually an object – it's an object that takes on the properties and methods of whatever object it is called from, in this case, it becomes equivalent to **document.getElementById('button')**, although is obviously much shorter!

- Using **this** has some limitations. For example, if we wanted to change the **src** attribute of any other image on the page when the mouse moved over "**button**", we would be unable to use this, and hence would have to define another function that could take an id as its argument and get the relevant object that way.

Project

- Copy the folder called **buttons** from the network drive to your project folder.

- Open your previous project file, and save it under the name **chapter_23.html**.

- Clear all JavaScript code from your script tags.

- Ensure that you have a script element in the head area of your document, and none in the body area.

- Within the body area of your page, create a paragraph containing six image elements. Set the **src** attributes of the images to load the following files in your copied **buttons** folder, and give each a sensible **id**:
- **contacts.jpg**
- **home.jpg**
- **people.jpg**
- **products.jpg**
- **quotes.jpg**
- **whatsnew.jpg**
- Create a JavaScript function in the head area script element that takes two arguments – an **id** and a file name. It should alter the src property of the appropriate image object to the file name given.
- Use this function to swap the **src** attribute of the contacts button to **contactsover.jpg** when the mouse moves over the image.
- Once you have this working, update the remaining five images with event handlers to swap their **src** attributes to their appropriate "over" image.
- Add event handlers to all six images to ensure that they return to their original state when the mouse is moved away from them. Check your work in your browser
- Add a new paragraph above the previous one, and add an **<img>** tag to it to containing the file **rocollogo.jpg** from the images folder.
- Add a text link to a new paragraph between the two paragraphs. The link should be a "dummy" link (ie use "**#**" as its **href** attribute value), but when the mouse moves over it, the image above it should change to show **rocollogo.gif**.
- Moving the mouse away from the text link should return the logo to its previous state.
- Check your work in your browser.

24 Object Instantiation and Better Rollovers

Key Points

- So far we have seen a very simple example of an image rollover. It is functional and works as desired, but it is lacking in a few finer details.
- Specifically, when we use JavaScript to change the src attribute of an <img> tag, the browser has to load this image from scratch. On our local machine, this will not cause an appreciable delay, but when dealing with a remote server (as we will be on the Internet), this delay can lead to a noticeable "lag", which can destroy the feeling of a dynamic interface.
- Ideally, we would like to instruct the browser to load any alternate images when it loads the page. This will allow us to ensure that the new images are sitting on the user's computer, ready to be swapped in and out instantly.
- To do this, we need to look at **object variables** and **object instantiation** (or *creation*). In particular, we need to look at the Image object.
- Each `<img>` tag on the page has a corresponding Image object in JavaScript. We have looked at ways of manipulating this directly, and have an idea of some of the properties an Image object can have.
- However, we can create Image objects directly in JavaScript, without the need for a corresponding `<img>` tag on the page. When we create such an object, and set its src property, the browser will load the appropriate file into memory, but will not display the image on the page.

- In other words, we can create "virtual" images that exist within JavaScript, and use these Image objects to store the alternate images for our "real" images.

- To create an Image object (in fact, to create *any* object), we need to use the following code:

```
virtual_image = new Image();
```

- We have seen this sort of syntax before – the use of the **`new`** keyword when we created our Arrays. **`new`** tells JavaScript that we are creating a new object. The **`virtual_image`** part of the assignment is just a variable name. In this case, the variable is an **Object Variable**, since it contains an object.

- To use this variable to preload our images, we take advantage of the fact that it has the same properties and methods as any other image:

```
virtual_image.src = "contactsover.jpg";
```

- The browser will now preload our image, ready to be swapped in at a later time.

Project

- Open your previous project file, and save it under the name **`chapter_24.html`**.

- Starting with your previous code, create a new function called preload_images in the head area script element of your page.

- Use this function to create seven new image objects, and use each objects corresponding object variable to preload your "over" image variations.

- Check your work in your browser to ensure that the image swapping still works as expected.

- Add your preload_images function to the body tag of your page to ensure that it runs when the page has finished loading. Use the following syntax:

```
<body onload="preload_images();">
```

- Once you have verified that the image swapping still works as expected, expand your preload_images function to define an array of images to preload, and then use a for loop to move through the array and assign each image to the src property of an object variable. ***Hint***: an object variable can be anything that can store information – for example an array element.
- Check your work in your browser.

25 Working with Browser Windows

Key Points

- Using standard HTML links, we can open new browser windows:

```
<a href="#" target="_new">link</a>
```

- The amount of control this method affords us over the resultant image, however, is nil. We cannot control the size, shape, location on the screen or anything else about that window with this method.

- JavaScript allows us much finer control, as we may expect:

```
window.open( page_url, name, parameters );
```

- As we can see from the above prototype, there are only three arguments that this method can take. However, the **`parameters`** argument is actually more complex than we might assume:

```
"param1,param2,param3,param4…"
```

  since we can use it to add many parameters to the method. Note there are no spaces in the parameter list.

- **`name`** is used to give the window an HTML name – so we can use that name to open links into the new window using the "**`target`**" method above.

- The return value from the **open** method is an object variable referring to the newly opened window. In other words, if we open a new window like so:

```
win = window.open("page.html", "", "");
```

we can use the object variable **win** to alter the new window through JavaScript.

- A practical example – let's say we want to open a new window 300 pixels high by 400 pixels wide, and display the BBC news page in it. The following code would suffice:

```
window.open("http://news.bbc.co.uk/", ↲
    "bbc", "width=300,height=300");
```

- Some available parameters are given in the table below:

Parameter	Value	Function
location	Yes/no	Specifies whether or not the location (address) bar is displayed
menubar	Yes/no	Specifies whether the menu bar is displayed
status	Yes/no	Specifies whether the status bar is displayed
width	Pixels	Specifies the width of the new window
height	Pixels	Specifies the height of the new window
resizable	Yes/no	Allow or disallow window resizing
scrollbars	Yes/no	Allow or disallow window scrolling

- If no parameters are set, then the value of each parameter is set to "yes" where appropriate. For example:

```
window.open("http://www.bbc.co.uk/", ↲
    "bbc", "");
```

is equivalent to:

```
<a href="http://www.bbc.co.uk/" target="bbc">
```

- However, if **any** parameter is set, all others default to “no”. So if you wanted to have a scrollbar but nothing else, the following would suffice:

```
window.open("http://www.bbc.co.uk/", ↵
    "bbc", "scrollbars=yes");
```

Project

- Open your previous project file, and save it under the name **chapter_25.html**.
- Remove all functions and HTML from your previous file, leaving only the logo image and the link.
- Create a function in the head area script element called **view_new_logo**. This function should:
  - Open a new window 200 pixels square.
  - Load the image **rocollogo.jpg** from the images folder.
  - Be called **RocolLogo**
  - Be stored in an object variable called **objNewLogo**.
  - Have no scrollbars, be of a fixed size, and have no other features where possible.
- Remove all event handlers from the link, and add a new one to run the function above when the link is clicked.
- Once you have verified that a window pops up as required when the link is clicked, test each parameter from the table above in the function.

26 Positioning Browser Windows

Key Points

- The **screen** object provides you with access to properties of the user's computer display screen within your JavaScript applications.

- Some of the available properties are:

Property	Description
availHeight	The pixel height of the user's screen minus the toolbar and any other permanent objects (eg the Windows Taskbar)
availWidth	As availHeight, but dealing with horizontal space
colorDepth	The maximum number of colours the user's screen can display (in bit format, eg 24 bit, 16 bit etc)
height	The true pixel height of the user's display
width	The true pixel width of the user's display

- The **`left`** and **`top`** parameters of the **`open()`** method enable you to specify the position of a window on screen by specifying the number of pixels from the left and top of the screen respectively.

- If you need to use a variable to specify the value of a parameter in the **`open()`** method, you would do so as follows:

```
window.open("index.html", "window_name", ↵
    "width=200,height=200,left="+var_left+ ↵
    "top="+var_top);
```

Where **`var_left`** and **`var_top`** are the appropriate variables.

- Note the use of the **+** operator to ensure that the third parameter in the **`open()`** method remains as one string when variables are added.

- In order to centre the window on the user's screen, a little simple geometry is required. The centre of the screen is obviously the point found when we take the width of the screen divided by two, and take the height of the screen divided by two. However, if we set the window's top and left values to these coordinates, the top left **corner** of the window will be centred, not the window itself.

- In order to centre the window, we need to subtract half of the window's height from our top value, and half of the window's width from our left value. A simple script to accomplish this is as follows:

```
win_width = 200;
win_height = 200;

win_left = (screen.availWidth/2) ↵
    - (win_width/2);
win_top = (screen.availHeight/2) ↵
    - (win_height/2);
```

- By using this script, the values of win_left and win_top will be set correctly for any window using win_width and win_height appropriately to be centred on the screen.

Project

- Open your previous project file, and save it under the name **`chapter_26.html`**.

- Modify your existing code to ensure that the logo appears centred on the user's screen. If possible, do not modify your original function by doing anything more than two new functions – **`get_win_left( width )`** and **`get_win_top( height )`**.

27 Focus and Blur

Key Points

- The **focus()** method of the **window** object gives focus to a particular window. In other words, it ensures that the window is placed on top of any other windows, and is made active by the computer. For example, if we have created a new window and stored the result in an object variable called **new_window**, we could ensure that the window was brought back to the front at any point after it had been opened by using the following code:

```
new_window.focus();
```

- Conversely, we can use the **blur()** method to remove focus from the specified window – returning focus to the previously selected one as appropriate. Its use is similar to the **focus()** method.

- Both these events have associated event handlers: **onblur** and **onfocus**. However, since they do not have associated HTML tags, how can we attach event handlers to the object?

- It turns out that, within JavaScript, each object has an individual property for each event handler it can have applied to it. For example, if we wanted to add an event handler to an image tag on the page, we could apply either of the following methods to do that:

```
<img src="…" id="test_img" ↵
onclick="do_something();" />

<script>
document.getElementById("test_img").onclick=↵
    do_something();
</script>
```

- So, to attach a function to a window's **`focus`** event, we could use:

  ```
  new_window.onfocus = some_function();
  ```

Project

- Open your previous project file, and save it under the name **`chapter_27.html`**.
- Modify your function to open another window as well as the original one, with the following features:
  - The second window should be called **`oldRocolLogo`**, should be assigned to the object variable **`objOldLogo`**, and display the old logo from the file **`images/rocollogo.gif`**.
  - When opened, the windows should be positioned so that both can be clearly seen.
- Test your modifications at this point.
- Observe the function's action when the windows are opened, then the original window is placed in front of them and the function is invoked again.
- Add a new statement to the function which uses the **`focus()`** method to ensure that when the function is called, both new windows are moved to the top of the "stack" of windows.
- Observe which logo appears "on top" when the function is called. Use the **`focus()`** method again to alter this.

28 Dynamically Created Content

Key Points

- It's quite easy to create a new page on demand using JavaScript. In this context, we are talking about creating a completely new page in a window without loading any file into that window.

- To do this, invoke the **`open()`** method of the **`window`** object, leaving the location parameter empty:

```
new_win = window.open("", "newWin", ↲
     "params…");
```

- Next, remember that you can write HTML code to a page using the window's **`document.write()`** method. Up until now, we have used only the current window's **`document`** object. However, we can specify *which* window's **`document`** we want to manipulate as follows:

```
document.write();  // write to current window
new_win.document.write(); // or new window
```

  For example:

```
new_win.document.write("<html><head>");
new_win.document.write("<title>demo</title>"):
new_win.document.write("</head><body>");
new_win.document.write("<h1>Hello!</h1>");
new_win.document.write("</body></html>");
```

Project

- Open your previous project file, and save it under the name **`chapter_28.html`**.

- Modify your existing script to create a third window with the following properties:
  - The window should be called **`newHTML`**, and be assigned to the object variable **`objNewHTML`**.
  - It should be 400 pixels square.
  - It should not load any page when it is created. It should display the word "Welcome" as an **`H1`** header.
  - It should contain a paragraph with the text "Please decide which logo you would like to choose."
  - The third window should carry the title "Rocol Art"
  - When all windows have been opened, the third window should be **`focus`**sed.
- Check your work in the browser.

29 Working with Multiple Windows

Key Points

- The window object's **`close()`** method enables you to close a window. If you have an object variable in the current window referring to the window you wish to close, you can simply use:

  ```
  new_window.close();
  ```

  to close the window.

- Things are a little more complicated when you wish to close a window from a window other than the one which opened the new window. In order to tackle this, we need to think about **window scope**.

- When we write JavaScript into our code, all functions and variables within that script are available for us to use **within that window**. For example, if we have a function called **`say_hello()`** in our main window, we can easily call that function. However, if we want to call the function from a newly opened window, we cannot call it directly from the new window, as our functions and variables are "tied" to the windows in which they were first defined.

- This is why, when we want to write to any window other than the one containing our JavaScript code, we must access the **`document`** object *of that window* in order to put content in the right place,

- But how about in the other direction? Let's say we use a function in our main window (call it **window 1**) to open a new window (**window 2**). If we store **window 2** as an object variable, we can access all properties of **window 2** from **window 1** by using that object. The question is, how do we access any properties of **window 1** from **window 2**?

- The key is that **window 2** and **window 1** have a special relationship – a “parent/child” relationship. We can access any property of **window 1** from **window 2** by using the special object called **`opener`**. This is an object created within any window that has been opened by JavaScript, and it always refers to the window that opened the new window.

- To illustrate this, let’s consider our previous project. We have three new windows, and one “parent” window. If we wanted to use an event handler on one of the new windows to close on of the other new windows, we would need to first access the parent window, and then access the object variable within the parent window that pointed to the window we wanted to close.

- Let’s say we wanted to close window_2 from a link in window_1. We would have to create an event handler in the link with the following code:

  ```
  onclick=”opener.window_2.close();”
  ```

Project

- Open your previous project file, and save it under the name **`chapter_29.html`**.

- Modify your existing script to achieve the following:

  - Add two new paragraphs to the third window’s content containing the following text:

    The Old Logo

    The New Logo

  - Each line should be contained in a hyperlink whose event handler accesses the parent window’s object variable pointing to the appropriate new window. Its **`close()`** method should then be invoked.

- Check your work in the browser.

30 Using an External Script File

Key Points

- JavaScript can easily save us from having to type lots of HTML. As we have seen, we can use JavaScript to generate large form elements and tables using a small amount of code. This is good news for us, as it makes our work easier. It is also good news for our visitors, as they only have to download a small amount of code for a relatively large amount of content.

- However, we can do better. As it is, if we have one function that will generate a year drop down menu for a form's date of birth section, we need to include that function in every page that requires it. This means that our visitors are downloading identical content multiple times. In addition, if we change or improve our function, we have to ensure that we update that function in every page that has it included.

- HTML allows us to solve this problem by providing a mechanism to load an external text file into the HTML page and treat its contents as JavaScript code. Since it is a separate file, once it has been downloaded once, the browser will not download further copies of the file if it is requested by another page. In other words, we can load all our JavaScript code into one file, and any changes there will instantly be reflected across the entire site.

- To use code in an external file, we still use the **<script>** tag – but with a new attribute, the **src** attribute. This is very similar to loading an image file on to a page:

```
<script language="JavaScript" ↲
    type="text/JavaScript" ↲
    src="s/script_file.js"></script>
```

- Note three things:

- The language and type attributes are essential here.
- The script tag still has a closing **`</script>`** tag.
- We cannot add any further JavaScript between the tags when we are using the tags to load an external file. To add JavaScript to the current page only, we have to use a second set of **`<script>`** tags.

Project

- Open your previous project file, and save it under the name **`chapter_30.html`**.
- Move all your JavaScript function definitions, and any other code in the head section script element to a new file called **`script.js`**.
- Modify your head section script element to load code from the new file.
- Check that your page still works as expected.
- **NOTE**: the **`.js`** file extension is just a naming convention. **`<script>`** tags will load JavaScript from any text file (hence the need to include the type and language attributes). However, it is a widely used convention, and it is worth sticking to in order to keep your code easily understood by anyone who may work on your code in the future.

31 Javascript and Forms

Key Points

- Without JavaScript, the server handles all validating and processing of information submitted via a form. Using JavaScript on the client side of the equation saves time for the user and creates a more efficient process.

- Some processing can be handled by JavaScript (for example, mathematical computations) and JavaScript can ensure that only correct data is sent to the server for processing.

- JavaScript is used in conjunction with server-side processing – it is not a substitute for it.

- To access information in a form we use the document object's **getElementById()** method, as we have previously to access other objects on the page. For example, if we have a form on the page like so:

```
<form id="testForm" … >

</form>
```

  we would access it in JavaScript by using:

```
document.getElementById('testForm');
```

- The resultant object is actually a multi-dimensional array. Each of its elements is itself an array containing information about the elements of the form (text boxes, buttons etc). By properly naming each of the form's elements in the appropriate <INPUT> tags, you can access information relating to each of the form's elements.

- To access the data stored in a text box called **Name** which is included in the form with the id **Enquiry** you use the value property like so:

```
objForm = document.getElementById('Enquiry');

strValue = objForm.Name.value
```

- There are two ways of sending form data to the server. Using the **method** attribute, you can specify either the GET or the POST methods:

```
<form id ="enquiryform" method="GET"…
```

or

```
<form id ="enquiryform" method="POST"…
```

- In general, you should use the POST method if you want to send a lot of data (eg files, large amounts of text) from your form. You should use GET if you want to process search forms etc, as a GET form will be transmitted just like a URL, and is hence "savable" as a bookmark or link.

- In general, you will send your form to a server side script, specified by the form's action attribute:

```
<form id="enquiryform" method="GET"
→ action="process.php"…
```

- When the user clicks on a form's Submit button, without JavaScript intervention, the form's data is sent straight to the server for processing. But you can intercept the data (so you can process it with JavaScript) before it is sent, by including the **onsubmit** event handler in the **<form>** tag. This enables you to run a JavaScript function before the data is sent:

```
<form id="enquiryform" method="GET"
→ action="process.php"
→ onsubmit="functionName()">
```

- In the above example, when the user clicks on the Submit button, the function **functionName()** is run first, then the data is sent to the server.

- When the submit event is triggered by the form's submission, the browser waits to discover what is *returned* from the event handler. By default, the event handler will return **`true`**. However, if the event handler returns a **`false`** value, the form's submission will be aborted, and the page will not be submitted to the server.

- By returning a value of either true or false from your function (**`functionName()`** in the above example), and ensuring that this is also the return value of the **`onsubmit`** event handler, you can decide whether or not the form's data is actually sent to the server.

- You specify that the return value of the function is also the return value of the **`onsubmit`** event handler in the following way:

```
onsubmit="return functionName();"
```

Project

- Open your previous project file, and save it under the name **`chapter_31.html`**.

- Clear any content from the body element of the page, and ensure that the head area script element has no code in it.

- Save a copy of this page as **`processing_31.html`**, and put an **`<h1>`** element in the body area saying "**success**!".

- Now, close your **`processing_31.html`** page and create a form on your original **`chapter_31.html`** page using HTML – if you have difficulty with this, the tutor will provide an example to duplicate. Your form should:

  - have an id of **`jsCourseForm`**.
  - have a single input box with an name value of **`name`**.
  - have a submit button.
  - use the GET method of submission.
  - have **`processing_31.html`** as its **`action`** attribute.

   - have an onsubmit event handler that returns the value of the (as yet non-existent) **`check_form()`** function.

- Now, create a function in the head area script element called check_form(). This function should:

   - use the document's getElementById() method to store a reference to the form's object in an object variable.

   - store the value of the form's **`name`** element in another variable.

   - if the value of the name element is not "**Bugs Bunny**", an alert box should appear stating:

     ```
     Sorry chum!
     Either you misspelled your name…
     Or you haven't got what it takes…
     Try again.
     ```

   - in addition, the form should be prevented from submitting.

- If the user enters the correct name, however, the form should submit without interruption.

- Check your work in your browser.

32 Form Methods and Event Handlers

Key Points

- Each form object (e.g. text, button etc) has a set of properties associated with it. This is different for each form element. The **`value`** property is common to most form elements and is one of the most useful properties.

- You can assign the data stored in a text box called **`Name`** which is included in the form with id **`Enquiry`**, to a **`variable`** like so:

```
variable = document.getElementById('Enquiry').
→ Name.value
```

- Form objects also have methods associated with them. The set of available methods is different for each form object.

- Below is a list of commonly used methods for the text object:

Method	Description
blur()	Removes the focus from the text box
focus()	Gives the focus to the text box
select()	Selects the text box

- Below is a list of commonly used methods for the button object:

Method	Description
blur()	Removes the focus from the button
focus()	Gives the focus to the button
click()	Call's the button's onclick event handler

- Form objects also have event handlers associated with them. The set of available event handlers is different for each form object.

- Below is a list of commonly used event handlers for the text object:

Event handler	Runs JavaScript code when…
onblur	The text box loses the focus.
onfocus	The text box receives the focus.
onselect	The user selects some of the text within the text box..
onchange	The text box loses the focus and has had its text modified.

- Below is a list of commonly used event handlers for the button object:

Event handler	Runs JavaScript code when…
onBlur	The button loses the focus
onFocus	The button receives the focus
onClick	The user clicks the button

- Finally if you are sending data to the server, a submit button is not the only way. You could use the submit() method in a function which is invoked by an event handler. This operates as if the Submit button was clicked:

```
document.getElementById('Enquiry').submit();
```

Project

- Open your previous project file, and save it under the name **chapter_32.html**.

- Modify your form in the following way:

  - Remove the submit button

  - Replace the submit button with a standard form button.

  - Add an event handler to this button to invoke the function in the head area script element.

  - Remove the **onsubmit** event handler from the form element.

- Now, modify the **`check_form()`** function in the following way:
    - If the user types in the name “Bugs Bunny”, the function submits the form using the form’s submit() method.
    - If the user types anything else, the previous alert box warning is displayed and the form is not submitted, but also:
        - The words “please try again” are displayed in the text box.
        - The text box is given focus.
        - The text in the text box is selected.
- Check your work in your browser.

33 JavaScript and Maths

Key Points

- The **`Math`** object is a pre-defined JavaScript object containing properties and methods which you can use for mathematical computation.
- Below is a selection of some useful **`Math`** methods:

Method	Returns
Math.cell()	The smallest integer greater than or equal to a number. That is, it rounds up any number to the next integer. **`Math.cell(2.6)`** returns 3 and so does **`Math.cell(2.2).`**
Math.floor()	The largest integer greater than or equal to a number. That is, it rounds down any number to the next integer. **`Math.floor(2.2)`** returns 2 and so does **`Math.floor(2.6).`**
Math.max(n1,n2)	The larger of the two arguments.
Math.min(n1,n2)	The smaller of the two arguments.
Math.random()	A random number between 0 and 1.
Math.round()	The number rounded to its nearest integer.
Math.sqrt()	The square root of a number.

- You don't need to include a Submit button in a form and you don't need to send form data to the server. You could use event handlers to invoke JavaScript code which merely processes the data on the form (e.g. you may just perform some mathematical computations on some user data).

- If you are not sending the data to the server, there is no need to include either the Action or Method attributes in the <FORM> tag, though by default the Action will usually submit to the current page, and the Method will be set to “get”.

Project

- Open your previous project file, and save it under the name **`chapter_33.html`**.
- Remove all content from the body section of the page.
- Copy the file **`max_wins.html`** from the network, and open it using NotePad’s File > Open command.
- Copy and paste the entire contents of **`max_wins.html`** into your current project file, into the body element of the page.
- Remove all JavaScript code from the script element in the head section of the page.
- Take some time to open your project file in a browser, and study the code. This project will enable the page to:
  - Generate two random numbers when the button is pressed.
  - Display the random numbers in the text boxes marked **player 1** and **player 2**.
  - Compare the two random numbers and display the name of the player whose number is higher in the text box marked **winner**.
- When studying the code, note that the form has no valid action or method attributes. It also has no “submit” button. We are not going to allow the form to submit to the server at all, but are going to use JavaScript to do all our processing on the form.
- Now, modify the HTML code on the page to add an event handler to invoke a new function defined in the head area script element. The function should perform the following operations when the form button is clicked:
  - Place two separate random integers between 0 and 100 in each of the Player text boxes.

  - Find a way to use a Math method to compare the two entered integers. Once compared, the function should then place the appropriate value of **Player 1**, **Player 2**, or **Draw** in the Winner text box.

- Check your work in your browser.

34 Object Variables – A Refresher

Key Points

- Referring to objects can be a lengthy process. Consider the Player 1 text box in the previous example. You refer to it in full as follows:

  ```
  document.getElementById("MaxWins").Player1
  ```

- This notation although precise, is tedious, time consuming and can be prone to error. Luckily, there are some shortcuts which can save time and reduce typing errors.

- We can use an object variable to simplify our work whenever we are in situations where certain objects need to be referred to repetitively. To use an object variable, you begin by simply assigning it a specific object:

  ```
  oPlayer1 =
  → document.getElementById("MaxWins").Player1
  ```

- Once assigned, you can use the object variable in any situation where you would use the specific object itself. Using the object variables from the previous paragraph:

  ```
  oPlayer1.value
  ```

  is the same as:

  ```
  document.getElementById("MaxWins").Player1.value
  ```

- Bear in mind that you assign objects to object variables. You would assign a text box or a button or an image etc to an object variable and then refer to that object's properties as shown above **(oPlayer1.value).** You don't assign text or string values to object variables. So:

  ```
  oPlayer1 =
  → document.getElementById("MaxWins").Player1.value
  ```

  would merely assign the *value* stored in **Player1** to the variable **oPlayer1**.

- Hopefully it is obvious that you don't need to include the 'o' at the beginning of the object variable's name. It's just a convention to help distinguish between the different types of variables.

Project

- Open your previous project file, and save it under the name **chapter_34.html**.

- Modify the function in the head area script element in the following way:

  - All specific object which are referred to more than once are each assigned their own object variable at the start of the function.

  - References to specific objects in the code are replaced by the appropriate object variables.

- Test your work in your browser to ensure that it functions as before.

35 Actions From Menu Items

Key Points

- The HTML **<select>** form tag enables you to create a menu (select box) of options from which the user can choose:

```
<form id="menu">

<select name="Product">
  <option value="one">Image one</option>
  <option value="two">Image two</option>
  <option Value="three">Image three</option>
</select>

</form>
```

- Ordinarily, selecting an option from a select box merely specifies the value of the select box. This is then used for further processing – either by JavaScript or by the server.

- You can use JavaScript to invoke actions based on the current value of a select box (the selected option). These actions might be directly loading another page or performing some other type of processing.

- Another name for this type of action is a “jump menu”.

- In order to do this, you need to know that each select box on a form has a select object associated with it. Using the above example, you can therefore access the current value of the Product select box (ie the value of its currently selected option) which is located on the ProductMenu form, in the following way:

```
document.getElementById("menu").Product.value
```

- As you would expect, the select option has methods and event handlers associated with it:
  - The focus() method gives the focus to a select box.
  - The blur() method removes the focus from the select box.
- The available event handlers for the select object are given below:

Event handler	Runs JavaScript code when….
onblur	The select box loses the focus
onfocus	The select box receives the focus
onchange	The select box has had its value modified

- The way to invoke action(s) when the user selects an option from a select box is to prepare a JavaScript function which will be invoked using the select object's **`onchange`** event handler:

```
<select name="name" onchange="functionname()">
```

Project

- Open your previous project file, and save it under the name **`chapter_35.html`**.
- Clear the head section script element of JavaScript, and remove all content from the body area of the page.
- Create a form in the body area of the page. Give the form an id of **jumpMenu**.
- In the form, place a select element. Give the select element the following values and labels eg:

```
<select name="menu">
   <option value="value">label</option>
   …
</select>
```

Value	Label
http://www.bbc.co.uk/	The BBC
http://www.google.com/	Google
http://www.hotmail.com/	Hotmail

http://www.ed.ac.uk/	The University of Edinburgh
http://www.apple.com/	Apple Computer
http://www.microsoft.com/	Microsoft Corporation

- Finally, add an option element to the beginning of your menu like so:

  <option value="">Select a destination</option>

- We are now going to use the menu's **onchange** event handler to invoke a function we are about to define. The function is to be called **jump_to_page()**.

- Define such a function in the head area script element. The function should:

  - Create an object variable referencing the form element that represents the menu.

  - Get the value of the menu at that moment.

  - If the value is not equal to "" (ie, if a valid selection has been made), the function should use the use JavaScript to load the selected page into the browser.

- Check your work in your browser.

36 Requiring Form Values or Selections

Key Points

- Form data may be invalid if it is sent to the server without certain information – for example, if the user has omitted to select an item from a menu. Better not to bother processing, than to waste the time of the user and server by trying to process the invalid information.

- In the case of a selection box, one way of validating is to include a null value for the default option. In the previous project, the default value of the selection box (always the first option unless specified otherwise) was "". If the user doesn't actively make another selection, then you can check the value of the selection box before sending it to the server.

- Another example is the case of a set of radio buttons. Imagine that for an imaginery company, an order form contained two form elements – a selection box to specify which product was being ordered and two radio buttons to specify whether it was being ordered as a photographic print or as a slide.

- Let's say the id of the order form is **`OrderForm`** and the name of each radio button is **`Format`**. (Remember that in HTML, radio buttons in a set are related to each other by their name that must be the same for each related radio button).

- The radio object is an array where each element of the array stores information relating to each of the radio button objects (remember, counting starts at 0).

- As the name is identical for each radio button in the set, you can't access an individual radio button by using its name. But you can access it using the standard array notation:

  ```
  oOrderForm.Format[i]
  ```

  where in this case, I is an integer between 0 and 1, and **oOrderForm** is an object variable pointing to the form element.

- Radio buttons in a set each have a checked property that stores a Boolean value specifying whether or not a radio button is checked. Obviously, you can access this value in the following way:

  ```
  oOrderForm.Format[i].checked
  ```

- So, to verify that in a set of radio buttons, at least one of them is checked, all you have to do is loop through each of the radio buttons using the array's length property to specify the number of iterations of the loop:

  ```
  FormatSelected = false;
  oOrderForm =
  → document.getElementById("OrderForm");

  for ( i=0; i < oOrderForm.Format.length; i++ )
  {
      if ( oOrderForm.Format[i].checked )
      {
          FormatSelected = true;
      }
  }
  ```

- In our order form example, one of the products might only be available as a slide. You could use the **onchange** event handler of the **select** object to invoke a function which automatically sets the value of the relevant radio object's checked property:

```
function SetFormat ()
{
    oOrderForm =
    → document.getElementById("OrderForm");
    if ( oOrderForm.Product.value ==
    → "Greek Boat" )
    {
        oOrderForm.Format[0].checked = true;
    }
}
```

where **Product** is the name of the select box, **Format** is the name of the group of radio buttons and the value of the first ratio button is Slide.

- Finally, to reset all form objects to their initial state, use the reset() method:

```
oFormObject.reset()
```

Project

- Open your previous project file, and save it under the name **chapter_36.html**.
- Remove all content from the body section of the page.
- Copy the file **order_form.html** from the network, and open it using NotePad's File > Open command.
- Copy and paste the entire contents of **order_form.html** into your current project file, into the body element of the page.
- Remove all JavaScript code from the script element in the head section of the page.
- Take some time to open your project file in a browser, and study the code.

- Add an event handler to the form element on your page that will invoke a function called **`check_form()`** when the form is submitted. This function will return **`true`** or **`false`** depending on whether the form passes a number of tests as described below. Remember to precede the event handler's function call with **`return`** to ensure that the form awaits confirmation from the function before proceeding.

- Create the **`check_form()`** function in the head area script element. The function should perform the following steps:

  - If the current value of the selection menu is "":

    - Display an alert with the message "**please select a product**"

    - Give focus to the menu

    - Return **`false`**

  - If no radio buttons are selected:

    - Display an alter with the message "**please specify a format**"

    - Return **`false`**

  - Otherwise, return **`true`**

- Check your work in your browser.

- The **Greek Fishing Boat** is only available in **Slide** format. Add an event handler to the select element which will run a function called **`set_format()`** when its value is changed.

- Create the function **`set_format()`** in the head area script element. The function should check first what the value of the menu is. If it is the **Greek Fishing Boat**, it should then check to see if the **Slide** radio button is checked. If it is not, then the function should correct this.

- Check your work in your browser.

- Modify your check_form() function to check that, if the value of the menu is Greek Fishing Boat, then the Slide radio button is checked. If not, it should report the error as before.

- Finally, note that your **check_form()** function is currently not very efficient, in that it will only report one error at a time. This can be tedious for an error prone user, and it is much better practise to observe a form for *all* errors simultaneously.

- Create a variable at the beginning of the function called **error**. Set the value of this variable to "".

- Instead of using an alert box each time an error is found, add the error message to the end of the variable, eg

  ```
  error += "error message\n";
  ```

  (note the new line code at the end of each message)

- After all checks have been made, if we have caught any errors, the value of **error** will no longer be "". Thus, we can use the following code to report all errors at once:

  ```
  if ( error != "" )
  {
      alert("The following errors were found: \n\n"
          + error);
      return false;
  }
  else
  {
      return true;
  }
  ```

- Modify your function to be more efficient.

37 Working with Dates

Key Points

- The date object stores all aspects of a date and time from year to milliseconds.

- To use a new date object, you must create it and assign it to an object variable simultaneously. The following code creates a new date object which stores the current date and time and assigns it to the variable **dtNow**

```
dtNow = new Date();
```

- You can specify your own parameters for the date object when you create it:

```
theDate = new Date(
    year, month, day, hours,
    minutes, seconds, mseconds
);
```

- Note that months are represented by the numbers 0 to 11 (January to December). Days are represented by 1 to 31, hours by 0 to 23, minutes and seconds by 0 to 59 and milliseconds by 0 to 999.

- The following code stores 17th July 2004 at 9:15:30pm in the variable **theDate**:

```
theDate = new Date(2004,6,17,21,15,30,0);
```

- The first three parameters are mandatory, while the rest are optional:

```
theDate = new Date(2004,6,17);
```

  gives you midnight of the date above.

- You can assign a date to a variable using a string in the following way:

```
theDate =
  new Date("Sun, 10 Oct 2000 21:15:00 – 0500);
```

- Everything from the hours onwards is optional, and in practical terms the day is not necessary either. For example:

```
theDate = new Date("10 Oct 2000");
```

  is a valid date.

Project

- Open your previous project file, and save it under the name **chapter_37.html**.
- Remove all content from the body section of the page.
- Create a function in a head section script element that displays an alert box containing the current date and time. Have this function called from a body section script element.
- Now modify your script so that it also writes the date and time 17th July 1955 1:00am to the page in standard date and time format. Do this by entering the appropriate parameters into the **new Date()** constructor.
- Now, using parameters once again, add a new line to your script that will display midnight on 17th July 2004 on a separate line under the existing date.
- Finally, using the string approach in the date object, add a new line to your script which will display midnight on 31st December 1999 on a separate line under the existing information and in standard date and time format.

38 Retrieving Information from Date Objects

Key Points

- Below is a selection of some useful methods which enable you to retrieve some useful information from date objects:

Method	Returns
getDate()	The day of the month (1-31)
getDay()	The day of the week (0 = Sunday, 6 = Saturday)
getFullYear()	The year as four digits
getHours()	The hour (0-23)
getMilliseconds()	The milliseconds (0-999)
getMinutes()	The minutes (0-59)
getMonth()	The month (0-11)
getSeconds()	The seconds (0-59)
getTime()	The date and time in milliseconds – also called Unix Time

- JavaScript stores date and time information internally as the number of milliseconds from 1st January 1970. This is common to most programming languages, and is actually the way most computer systems store time information.

- For example:

  ```
  dtNow = new Date();
  document.write(dtNow.getTime());
  ```

  writes the number of milliseconds that have passed since 1st January 1970.

- Being aware that there are 1000 milliseconds in a second, 60 seconds in a minute, 60 minutes in an hour and 24 hours in a day, you can establish the number of days (or hours or minutes etc) between two dates by subtracting on date in millisecond format from the other in millisecond format. To achieve the units you require, perform the appropriate division (so for the number of minutes, divide the result of the subtraction first by 1000, then by 60), and then use Math.floor() on the result to round the number down as required.

Project

- Open your previous project file, and save it under the name **`chapter_38.html`**.

- Clear any JavaScript from the page's script elements.

- Create a function in the head section script element that performs the following:

  - Create two arrays, one storing the days of the week ("Sunday", "Monday" etc), the other storing the months of the year ("January", "February" etc)

  - Use a new date object, along with the arrays and the appropriate date methods to write today's date to the page in the following format:

    Today it is: dayName, month, dayNumber.

    for example:

    Today it is Tuesday, October 17.

  - Below the date, write the time in the following format:

    It is currently hh:mm am (or pm)

    Note: *to convert from the 24hr clock, subtract 12 from any value over 12, and replace a zero value with 12. Anything greater than or equal to twelve should receive a "pm" suffix.*

  - Below the time, write the number of days since the start of the millennium in the following format:

    It is n days since the start of the millennium.

- Finally, call your function from the body area script element on your page, and check your work in your browser.

39 Creating a JavaScript Clock

Key Points

- **`setTimeout()`** is a method of the window object. In its common form, it enables you to run any JavaScript function after an allotted time (in milliseconds) has passed. For example:

  ```
  window.setTimeout("functionName()", 1000)
  ```

  will run **`functionName()`** after one second. Note the quote marks!

- You can have any number of "timed out" methods running at any one time. To identify each "timeout", it is a good idea to assign the return value of each one you create to an object variable:

  ```
  firstTimeout =
    window.setTimeout("functionName()", 1000)
  ```

- The most common reason to track timeouts is to be able to cancel them if necessary – for example, you may wish a window to close after 5 seconds unless a button is clicked:

  ```
  firstTimeout =
     window.setTimeout("window.close()", 5000);
  ```

  adding the following code to the button's **`onclick`** event handler:

  ```
  window.clearTimeout(firstTimeout);
  ```

  will do the trick.

- Ordinarily, it is a bad idea to “recurse” a function – to have a function call itself. For example, imagine the annoyance of the following:

  ```
  function annoy_me( )
  {
      window.alert(“BOO!”);
      annoy_me();
  }
  ```

  This will potentially keep popping up alert boxes, and perhaps even lock up the user’s computer!

- However, you may wish a function to use a **`setTimeout`** method to call itself periodically. Imagine a function that calls itself every second to check on the time, and then displays the result in the same place. In effect, this could be seen as a digital clock.

- If you want to clear the clock’s timeout – which in this case would stop the clock – the timeout must already be in operation. You can’t clear a timeout that doesn’t yet exist. So, you have to check first whether a timeout is in operation. One way of doing this is to set a Boolean variable to true whenever the clock is started. By checking this variable before you attempt to clear the timeout, you know whether or not the clock is running and therefore can be stopped.

- To show an am/pm clock you obviously need to carry out the conversion you created in the previous chapter’s project.

- But, so as the display doesn’t show a “moving” effect at different times, you need to consider the situation where either the minutes or seconds on the clock are fewer than 10. For example, consider:

  `10:59:59 am`

  changing to:

  `11:0:0 am`

- In this situation, we need to concatenate an extra “0” on to the front of the actual value to produce:

  11:00:00 am

  as we might expect.

- We can use a shortened form of the if conditional to make this simple. For example:

  s = theDate.getSeconds;
  s = (s < 10) ? “0” + s : s;

- What is happening here? First, we get the current value of the seconds and store that in a variable called “s”. Next, we reset the value of s depending on the condition in the brackets. The prototype of this form of the if conditional is:

```
var = ( condition )
    ? value if true
    : value if false;
```

  in other words, we can think of the ? as being like the opening brace { of an if conditional, the : as being the } else {, and the ; as being the closing brace. The result of the conditional test is then stored in var.

  Note that this has been split over three lines to aid reading. In practice (see above) we can place this on one line for ease of use.

Project

- Open your previous project file, and save it under the name **chapter_39.html**.

- Remove all content from the body section of the page.

- In the body section, create a form input element with the id **clockBox**, and two form input buttons labelled Start and Stop.

- In the head section scrip element, create two empty functions – **start_clock()** and **stop_clock().**

- Before the two function definitions, as the first statement of the script element, create a variable called **timer**, and assign it a value of **null**.

- Immediately below that statement, create a variable called timer_running. Assign it a value of false.
- We will use these variables to track the clock's status.
- Now, add statements to the start_clock() function that will:
  - Get the current time.
  - Format the current time as hh:mm:ss am/pm (as appropriate).
  - Display the formatted time in the text field on the page.
  - Create a timeout which will run the **`start_clock()`** function again in half a second, and assign that timeout's return object to the **`timer`** object variable.
  - Set the **`timer_running`** variable to **`true`**.
- Add statements to the **`stop_clock()`** function that will:
  - Check to see if the clock is running.
  - If it is, clear the timer timeout and set **`timer_running`** to **`false`**.
- Finally, add event handlers to the two form buttons to ensure that the one labelled Stop calls the function **`stop_clock()`** when clicked, and the one labelled Start calls the function **`start_clock()`** when clicked.
- While this should work (test your code!), you'll notice that the output is a little ugly. Displaying the time in a text field is not the most unobtrusive way to show a clock on a page.
- Luckily, we can modify our code very slightly to achieve something much more professional.
- Replace your input field with the following:

```
<span id="clockBox"></span>
```

- **`<span>`** is an HTML tag that allows you to mark areas of content without any semantic meaning – eg, the clock is not a paragraph or a header, so we don't want to label it as such.

- With JavaScript, we can control the content of just about any element on the page. In our previous example, we used **getElementById** (hopefully!) to obtain a reference to the input field, and then altered its **value** property to show our clock.

- Since we are using the same id value here, we do not have to alter our function too much. However, the **getElementById** method now returns a different type of object – one that has no **value** property.

- However, all content tags in HTML (like **<p>**, **<h1>**, **<div>** etc) have a special property when they are returned as JavaScript objects which refers to their text content – the property is **innerHTML**.

- We can use this to alter the content of the tags. For example, if our previous input-field solution had the following code:

  ```
  clk = document.getElementById("clockBox");
  clk.value = formatted_time;
  ```

  where **formatted_time** is a variable containing the formatted time as required, then replacing it with this code:

  ```
  clk = document.getElementById("clockBox");
  clk.innerHTML = formatted_time;
  ```

  will allow us to use the modified **<span>** element in place of the **<input>** element.

- Try adapting your code to use this method of displaying content on the page.

Introduction to PHP

Lesson 1: **Introduction**
- Working in CodeRunner
  - Creating a File
  - Managing your Files
- Four characteristics of PHP
  - 1. PHP is a server-side language, with HTML embedding.
  - 2. PHP is a Parsed language.
  - 3. PHP works jointly with SQL.
  - 4. PHP is part of the LAMP, WAMP, and MAMP stack.

Lesson 2: **PHP Basics**
- PHP Delimiters and Comments
- Variables in PHP
- Modifying Variables and Values with Operators
- Superglobals
  - $GLOBALS
  - $_SERVER:
  - $_GET
  - $_POST

Lesson 3: **Decisions**
- Comparison Operators and Conditions
- IF and ELSE Control Structure
- Logical Operators
  - A Brief Preview of Forms

Lesson 4: **Multiple Control Structures and Loops**
- Multiple Control Structures
- WHILE and FOR Loops

Lesson 5: **Functions**
- Creating Code Reusability with Functions
- Function and Variable Scopes
- Using Functions with Parameters and Return Values
  - Sneaking In with Parameters
  - Sneaking out with Return Values
  - Multiple Parameters and Default Values

Lesson 6: **Arrays**
- Creating an Array
- Associative Arrays
- Creating Multi-Dimensional Arrays
- Traversing and Manipulating Arrays
  - Traversing Associative Arrays with list() and each()
- More built-in functions

Lesson 7: **Strings**

- What's a String Anyway?
- Manipulating Strings
  - Other nifty string shortcuts
- Built-in String Functions
- Regular Expressions
  - Character Ranges and Number of Occurrences
  - Excluding Characters
  - Escaping Characters

Lesson 8: **Fixing Broken PHP**

- Things Professors Don't Talk About Enough
- Debugging Tips
  - Utilizing Error Messages
  - Riddle-Me-This Error Messages
  - Errors without Error Messages
  - Logical Errors
  - Infinite Loops, Infinite Headaches
- Notes on Scalable Programming
  - Before you Code, Pseudocode
  - Make your Program Readable
  - Comment Until You're Blue in the Face
  - Code in Bite-Size Chunks
  - Debug as You Work
  - Reuse Functions as Much as Possible
  - Utilize Available Resources

Lesson 9: **Forms in PHP**

- Forms Review
- Using Superglobals to Read Form Inputs
- Extracting Superglobals into Variables
- Nesting Variable Names

Lesson 10: **Utilizing Internet Tools**

- Environment and Server Variables
- Using HTTP Headers
- Manipulating Query Strings
  - Customizing specific error messages
- Sending Emails

Lesson 11: **Date and Time**

- Date and Time Standards
- Date and Time Functions
  - Constructing Dates and Times

Lesson 12: **Using Files**

- Including and Requiring Files
- Reading and Writing Files

Allowing Users to Download Files

Lesson 13: **Cookies and Sessions**

Using Cookies

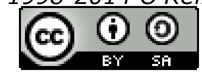
Knowing the User Through Sessions

Deleting Sessions

Lesson 14: **Final Project**

Final Project

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Introduction

Welcome to the O'Reilly School of Technology **Introduction to PHP** course!

In this PHP class, you will learn basic to intermediate programming aspects of PHP--hypertext preprocessor. PHP is a versatile server-side programming language that works hand-in-hand with front-end web languages such as HTML and JavaScript. PHP can be used to create all types of dynamic web interfaces, and because of its open-source robustness, has become one of the most widely used programming languages for the internet.

Course Objectives

When you complete this course, you will be able to:

- develop web applications using basic PHP elements such as delimiters, control structures, operators, variables, arrays, and functions.
- manipulate dates and strings using built-in PHP functions and regular expressions.
- debug and improve code for better reusability and scalability.
- create dynamic web forms using internet tools such as input, environment and server variables, HTTP headers, and query strings.
- read, write, manage and download files through PHP-based web applications.
- track user information using cookies and sessions.
- build a full-fledged shopping cart system.

From beginning to end,you will learn by doing your own PHP based projects. These projects, as well as the final project, will add to your portfolio and provide needed experience. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove will look like this.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

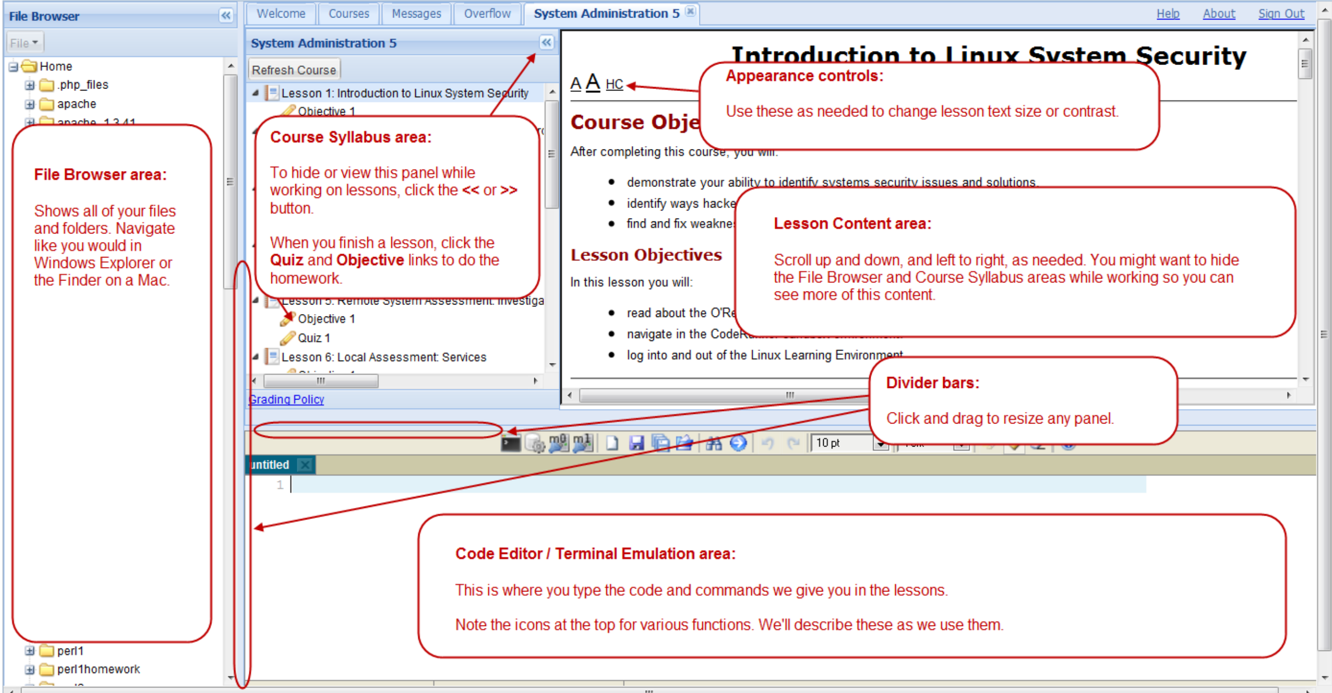
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

File Management Demo

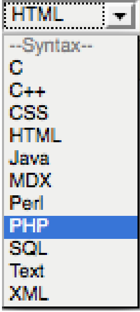
Code Editor Demo

Coursework Demo

Working in CodeRunner

Since CodeRunner is a multi-purpose editor, you need to make sure you're using the correct **syntax**. In this course, you will be using HTML and PHP. To start using HTML, choose the **HTML** option:

To change to PHP, choose the PHP option:



Creating a File

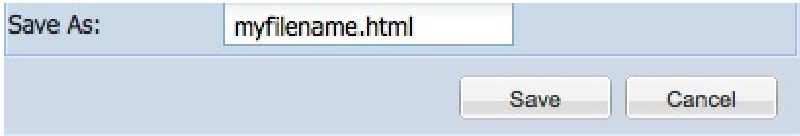
Let's create a file now. Select the HTML syntax and type the code as shown below.

Make sure you're using HTML syntax and type the following into CodeRunner:

```
Four characteristics of PHP:

<ol>
 <li> PHP is a server-side language with HTML embedding.</li>
 <li> PHP is a parsed language.</li>
 <li> PHP works hand-in-hand with SQL.</li>
 <li> PHP is part of the LAMP stack.</li>
 </ol>
```

Managing your Files

Click the button. In the Save As text box, type **fourfacts.html** (be sure to include the html extension when you Save html files).



You can also use the **Save As** ( ) button to save a file with a different name. Try it now with the name **fourfacts2.html**. Note that you are now editing fourfacts2.html, not fourfacts.html.

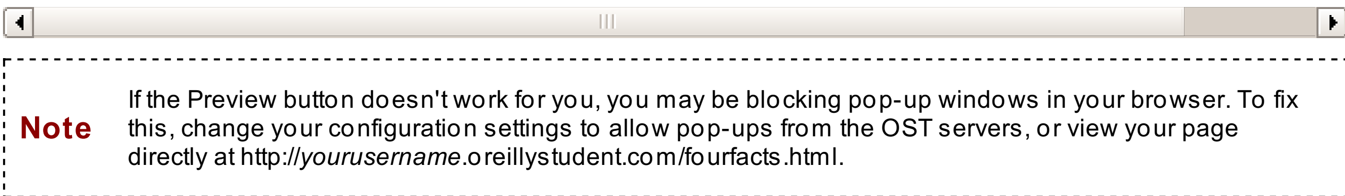
After you successfully save your file, anybody can go on the web, type the URL (http://*yourusername*.oreillystudent.com/fourfacts.html) in the location bar of their browser, and see this page.

To retrieve the original **fourfacts.html**, click the **Load File** ( ) button or double-click the file name in the **File Browser** window.

Four characteristics of PHP

Look again at the HTML top-four list you just typed into CodeRunner, and click Preview:

Note Keep in mind that every time you Preview a file, your changes will be saved. Think about whether you want the previous code overwritten or not. If not, use Save As before you Preview.



Note If the Preview button doesn't work for you, you may be blocking pop-up windows in your browser. To fix this, change your configuration settings to allow pop-ups from the OST servers, or view your page directly at http://*yourusername*.oreillystudent.com/fourfacts.html.

This example serves more than one purpose for us. It demonstrates how to use CodeRunner and it introduces some keys to using PHP. Of course there's much more to PHP than this, but let's start with this.

1. PHP is a server-side language, with HTML embedding.

On the web there are two sides to everything: the Client Side and the Server Side. The Client side is the side you are on right now. It consists of your computer and your web browser. The server side is the side where the web pages are stored and where programs are executed to build dynamic web pages with PHP.

Still have your HTML list? It's time to convert it to PHP. **Switch CodeRunner to PHP**, and retype the top four list into the editor. Then add the blue code below:

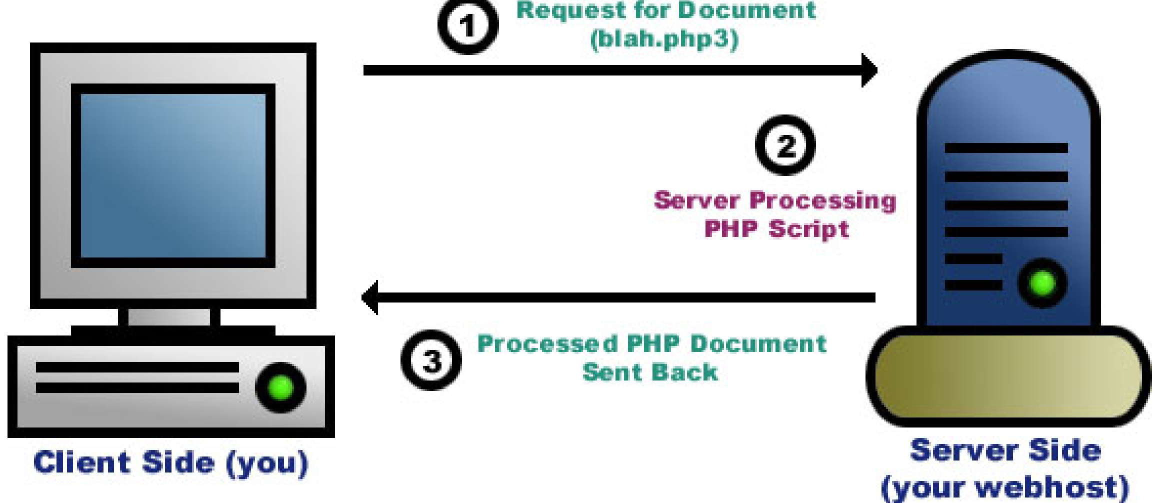
Make sure you're using PHP then type the following into CodeRunner:

```
<?php echo "Four characteristics of PHP:"; ?>

<ol>
<li> PHP is a server-side language with HTML embedding. </li>
<li> PHP is a parsed language. </li>
<li> PHP works hand-in-hand with SQL. </li>
<li> PHP is part of the LAMP stack. </li>
</ol>
```

Click **Preview**. This time save with the php (.php) extension. It looks exactly the same, right? But something more happened this time on the back end.

You see, HTML is a *Client side* language. When you clicked **Preview** while in **HTML**, the Sandbox simply asked your browser to process the HTML tags without any outside help.

Conversely, PHP is a *server side* scripting language and builds HTML dynamically before sending to your browser. Here's a diagram of how PHP works:



When you used **Preview** after adding the PHP code while using **PHP** syntax, the Learning Sandbox:

- Took your code back to your Lab Account on our web server
- *Parsed* it using the **PHP Engine** that's installed within your account
- Returned the results to the browser as HTML

Then your browser rendered the HTML to make it look pretty. Did you notice how the addition of PHP code at the top of the file did nothing to change the HTML list below? This is because the HTML is *embedded* into the PHP file, and doesn't require anything else to output it.

2. PHP is a Parsed language.

The fact that PHP is a parsed language as opposed to a compiled language is a technical concern and probably only interesting to programmers with experience in *Compiled* programming languages like Java or C++. Those languages perform an additional task called compiling that turns the text from the program into a form the computer understands. A binary file is created that serves as the thing that gets executed when a program is running.

PHP is a **Parsed** language, meaning that you can see the results of your code immediately after saving the file, without any compiling or linking steps in between. That's because the compiled PHP engine installed on your account takes the PHP file you've created and "parses" it and uses the commands you created to make the server do something. All the work is still done by a compiled program, but the program you created doesn't have to be compiled, since it just tells the compiled program what to do.

For the geeks out there, this is similar to an **Interpreted** language such as Perl; however, the parsing process has been optimized to use a combination of interpreting and compiling at **run-time**, enabling PHP to be powerful AND fast.

The bottom line is the parsing action of PHP makes your life easier. If you want to know more about parsed, interpreted, and compiled languages, here's a good link.

3. PHP works jointly with SQL.

Let's look at your first PHP script again and add one more little piece of code. Don't worry yet about what the code means, at this point we're just playing around.

Type the following (in BLUE) into CodeRunner:

```
<?php echo "Four facts about PHP:"; ?>
<ol style="font-size:16px;">
<li> PHP is a server-side language with HTML embedding. </li>
<li> PHP is a parsed language. </li>
<li> PHP works hand-in-hand with SQL. </li>
<? printf("MySQL client info: %s\n", mysqli_get_client_info()); ?>
<li> PHP is part of the LAMP stack. </li>
</ol>
```

Click **Preview**. Now you should see the version of **MySQL** library that's included with your account's PHP engine, embedded within your HTML list. You'll learn a lot more about the MySQL database in later courses, but for now just roll with it.

PHP makes it easy to add database-driven content to any website. It supports popular database systems - MySQL, PostgreSQL, Oracle, and others - with libraries of built-in **functions** like the one you added above. These libraries can be referred to by the acronym **DBI**: Database Interface.

Other programming languages such as Perl contain their own sets of DBI libraries too. However, unlike Perl, PHP was designed with database-driven websites in mind, and has become so closely intertwined with MySQL that the two organizations now work together to ensure continued reciprocal support.

Here's a good O'Reilly article about PHP and MySQL.

4. PHP is part of the LAMP, WAMP, and MAMP stack.

What's a **(L|W|M)AMP Stack**?

It's yet another acronym.

- **L**inux, **W**indows, **M**ac
- **A**pache
- **M**ySQL
- **P**HP (or **P**erl, or both)

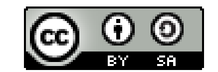
The **Stack** part refers to a group of technologies which, when used together, create powerful and dynamic web applications. There are competing stacks, such as Microsoft's .NET framework and Sun's Java/J2EE technologies. However, corporations are realizing more and more that the free, open-source LAMP Stack can be just as powerful, safe, and lucrative for their businesses as the expensive, proprietary competitors.

And by the way, lucky you! You have all the LAMP technologies you need at your fingertips *RIGHT NOW*:

- Your Learning Lab account is on a **Linux RedHat** server.
- It's equipped with its own **Apache** web server.
- The Apache server has **MySQL** installed on it.
- It also has **PHP** *AND* **Perl** on it.

Alright, you're doing great so far! Don't forget to **Save** your first PHP file (**call it "first.php"**), and work on this lesson's assignments on the syllabus page. Be sure to read the comments on each project or quiz using the "Graded" link. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.


This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

PHP Basics

Welcome back! In the next three lessons, we'll be playing around with a make-believe program to help demonstrate a few programming concepts. While the faux-program isn't one you'd likely create while on the job, the concepts and techniques used are the same. Let's get started!

PHP Delimiters and Comments

Open the file **first.php**. Or, if you're completely sick of the Top Four list, start a new file. Make sure you're using PHP.

Type the following green and blue code into your chosen file in CodeRunner:

```
<ol style="font-size:16px;">
  <li>
    PHP is a server-side language, with HTML embedding.
  </li><br/>
  For instance:<br\>
  <ul>
    <?php
      echo "<li style='color:blue;'>
        This PHP code is INSIDE the PHP delimiters
      </li>";
    ?>
    <li style="color:green;">
      This HTML code is OUTSIDE the PHP delimiters
    </li>
  </ul>
</ol>
```

Now click **Preview** to see the results. What happened?

It should look something like this:

1. PHP is a server-side language, with HTML embedding.
   For instance:
   - This PHP code is INSIDE the PHP delimiters.
   - This HTML code is OUTSIDE the PHP delimiters.

PHP code is separated from embedded **HTML** with **delimiters**. The delimiters are the **<?php** and the **?>**. All PHP is written between these delimeters. An open delimiter **<?php** must have a closing delimiter **?>**. Try taking the delimiters away and see what happens.

Take a look at the PHP - notice that between the delimiters (**<?php and ?>**), the word **echo** showed up. That's a PHP command that means "make this show up." Without echo, it won't show up. Remove the echo command and check out the results.

Here the **echo statement** is used to return text to the web browser from a PHP script. The "echo" command just means "print this out." And one more thing, all statements in PHP must end with a semicolon (**;**).

By the way, why did we put in those slashes (**//**) in the sample code above? Well, those are called "comments." Commenting is a common practice in programming. Commenting records the specific reasons you had for writing the program a particular way.

Type the blue text below into CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is a server-side language, with HTML embedding.</li>
    <br/> For instance: <ul>

    <?php
      //Jerry says, "What's the deal with this line of PHP code?"
      /* Elaine says, "I want to talk about this line, that line, AND the other line!"
*/
      #George says, "...yadda yadda yadda..."
      echo "<li style='color:blue;'>
        This PHP code is INSIDE the PHP delimiters
      </li>";
    ?>
    <li style="color:green;">
      This HTML code is OUTSIDE the PHP delimiters
    </li>
  </ul>
</ol>
```

Now click **Preview** to see the results. Notice what ISN'T printed out on the screen. Our results look exactly the same as before.

What happened to the entire conversation we added?

Ah, Jerry, Elaine, and George, always commenting on everything, yet doing pretty much nothing. In fact, they behave a lot like **comments** in PHP, but in PHP, it's a developer who comments on the code without impacting the results at all.

Note Two slashes (**//**) or one pound sign (**#**) will "comment out" the line that follows it. Or, you can comment out multiple lines by surrounding them with **/*** and ***/**. Play around a bit to get the hang of it.

Comments may not seem like a big deal (they didn't to me at first either), but as our programs become more complicated, it's useful to have reminders (in your own words) of the specific reasons you chose to write your code a certain way. Also, comments are essential for reusing and sharing code. They allow other developers to decipher and understand the specifics of the code you've written.

Let's experiment to discover the answers to these questions:

- Can **delimiters** share the same line as the PHP code?
- Can they share the same line as HTML?
- Can multiple **statements** share *one* line?
- Can **statements** span *multiple* lines?
- Does it matter whether you use **<?php** or just **<?** ?
- Can you mix a line of code with a comment?

Variables in PHP

Every programming language has **variables**. Variables are places to store things. You can insert values into variables and you can extract them. Variables are a lot like dresser drawers. You can put things in and then take them out, and you usually know what each drawer contains.

Add the following blue text to your file in CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack.</li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
  ?>
</ol>
```

Now Preview that.

Not much happening, right? That's because we forgot to *echo* something. Let's go ahead and do that so we get an output.

Add the following blue text to your file in CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack.</li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
    echo $lamp_a;
  ?>
</ol>
```

Now you should see the word **Apache** printed on your page. Why do you suppose that is? After all, we wrote **Echo $lamp_a;**, you'd think that would be the word that printed. Well, as it turns out, **$lamp_a** is a variable. Variables in PHP always begin with a **$**. And, as if **$lamp_a** was a drawer, we put something in it, we inserted **= "Apache"**. Then, in order to get something out of the variable **$lamp_a**, we "echoed" by adding **echo $lamp_a**. Finally, "Apache," the value we put into the variable (or drawer) in the first place was printed out. They are called "variables" because the value can *vary*. We can change the contents of the variable at anytime, and that makes them very useful for storing and retrieving values dynamically.

Just like you can put different items in the appropriate drawers of your dresser--you might have a sock drawer for your socks, a shirt drawer for your shirts, etc.--you can put different kinds of values in variables. In the example above, we've entered words into our variables. In programming, these words are called **strings** because they comprise a string of letters or characters.

Let's put different kinds of values into some other variables.

Add the following blue text to your file in CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack.</li>
  <?php
    $lamp_l = "Linux"; $lamp_a = "Apache"; $lamp_m = "MySQL"; $lamp_p = "PHP";
  ?>
  /* Here are some imaginary numbers for a possible salary package associated with the
skills we're learning in this course (play along!): */
  <?php
    $base_salary = 158470;
    //whoa. we hit the jackpot
    $bonus = 25815.25;
    $benefits = 0.2;
    //percentage of total
    $time_off = 6476; //in dollars
  ?>
</ol>
```

Here we're *assigning* different kinds of **values** (like **158470**) to **variables**. In this context, *assigning* simply means to "fill" the variable with a value.

We've *assigned* the various PHP variables values of three basic **types**: **integer**, **floating point (decimal)**, and **string**. Integers are whole numbers, including negative numbers and 0. Floating point numbers are numbers that may have a decimal point. Strings, as we already mentioned, are simply successive *strings* of characters.

Below is a list of the **types** we've assigned to the variables listed:

Variable Name	Assigned Value	Value Type
$lamp_l	"Linux"	string
$lamp_a	"Apache"	string
$lamp_m	"MySQL"	string
$lamp_p	"PHP"	string
$base_salary	158470	integer
$bonus	25815.25	float
$benefits	0.2	float
$time_off	6476	integer

Let's go back to our dresser analogy for a moment. It will help us to understand the difference between "strongly typed" programming languages and languages like PHP, which are not strongly typed. Programming languages that are *strongly typed* require you to decide the types of variables you're going to have upon creating your files. Once you've created your variables, you are committed. They cannot be revised. It's like labeling your drawers, so that you can only put socks in your sock labeled drawer and shirts in your shirt labeled drawer. And after you've labled these drawers, you can't change them. PHP, however, is not strongly typed. Therefore, the variables remain flexible, we are allowed to change them, and we can put any type of information into a PHP file that we wish.

Since PHP is NOT a **strongly typed** programming language, the following won't break your script:

Type the following blue text below into CodeRunner:

```
<ol style="font-size:16px;">
<li> PHP is part of the LAMP stack. </li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
  ?>
  <?php
    /* Here are some imaginary numbers for a possible salary package for the skills we'
re learning: */
    $base_salary = 158470;
    $bonus = 25815.25;
    $benefits = 0.2;
    //percentage of total
    $time_off = 6476;
    //in dollars
    $benefits = "this is a string now"; // You just changed the value of $benefits from
 a float number to a string!
  ?>
</ol>
```

Go ahead and Preview it. Our variables are shy creatures! So far they've been hiding like comments when we **Preview**. Let's add some HTML and echo out some of those variables.

Type the following blue and green text into your document in CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack. </li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
  ?>

  <?php
    /* Here are some imaginary numbers for a possible salary package for the skills we'
re learning: */
    $base_salary = 158470;
    $bonus = 25815.25;
    $benefits = 0.2;
    //percentage of total
    $time_off = 6476;
    //in dollars
    $benefits = "this is a string now"; // You just changed the value of $benefits from
 a float number to a string!
  ?>

  <br/>
  <ul>
    <li> My Base Salary might be: <?php echo $base_salary; ?> </li>
    <li> My Bonus might be: <?php echo $bonus; ?> </li>
    <li> My Benefits might be: <?php echo $benefits; ?> </li>
    <li> My Time Off might be worth: <?php echo $time_off; ?> </li>
  </ul>
</ol>
```

Click **Preview**. Those variables should show up now.

1. PHP is part of the LAMP stack.
    - My Base Salary might be: 158470
    - My Bonus might be: 25815.25
    - My Benefits might be: To be determined
    - My Time Off might be worth: 6476

There they are. Actually, instead of displaying themselves, our variables displayed the **values** they were holding.

By the way, they're not just sneaky; they're picky too. Turns out, **variable names** may consist of only **letters, numbers, and the underscore(_) character.** Not just that, the first character of the variable name CANNOT be a number.

Here's a list of valid and invalid variable names:

VALID variable names:

- **$_var**
- **$heres_a_name**
- **$t12345**
- **$x**

INVALID variable names:

- **$1_var**
- **$here's-a-name**

- $t+12345
- $x?

Modifying Variables and Values with Operators

Variables are not useful unless they've been modified. **Operators** can be used to modify variables and their values. Operators are fairly simple to use, in fact, you've already learned one: the **assignment** operator, represented by the equal sign (**=**). The assignment operator is a quick, easy, and intuitive way to instruct a variable to hold a certain value.

But what if we want to *change* a variable's value?

Type the following into CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack. </li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
  ?>
  <?php
    /* Here are some imaginary numbers for a possible salary package for the skills we'
re learning: */
    $base_salary = 158470;
    $bonus = 25815.25;
    $benefits = 0.2;
    //percentage of total
    $time_off = 6476;
    //in dollars
    $benefits = "this is a string now"; // You just changed the value of $benefits from
 a float number to a string!
  ?>
  <br/>
  <ul>
    <li> My Base Salary might be: <?php echo $base_salary; ?> </li>
    <li> My Bonus might be: <?php echo $bonus; ?> </li>
    <li> My Benefits might be: <?php echo $benefits; ?> </li>
    <li> My Time Off might be worth: <?php echo $time_off; ?> </li>
  <li> My Base Salary plus Bonus would total:  <?php echo $base_salary + $bonus; ?> </l
i>
  </ul>
</ol>
```

Preview this code. We added the values of the variable's *$base_salary* and *$bonus*. Sweet.

1. PHP is part of the LAMP stack.
    - My Base Salary might be: 158470
    - My Bonus might be: 25815.25
    - My Benefits might be: To be determined
    - My Time Off might be worth: 6476
    - My Base Salary plus Bonus would total: 184285.25

The plus sign (**+**) is also an **operator**. More specifically, a **binary operator**, since it takes *two* variables or values (in this case, called **arguments**), performs the addition operation on them, and returns the result - just like those shy variables do. In this case, we displayed the result through the "echo" statement.

Below is a list of some binary operators, and some examples of them in action.

Assuming $i = 12 and $j = 5 then...

Operator	Name	Usage	Result

=	assign	$i = $j	5
+	add	$i + $j	17
-	subtract	$i - $j	7
*	multiply	$i * $j	60
/	divide	$i / $j	2.4
%	mod (remainder of division)	$i % $j	2

Play around with these in your program and see what you get. Seriously, practice! Try applying different operators to the example you've been working on in this lesson, and echo out the results.

The operations (except for addition) need to be executed in the order they appear, from left to right, to work properly.

By the way, the operators above only operate on integers and floating point number values. There are different operators that work on strings. Most specifically, the **concatenation** operator. Here is an example using concatenation. Notice the period in front of $lamp_l:

Type the following into CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack. </li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
    echo "<br />The stack begins with " . $lamp_l;
  ?>
  <?php
    /* Here are some imaginary numbers for a possible salary package for the skills we'
re learning: */
    $base_salary = 158470;
    $bonus = 25815.25;
    $benefits = 0.2;
    //percentage of total
    $time_off = 6476;
    //in dollars
    $benefits = "this is a string now"; // You just changed the value of $benefits from
 a float number to a string!
  ?>
  <br/>
  <ul>
    <li> My Base Salary might be: <?php echo $base_salary; ?> </li>
    <li> My Bonus might be: <?php echo $bonus; ?> </li>
    <li> My Benefits might be: <?php echo $benefits; ?> </li>
    <li> My Time Off might be worth: <?php echo $time_off; ?> </li>
    <li> My Base Salary plus Bonus would total: <?php echo $base_salary + $bonus; ?> </
li>
  </ul>
</ol>
```

Preview the code and see what happens. After you Preview you should see the following sentence on your page:

The stack begins with Linux.

Let's break it down. How did this:

echo "The stack begins with " . $lamp_l;

become this?:

The stack begins with Linux.

Well, the first part in quotation marks is a string and the $lamp_l is a variable holding the string "Linux". To "add" them together, we use a period **.** which is the concatenation operator. Did you understand that? If not look again...

echo "The stack begins with ".$lamp_l;

When used properly in PHP, suddenly that lowly punctuation mark, the period (.), becomes a powerful concatenation tool. Yes, the concatenation operator (.) is yet another **binary operator** in PHP, and an extremely useful one at that.

But the most useful characteristic of these operators is that they can be **nested**. To nest operators essentially means we can use them together.

Type the following blue code (notice the periods in red) into CodeRunner:

```
<ol style="font-size:16px;">
  <li> PHP is part of the LAMP stack. </li>
  <?php
    $lamp_l = "Linux";
    $lamp_a = "Apache";
    $lamp_m = "MySQL";
    $lamp_p = "PHP";
    echo "<br />The stack begins with " . $lamp_l . " and goes on to include " . $lamp_
a . ", and " . $lamp_p . "!<br />";
  ?>
  <?php
    /* Here are some imaginary numbers for a possible salary package for the skills we'
re learning: */
    $base_salary = 158470;
    $bonus = 25815.25;
    $benefits = 0.2;
    //percentage of total
    $time_off = 6476;
    //in dollars
    $benefits = "This is a string now."; // You just changed the value of $benefits fro
m a float number to a string!
    $total = $base_salary + $bonus + $time_off;
    $total_compensation = $total + ($total * 0.2); // Adding in benefits
  ?>
  <br/>
  <ul>
    <li> My Base Salary might be: <?php echo $base_salary; ?> </li>
    <li> My Bonus might be: <?php echo $bonus; ?> </li>
    <li> My Benefits might be: <?php echo $benefits; ?> </li>
    <li> My Time Off might be worth: <?php echo $time_off; ?> </li>
    <li> My Base Salary plus Bonus would total: <?php echo $base_salary + $bonus; ?> </
li>
    <li> My total compensation would be <?php echo $total . " without benefits, and " .
 $total_compensation . " with benefits."; ?> </li>
  </ul>
</ol>
```

In the code above there is operator nesting happening all over the place. Preview your file. You should get something like this:

1. PHP is part of the LAMP stack.
   The stack begins with Linux, and goes on to include Apache, MySQL, and PHP!

   - My Base Salary might be: 158470
   - My Bonus might be: 25815.25
   - My Benefits might be: To be determined
   - My Time Off might be worth: 6476
   - My Base Salary plus Bonus would total: 184285.25
   - Your total compensation would be 190761.25 without benefits, and 228913.5 wi
     benefits.

It appears that operator nesting worked just fine. That's not to say that our **binary** operators started taking on more than two arguments. Instead, we've executed a succession of binary operations, with the one operation taking the results of the last operation into consideration.

Here are a couple more things to consider:

- Why do you suppose the "concat" operator (.) had no problem mixing strings, floats, and integers?
- Were the parentheses (()) necessary in the $total_compensation line?
- What role do you think the parentheses play?

Finally, here are some useful PHP "shortcut" operators. These operators reduce the need for nesting to execute some common tasks and they are really handy. Can you figure out which operators are **unary operators**? (Hint: unary operators need only *one* argument.)

Play around with these and see what you get:

Operator	Equivalent
$i += $j	$i = $i + $j
$i -= $j	$i = $i - $j
$i *= $j	$i = $i * $j
$i /= $j	$i = $i / $j
$i++	$i = $i + 1
$i--	$i = $i - 1
$i .= $j	$i = $i . $j

Superglobals

PHP has a set of predefined variables to make our lives easier. Superglobals can be accessed by classes, functions, or files at any time without having to do anything special! Very nice. So, what are these *Superglobals*?

Before we delve too deeply, let's get a small taste of what's to come. After all, we can't simply give out all the secrets in the beginning. There wouldn't be anything to look forward to!

$GLOBALS

1. References all variables that are in the global scope.
2. Associative array.
3. Variable names are keys of $GLOBALS array.

CODE TO TYPE: $GLOBALS example

```
<?php
 function testScope() {
  echo "The variable in the main code doesn't extend to within the function: $sc
ope<br>";

  //assign a value to the variable named $scope that IS within function scope
  $scope = "WITHIN FUNCTION";
  echo "The local scope within the function: $scope<br>";

  //the superglobal DOES extend within the function
  echo "The global scope: {$GLOBALS['foo']}<br>";
 }

 //define $scope in the main code
 $scope = "MAIN CODE";
 echo "The local scope in the main code body: $scope<br>";

 //define a global value (
 $GLOBALS['foo'] = "SUPERGLOBAL";
 echo "The value in the superglobal is {$GLOBALS['foo']}<br>";

 //now run function, which has separate scope and $scope variable
 testScope();

 //show that main code's $scope is unaffected
 echo "The local scope in the main code body: $scope<br>";
?>
```

The local scope in the main code body: MAIN CODE
The value in the superglobal is SUPERGLOBAL
The variable in the main code doesn't extend to within the function:
The local scope within the function: WITHIN FUNCTION
The global scope: SUPERGLOBAL
The local scope in the main code body: MAIN CODE

$_SERVER:

1. Array containing information to Headers, Paths, and Script locations.
2. Entries generated by web server.

CODE TO TYPE: $_SERVER example

```
<?php
echo $_SERVER['SERVER_NAME'];
?>
```

mchou.oreillystudent.com

$_GET

1. Associative array.
2. Populated by URL parameters.

CODE TO TYPE: $_GET example

```
<form action="" method="get">
Enter your name: <input type="text" name="myname" placeholder="Tim O'Reilly"/>
<input type="submit" />
</form>
<?php
 echo "Your name is: " . htmlspecialchars($_GET["myname"]);
?>
```

$_POST

1. Associative array.
2. Array is passed via HTTP POST method.

CODE TO TYPE: $_POST example

```
<form action="" method="post">
Enter the next phrase: <input type="text" name="next_phrase" size="50" placehold
er="he played knick-knack on my door."/>
<input type="submit" />
</form>
<?php
 echo "This old man, he played four, " . htmlspecialchars($_POST["next_phrase"])
;
?>
```

Note This is not an exhaustive list of PHP's Superglobals; however, click here for a full list with examples, definitions, and a peek of what's to come!

Phew! We've covered a lot of ground. Don't forget to **Save** your work, and hand in the assignments from your syllabus. See you at the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Decisions

In the last lesson we learned that storing values in variables and manipulating them with operators are among the most important tools we have for programming in PHP. Now let's talk about automating repetitive tasks and the decisions you'll make based on the values present in your programs.

Comparison Operators and Conditions

We make comparisons everyday. When we shop, we look at prices of similar items to determine which deals are best. When we take a trip, we compare alternative routes to decide which will be most expedient. Well, it turns out that we can do comparisons in PHP and other computer languages as well. Let's look at this process using a simple example. Let's try a comparison of **Captain Crunch** breakfast cereal to **Frosted Flakes** breakfast cereal.

Suppose Captain Crunch is 4 dollars a box, while Frosted Flakes is 5 dollars. We can use PHP to figure out which price is greater. (We realize you can determine this fairly easily without PHP, but let's go ahead and work the example anyway so we can see how PHP works.)

Add the following BLUE and GREEN code to your file in CodeRunner:

```
<?php
  $captain_crunch = 4;
  $frosted_flakes = 5;
?>
Does Captain Crunch cost less than Frosted Flakes? <?php echo ($captain_crunch < $frost
ed_flakes); ?>
<br />
<br />
Or, is Captain Crunch priced greater than Frosted Flakes? <?php echo ($captain_crunch >
 $frosted_flakes); ?>
```

Preview that. What was returned from the echo command here?

In the the last lesson we learned to modify variables using some of the standard operators: add, subtract, concatenate, and others. The **comparison operators** we're using now compare two variables, producing TRUE or FALSE results.

For instance, the program above compares two integers to determine if one is larger than the other. ("<" is a symbol that means "less than," while ">" means "greater than").

This statement is TRUE:

```
4 < 5
```

We all know that 4 is less than 5.

This statement is FALSE:

```
4 > 5
```

So how did your program answer the question it was asked about Captain Crunch and Frosted Flakes?

Here's what you saw:

Does Captain Crunch cost less than Frosted Flakes? **1**

Or, is Captain Crunch priced greater than Frosted Flakes?

Obviously, in our example, Captain Crunch is less (expensive) than Frosted Flakes, but what's with the number one (1) at the end there? Was that a typo?

No, that wasn't a typo. Turns out, this is how PHP interprets the **Boolean** result "TRUE." (Boolean, by the way, is just a fancy programming word referring to the results of "true or false" inquiries.) Similarly, instead of returning "FALSE" or "no" when asked if Captain Crunch is greater than Frosted Flakes, your program returned the **NULL** character. "Null" is computer-speak for "nothing." When things are false, nothing gets returned, so nothing is printed.

Here's a table of values that PHP can interpret as TRUE or FALSE:

FALSE	TRUE	Notes
0 (zero)	any non-zero number	non-zero examples: 1, -1, 0.5
false	true	no quotes ("), otherwise it's just a string
NULL, null, '', or ""	any non-null string	The space (" ") character is NOT the same as the null ("") character

Here's a list of comparison operators you can experiment with in your program:

Operator	Name	Usage	Result
==	Equal	$a == $b	TRUE if $a and $b are equal.
===	Identical	$a === $b	TRUE if $a and $b are equal AND if they are of the same type (ie $a and $b are both integers).
!= <>	Not equal	$a != $b $a <> $b	TRUE if $a and $b are not equal.
!==	Not identical	$a !== $b	TRUE if $a is not equal to $b OR if $a and $b are not of the same type.
<	Less than	$a < $b	TRUE if $a is less than $b.
>	Greater than	$a > $b	TRUE if $a is greater than $b.
<=	Less than or equal to	$a <= $b	TRUE if $a is less than OR equal to $b.
>=	Greater than or equal to	$a >= $b	TRUE if $a is greater than OR equal to $b.

Note

In PHP we use two equal signs (**==**) to test for equality. (When you use "==" you're essentially asking: **Are these values equal?**). Two equal signs, like `$a == $b`, compare $a to $b (in English it would read "is $a equal to $b?"), whereas one equal sign `$a = $b` assigns $b to $a (in English it would read "set $a **is** equal to $b").

IF and ELSE Control Structure

You may not know it, but you actually already understand **if** and **else** control structures. You use them everyday when you decide things like **"I'll buy Captain Crunch if it's less expensive than Frosted Flakes, or else I'll buy Frosted Flakes**. You've set conditions and also decided on an alternative course of action should those conditions fail to be met. In a program, this is called a control structure.

In PHP, you would write the sentence above like this:

Type the following into a new file in CodeRunner:

```
<?php $Captain_Crunch = 4; $Frosted_Flakes = 5; if ($Captain_Crunch
        < $Frosted_Flakes) { echo "I'll buy Captain Crunch"; }
    else { echo "I'll buy Frosted Flakes"; } ?>
```

Preview the code above. Which cereal does PHP instruct you to buy? Try changing the numbers assigned to $Captain_Crunch and $Frosted_Flakes to see what happens.

if statements have a specific form.

OBSERVE:

```
if (expression){
                    statement(s) executed if expression is TRUE } else {
                    (optional) statement(s) executed if expression IS FALSE
                }
```

Again, this **if** statement is also referred to as a **control statement**. PHP first evaluates the **expression** to see if it is true or false. If the **expression** is true, then the statements in **blue** are executed. If not, then the statements in **BLUE** are not executed, but the **green** ones are.

Logical Operators

Questions can be more complicated than statements. For instance, if Captain Crunch is more (expensive) than Frosted Flakes, but Fruit Loops are less (expensive) than Frosted Flakes, we might want to choose Fruit Loops. The following code can handle these kinds of complications:

Add the colored code below into your document in CodeRunner:

```
<?php $Captain_Crunch = 5; $Frosted_Flakes = 4;
    $Fruit_Loops = 3;
 if ($Captain_Crunch < $Frosted_Flakes) {
     echo "I'll buy Captain Crunch";
    } else if ($Captain_Crunch > $Frosted_Flakes && $Frosted_Flakes > $Fruit_Loops) {
          echo "I'll buy Fruit Loops.";
    } else { echo "I'll buy Frosted Flakes.";
 }
?>
```

Preview this code. Which cereal does PHP recommend that you buy? Try changing the numbers representing the prices and observe the results.

In this example, we've added a couple of things for your consideration. First we added a **logical operator**. We used **&&** which simply means **AND**. The other addition was the **else if**. We can have as many of those within an **if** statement as we need.

So now the line reads:

OBSERVE:

```
else if ($Captain_Crunch > $Frosted_Flakes && $Frosted_Flakes > $Fruit_Loops) {
 echo "I'll buy Fruit Loops.";
```

In English, the line reads:

OBSERVE:

```
"Or else if Captain Crunch is greater than Frosted Flakes,
AND
Frosted Flakes is greater than Froot Loops, then I'll buy Fruit Loops.
```

Notice that when Captain Crunch is 6 dollars **and** Frosted Flakes is 5 dollars **and** Fruit Loops is 4 dollars, **then** **$Captain_Crunch > $Frosted_Flakes** is **TRUE**, and that **$Frosted_Flakes > $Fruit_Loops** is **TRUE**. So the whole thing is TRUE and so you'll buy Fruit Loops!

Here are some rules to remember about logical operators:

- (**TRUE AND TRUE**) is **TRUE**
- (**TRUE AND FALSE**) is **FALSE**
- (**TRUE OR FALSE**) is **TRUE**
- (**FALSE OR FALSE**) is **FALSE**

Like comparison operators, the logical operator performs a comparison on two arguments, and returns a TRUE or FALSE (1 or null) answer. However, the logical operator compares things that are already TRUE or FALSE.

Below is a list of logical operators.

Operator	Name	Usage	Result
AND	AND	$a AND $b	TRUE if $a and $b are TRUE

&&	AND	$a && $b	TRUE if $a and $b are TRUE.
OR ||	OR	$a OR $b $a || $b	TRUE if $a or $b is TRUE.
XOR	Exclusive OR	$a XOR $b	TRUE if $a OR $b is TRUE, but not both.

Look again at this condition:

```
($Captain_Crunch >= 3 && $Frosted_Flakes < 10)
```

Remember that nested operators perform in a certain order, depending on certain rules? There are rules here too. Specifically, the comparison operators are evaluated *before* the logical evaluator. This way, the logical evaluator only needs to look at the TRUE or FALSE results, and act accordingly.

Like this:

```
(($Captain_Crunch >= 3) &&
($Frosted_Flakes < 10)) ((TRUE) &&
(TRUE)) (TRUE)
```

Operator nesting is really useful. And fortunately, you can do it with logical operators too.

Type the following into CodeRunner:

```
<?php
    $Captain_Crunch = 5;
    $Frosted_Flakes = 4;
    $Fruit_Loops = 5;
    $Oatmeal = 2;
 if ($Captain_Crunch < $Frosted_Flakes) {
     echo "I'll buy Captain Crunch";
    } else if ($Captain_Crunch > $Frosted_Flakes && $Frosted_Flakes > $Fruit_Loops) {
        echo "I'll buy Fruit Loops.";
    } else if ($Captain_Crunch > 4 && $Fruit_Loops > 4 && $Oatmeal < 4 ) {
        echo "I'll get some Oatmeal.";
    }
 ?>
```

Play with this one for a while—try entering different values for the different cereals. Be sure to Preview often to see what happens.

NOTICE: Before you move on, Save the PHP file you've been working on as **compare.php**.

A Brief Preview of Forms

Before we continue on with control structures, let's make these examples a little more interesting by getting your PHP program to take input from a user on the web. We're going to make a form that will help us understand these control structures better. This is a brief introduction. We'll cover forms in much more detail later in the course.

So far in this lesson, we've been changing the values in the variables **$Captain_Crunch**, **$Frosted_Flakes**, **$Fruit_Loops**, and **$Oatmeal** by hand. Typically though, control structures evaluate changes made to variables and then react to those changes. If the user changes an input, we can account for all the possibilities through control structures.

To make our program more interactive, we're going to make a web page with a few input forms we'll use to submit values to our PHP program. Then we're going to take those inputs and assign them to the variables listed above.

Let's make an HTML form.

Make sure you're in HTML and type the following into CodeRunner:

```
<body>
<h3>Choosing your Cereal</h3>
<form method="GET" action="compare.php"> Enter the price of Captain Crunch:
    <input type="text" size="25" name="crunch_price" value="" />
<br /> Enter the price of Fruit Loops: <input type="text" size="25"
        name="loops_price" value="" />
<br /> Cash in my wallet: <select name="cash_money">
<option value="">How much cash?</option>
<option value="1">$1.00</option>
<option value="2">$2.00</option>
<option value="3">$3.00</option>
<option value="4">$4.00</option>
<option value="5">$5.00</option>
<option value="10">$10.00</option>
</select>
<input type="submit" value="SUBMIT" />
</form>
</body>
```

Now Preview this in HTML. If you select an item and click submit, it won't do anything. You should see a page that looks like this:

Choosing your cereal

Enter the price of Captain Crunch:

Enter the price of Fruit Loops:

Cash in my wallet: How much cash do you have? SUBMIT

Save this page as **userinput.html**, or anything you like, so long as you can remember the name.

Now we've made a web page that will take input from a web user, and then send that input to the PHP program that we'll use to process it. Now we just need to make our PHP program retrieve the input. To do this, we have to use something called a **superglobal array**. Now that's a mouthful! We'll actually study superglobals in detail in a later lesson. For now let's just try it!

Switch back to **PHP** with the **compare.php** PHP program we've been using, and make the following changes.

Type the changes in BLUE into CodeRunner:

```
<?php
 $Captain_Crunch = $_GET["crunch_price"];
    $Frosted_Flakes = 4;
 $Fruit_Loops = $_GET["loops_price"];
 $Oatmeal = 2;
 $my_cash = $_GET["cash_money"];
    $total = $Captain_Crunch + $Frosted_Flakes;
 if ($total < $my_cash) {
     echo "I'll buy both Captain Crunch and Frosted Flakes!";
 } else if ($Captain_Crunch < $my_cash ) {
     echo "I'll buy Captain Crunch.";
 } else if ($Captain_Crunch > $my_cash && $Fruit_Loops < $my_cash ){
  echo "I'll buy some Fruit Loops.";
 } else {
  echo "Forget it, I'm going home.";
 }
?>
```

Now Save this PHP program as **compare.php**, then go back to your **userinput.html** file in HTML. Preview

it. Enter different prices for the two cereals, select the amount of cash you have in your wallet, then click submit. Now your program should change according to the input you submitted in the form. Cool, huh?

We're just getting started with **control structures**, so be sure to save your work and hand in your assignments. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Multiple Control Structures and Loops

Let's continue on from the last lesson. Make sure you've opened **userinput.html** in **HTML**, and **compare.php** in **PHP**. Ready? Let's go!

Multiple Control Structures

In the last lesson we introduced the **else if** statement, which allows us to work with multiple conditions.

Check out this else if statement:

```
<?
      if ($total < $my_cash) {
      echo "I'll buy both Captain Crunch and Frosted Flakes!";
      }
      else if ($Captain_Crunch < $my_cash ) {
      echo "I'll buy Captain Crunch.";
      }
      else if ($Captain_Crunch > $my_cash && $Fruit_Loops < $my_cash ){
         echo "I'll get some Fruit Loops.";
      }
      else {
         echo "Forget it, I'm going home.";
      }

?>
```

When the first **if** statement fails, PHP checks the **else if** statement before going on to **else**. As long as you begin with an **if** and end with an **else**, (*if* you have a default action), you can add any number of **else if** statements in the middle. This way we can check and react to a lot of conditions. Let's add some conditions to the example we worked on last lesson.

Type the code in blue into your PHP program in CodeRunner:

```
<?

   $Captain_Crunch = 5;
   $Frosted_Flakes = 4;
   $Fruit_Loops = 3;
   $Oatmeal = 2;
   $my_cash = $_GET["cash_money"];

   $total = $Captain_Crunch + $Frosted_Flakes;


   if ($my_cash == 10) {
      echo "I'll buy both Captain Crunch and Frosted Flakes!";
   }
   else if ($my_cash == 5  ){
      echo "I'll buy Captain Crunch.";
   }
   else if ($my_cash == 4){
      echo "I'll buy Frosted Flakes.";
   }
   else if ($my_cash == 3 ){
      echo "I'll buy Fruit Loops.";
   }
   else if ($my_cash == 2){
      echo "I'll buy Oatmeal.";
   }
   else {
      echo "Forget it, I'm going home.";
   }

?>
```

Save this code using the filename compare.php, and open your **userinput.html** file in HTML. Try entering some numbers for the amount of money you have and Preview it (the other variables are set in the program, so the input for those fields won't matter in this example).

Even though the above code works just fine, the procedure could be streamlined by using a **switch** statement. The `switch` control structure is similar to the `if` control structure, but it's especially useful when you have one variable with many possible values. The switch control structure is a more efficient means of accomplishing the same task. It's up to you decide which control structure you like better.

This is how we'd change our code into a switch statement:

```
   switch($my_cash) {
   case "10":
   echo "I'll buy both Captain Crunch and Frosted Flakes.";
   break;
    case "5":
    echo "I'll buy Captain Crunch.";
      break;
    case "4":
      echo "I'll buy Frosted Flakes.";
      break;
    case "3":
      echo "I'll buy Fruit Loops.";
      break;
    case "2":
      echo "I'll buy Oatmeal.";
      break;
    default:
      echo "Forget it, I'm going home.";
   }
```

Give it a try. Save the old file as new_file.php (don't forget to make a new HTML file as well), then replace the block of code that contains the **if** statements with the switch statements above. Use whichever method you prefer, it's your call.

And of course we want you to practice! Especially since we're going to assign this task as an objective later.

Before we go on, experiment and find answers to these questions:

- What happens when you **nest** comparison operators?
- Would (null == 0) be TRUE or FALSE?
- How about (null === 0)?
- In an **If** statement, do you have to have parentheses (**()**) around the condition?
- What about brackets (**{}**)?
  Hint: Try this with one action statement AND with two.
- Can you put the *whole* **if** control structure in one line?
- Do you always need to have an **else** statement?
- When nesting **logical** operators, do you need parentheses?
- What happens when you remove **break;** statements?

WHILE and FOR Loops

A **loop** is a repetitive task that goes on *while* something is true or *for* some number of steps. That's why they are called "while" and "for" loops.

A **while loop** has the following structure:

while (*something* is true) { do some stuff };

As soon as that something is false, the while loop stops.

Whereas a **for loop** has the following structure:

for (some number of steps) { do some stuff };

Let's look at an example. Type this into a new PHP file in CodeRunner:

```
<?

   echo "Hide and go seek, I'm counting to 25:<br>";

      $counts = 1;
      while ($counts <= 25) {
          echo $counts." Mississippi...<br>";
          $counts++;
      }

      echo "Ready or not, here I come!<br>";

?>
```

In case you didn't play hide-and-seek in your childhood, this is how you'd count out loud while giving the other kids a chance to hide. Such fun!

We have introduced a new form of control structure - the **WHILE loop**. And like with any control structure, the WHILE loop does something in response to a TRUE conditional statement. In this case, however, the loop continues to

repeat the action *until* the conditional statement is FALSE.

All loops have four essential parts:

1. The **initial value statement**, in this case **$counts = 1;**
2. The **conditional statement**, in this case **$counts <= 25**
3. The **action statement(s)**, in this case **echo $counts." Mississippi...
";**
4. The **increment statement**, in this case **$counts++;** (Remember this **unary operator**?)

In order for a loop to work, it has to have a starting point, an ending point, and something to do in between. What would happen if any of the four elements in our example were missing? Try messing with them, and you'll find out. (Go ahead, try. I'll wait.)

Note	Loops follow the same scheme as any control structure, in that you can **nest** all kinds of conditional statements and actions within them, including more loops. This can be lots of fun -- especially for duping your buddies into thinking the computer screen is possessed by gremlins!

Our *counting* loop example above is a really common loop--so common, in fact, that almost all programming languages have developed an alternate type of loop that can be used as a shortcut: the **FOR loop**.

The FOR loop structure looks like this:

```
for ($counts = 1; $counts <= 25; $counts++) {
    echo $counts." Mississippi...<br>";
}
```

Try replacing your WHILE loop with this FOR loop, then Preview the code. You should see the exact same result. In fact, the FOR loop has exactly the same four elements as the WHILE loop. The only difference is the order of the elements in the syntax. Well, that, and if you forget the **increment statement** in this one, PHP will yell at you. Sounds mean, but sometimes we need a little kick to keep from inadvertently causing an **infinite loop**. Yuck.

You may think that learning both WHILE loops and FOR loops in PHP is needless and redundant, and it's true that most of the time they are interchangeable. However, as your scripts gain more complexity, you'll find that some tasks are a perfect fit for using **FOR**, while using **WHILE** is best for others.

You've come really far! Now you can program the majority of what you'll need in PHP. Congratulations! You are one of the best PHP programmers in the world!

...Circa 1995, that is. To create cutting-edge web software for *this* century, we have a ways to go. Take heart—you've accomplished a lot already. Save your work, and don't forget the assignments. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Functions

A **function** acts like a small program within your larger program. You invoke a function, send it information, and get something back. A great way to learn to use functions, is to create one and use it. Let's go!

Creating Code Reusability with Functions

Let's create a new program so we can practice using functions. Our new program will take user input just like our previous programs, only this time we'll inquire about the user's state of mind and recommend a mantra for them to repeat.

Make sure you're in HTML syntax and type the following into CodeRunner:

```
<body>

   <h3>OST's Mantra generator</h3>

   <form method="GET" action="mantra.php">

   My current mood:
      <select name="my_mood">
         <option value="">Please choose...</option>
         <option value="happy">I'm happy.</option>
         <option value="sad">I'm sad.</option>
         <option value="angry">I'm angry.</option>
         <option value="indifferent">I'm indifferent.</option>
      </select>


      <input type="submit" value="SUBMIT" />

   </form>

   </body>
```

Save this HTML file as **moodinput.html**

In PHP Mode, type the following code highlighted in BLUE into CodeRunner:

```
<?
$my_mood = $_GET["my_mood"];

   if ($my_mood == "happy") {

      echo "Repeat the following: <br><br>";

      for ($chant = 1; $chant <= 10; $chant++) {
         echo " OM...  ";
      }

   }
   else if ($my_mood == "sad") {

      echo "Repeat the following: <br><br>";

      for ($chant = 1; $chant <= 10; $chant++) {
         echo " okay... ";
      }

   }
   else if ($my_mood == "angry") {

      echo "Repeat the following: <br><br>";

      for ($chant = 1; $chant <= 10; $chant++) {
         echo " Mississippi... ";
      }

   }
   else if ($my_mood == "indifferent") {

      echo "Repeat the following: <br><br>";

      for ($chant = 1; $chant <= 10; $chant++) {
         echo " Wake up... ";
      }

   }
   else {

      echo "Repeat the following: <br><br>";

      for ($chant = 1; $chant <= 10; $chant++) {
         echo " Try harder... ";
      }

   }

?>
```

Save this code as **mantra.php** and, once again, open up the HTML file moodinput.html.

Preview the HTML. Enter different values for your mood (after all, we're all pretty moody).

We have used pretty much the same PHP code in several places. When you have the same code or similar code in lots of different places, use a **function**. It will make your program more readable and save time. Say we decided to recommend chanting our mantra 20 times instead of 10. Unless we used a function, we'd have to change the code by hand in each of the "for" loops. Even for a simple code like ours, that would be truly annoying. You can imagine trying to do this with long and complicated code--it could get downright ugly. Fortunately, **functions** enable us to avoid such unpleasantness. We can create a function to execute a particular task each time we enter the name of the function into our program. This is referred to as "calling" a function. Change your program so it looks like the stuff below—be sure to remove the "for" loops within the "else if" statements.

Switch back to PHP and add the following code in BLUE into CodeRunner:

```
<?

function mantra($the_sound) {

   for ($chant = 1; $chant <= 10; $chant++) {
       echo $the_sound . "... ";
   }
}


$my_mood = $_GET["my_mood"];

   if ($my_mood == "happy") {

      mantra("OM");

   }
   else if ($my_mood == "sad") {

      mantra("okay");

   }
   else if ($my_mood == "angry") {

      mantra("mississippi");

   }
   else if ($my_mood == "indifferent") {

      mantra("Wake up");

   }
   else {

      mantra("Try harder");

   }
?>
```

Save this file as **mantra.php** and then switch back to your **moodinput.html** file. Try entering different moods. You should get the same results as before, but this time you used functions.

Each time the **mantra("something")** function is encountered by PHP, it calls the function definition **function mantra($the_sound)**. The variable (**$the_sound** in this case) is set to whatever is in between the parentheses when mantra("something") is encountered. For instance, **mantra("OM")** is calling the **function mantra($the_sound)** and setting **$the_sound = "OM"**. This is known in functions as "setting a parameter." This particular function takes one parameter: $the_sound. However, functions can take no parameters at all or many different parameters.

Much like a variable holds values, **functions** hold **processes** (snippets of code) that we want to reuse. So instead of having to add the same code over and over again, we can simply *call* the **function**. In this case, when **mantra()** is encountered, the code inside of the brackets **{** and **}** in the definition **function mantra(){ }** is executed. Functions will only be executed when they are *called*. Try removing the calls to *mantra()*. You'll see that the function doesn't do anything then.

Note If the function you've created doesn't have any parameters, you still need to have the parentheses in place, they just won't contain anything. Notice how no dollar signs (**$**) are used in the function name **mantra**, but instead we follow the name with parentheses(**()**). They need to be there, that's just the way it is.

Congratulations—you've just worked through your first example of **code reusability**!

Function and Variable Scopes

Scope refers to a variable's area of influence. If a variable is defined inside of a function, then its area of influence is only within that function. That means we can use that variable name again in another function--setting values to it in one function won't affect the setting in another function. Let's try using a function to encapsulate those "if" statements in our example from the last section. In the process, we can see how scope may affect the outcome of our program.

Revise your PHP program so it looks like this in CodeRunner:

```
<?

function mantra($the_sound) {

   for ($chant = 1; $chant <= 10; $chant++) {
       echo $the_sound . "... ";
   }

}

function Mood_Chant(){


   if ($my_mood == "happy") {

      mantra("OM");

   }
   else if ($my_mood == "sad") {

      mantra("okay");

   }
   else if ($my_mood == "angry") {

      mantra("mississippi");

   }
   else if ($my_mood == "indifferent") {

      mantra("Wake up");

   }
   else {

      mantra("Try harder");

   }

}

$my_mood = $_GET["my_mood"];

Mood_Chant();

?>
```

Save this as **mantra.php**, then open your **moodinput.html** file. Try altering ANY of the moods on the list. No matter what you choose, this code will always return **Try Harder** as the output.

So why is the program returning **Try Harder** as a result, no matter what we select? Let's perform some diagnostic tests to find out. We'll enter some echo statements to print out variable values in different parts of our program. Then we can use the information we get to determine the path our program is taking and the steps we need to take to correct our problem. Let's try it.

Add some echo statements into CodeRunner:

```
<?
function mantra($the_sound) {

   for ($chant = 1; $chant <= 10; $chant++) {
       echo $the_sound . "... ";
   }

}

function Mood_Chant(){

echo "INSIDE the Mood_Chant function, your mood is ".$my_mood.".<br>";

   if ($my_mood == "happy") {

      mantra("OM");

   }
   else if ($my_mood == "sad") {

      mantra("okay");

   }
   else if ($my_mood == "angry") {

      mantra("mississippi");

   }
   else if ($my_mood == "indifferent") {

         mantra("Wake up");

   }
   else {

         mantra("Try harder");

   }

}

$my_mood = $_GET["my_mood"];

echo "OUTSIDE the Mood_Chant function, your mood is ".$my_mood.".<br>";

Mood_Chant();

 ?>
```

Once again, Save this as **mantra.php**, then go back to your **moodinput.html** page. Select **angry** from the drop down list and submit it.

You should get something like this:

OUTSIDE the Mood_Chant function, your mood is angry.
INSIDE the Mood_Chant function, your mood is .
Try Harder...Try Harder...Try Harder...Try Harder...Try Harder...Try Harder...Try Harder...Try Harc
Harder...Try Harder...

Look closely. What printed out? What didn't? The first result printed out **OUTSIDE the Mood_Chant function, your mood is angry.** We asked PHP to print it with the statement **echo "OUTSIDE the Mood_Chant function, your mood is ".$my_mood.".";**. So as expected, the variable *$my_mood* was defined as *angry*. However, the second result printed **INSIDE the Mood_Chant function, your mood is .** Even though we asked PHP to print **echo "INSIDE the Mood_Chant function, your mood is ".$my_mood.".";** , it didn't print a value for $my_mood.

In the above example, you would think the value of **$my_mood** (**"angry"**) would print both inside and outside of the function **Mood_Chant()**. But, once the function was called, the value **$my_mood** wasn't seen INSIDE of the Mood_Chant() function at all. This is because the variable **$my_mood** is completely different depending on whether it is located outside or inside of the function. Although variables may share the same name, their location determines their effect on the program. When a variable within a function is *encapsulated*, as if the function was its own program, this is referred to in programming as the function's **scope**.

In the next section, we'll learn to set parameters so that scope doesn't prevent us from using functions to the fullest.

Note

PHP isn't as strict with scope as some other languages are. Since PHP isn't strongly typed, you're not required to **declare** variables before you use them. Therefore, within a PHP function, a variable declared within a loop will retain its value outside of that loop. To see this concept at work, try using **echo** to output **$chant** after the **for loop** is finished in **mantra()**.)

Using Functions with Parameters and Return Values

As interesting as scope can be, it doesn't help lighten your work load. What's the use of reusing your code in a function, if you have to re-define **$my_mood** within the function? Worse, what if you want to have different values for **$my_mood** anytime you use the function **Mood_Chant()**? We could save ourselves a lot of work if we could feed our function different values and get an output each time. We already did this in the first section above using **parameters**.

Sneaking In with Parameters

Type the following into CodeRunner:

```
<?

function mantra($the_sound) {

   for ($chant = 1; $chant <= 10; $chant++) {
       echo $the_sound . "... ";
   }

}

function Mood_Chant($my_mood){

echo "INSIDE the Mood_Chant function, your mood is ".$my_mood.".<br>";

   if ($my_mood == "happy") {

      mantra("OM");

   }
   else if ($my_mood == "sad") {

      mantra("okay");

   }
   else if ($my_mood == "angry") {

      mantra("mississippi");

   }
   else if ($my_mood == "indifferent") {

      mantra("Wake up");

   }
   else {

      mantra("Try harder");

   }

}

$my_mood = $_GET["my_mood"];

echo "OUTSIDE the Mood_Chant function, your mood is ".$my_mood.".<br>";
Mood_Chant($my_mood);

 ?>
```

Save this as **mantra.php**, open **moodinput.html**, and select *angry* from the drop-down list. This time, you should have gotten the results you expected.

Look at your function again:

```
function Mood_Chant($my_mood) {

//code that processes the value of $my_mood

}

.
.
.


//passing the value of $my_mood UP to the Mood_Chant function above
Mood_Chant($my_mood);
```

Passing a **parameter** essentially drills through the wall of your function's scope, making it a more useful machine.

Whatever **parameter** we call to **Mood_Chant(parameter);** becomes the value for **$my_mood**. And you don't even have to use the name **$my_mood**, since it's a completely different variable within the function and outside the function. Try using this on your own.

Look at your function again:

```
function Mood_Chant($my_mood) {

//code that processes the value of $my_mood
}

.
.
.


//passing the value of $my_mood up to the Mood_Chant function above
Mood_Chant("happy");
```

The value of **$my_mood** inside of the function Mood_Chant($my_mood) is **"happy"**. It's like setting *$my_mood = "happy"* INSIDE of the function.

Now that we've *snuck in* with parameters, let's *sneak out* with return values.

Sneaking out with Return Values

In the examples above, we saw that we can sneak into a function using parameters. We can also sneak out using **return** values. The best way to understand "return" is to use it. Let's get to it.

Type the following into CodeRunner:

```
<?

function mantra($the_sound) {

   for ($chant = 1; $chant <= 10; $chant++) {
       echo $the_sound . "... ";
   }

}

function Mood_Chant($my_mood){

   if ($my_mood == "happy") {

      mantra("OM");
      $after_chant = "<br>I feel serene now.";

   }
   else if ($my_mood == "sad") {

      mantra("okay");
      $after_chant = "<br>I feel better now.";

   }
   else if ($my_mood == "angry") {

      mantra("mississippi");
      $after_chant = "<br>I've calmed down now.";

   }
   else if ($my_mood == "indifferent") {

      mantra("Wake up");
      $after_chant = "<br>I'm awake now.";

   }
   else {

      mantra("Try harder");
      $after_chant = "<br>I'll try harder now.";

   }

   return $after_chant;
}

$my_mood = $_GET["my_mood"];

$after_chant_mood = Mood_Chant($my_mood);

echo $after_chant_mood;

?>
```

Save this as **mantra.php**, open up **moodinput.html**, and select anything you like from the drop down list. Now you should get the chant and at the end you should have an "after-chant mood" expressed. All we did here was add some statements into the variable $after_chant and then use **return $afterchant** at the end of the function. When we use return, we are setting a value in place of the function.

But instead of just letting a parameter sneak in, you've allowed a **return value** to sneak *out* of the function scope. Suddenly, your function is an efficient factory, taking in raw ingredients (parameters) and spitting out a refined product -- that is, it *returned a value*. Allowing a return value to sneak out of the function scope is used often in programming to return true or false values in functions that perform tests.

Multiple Parameters and Default Values

We practiced using parameters earlier in the lesson and now we can pass parameters to a function. Let's change our function so that the end user can set how many times we chant our mantra.

Type the following code into your moodinput.html file in CodeRunner:

```
<body>

      <h3>OST's Mantra generator</h3>

      <form method="GET" action="mantra.php">

      My current mood:
      <select name="my_mood">
         <option value="">Please choose...</option>
         <option value="happy">I'm happy.</option>
         <option value="sad">I'm sad.</option>
         <option value="angry">I'm angry.</option>
         <option value="indifferent">I'm indifferent.</option>
      </select>

      Pick a number:
      <select name="my_number">
         <option value="2">Please choose...</option>
         <option value="10">10</option>
         <option value="20">20</option>
         <option value="30">30</option>
         <option value="40">40</option>
      </select>


      <input type="submit" value="SUBMIT" />

      </form>
</body>
```

Save this as **moodinput.html** again. We've added the option of selecting a number, so let's change our program to accept this information and process it.

Type the following in your PHP file in CodeRunner:

```
<?

function mantra($the_sound,$the_number = 10) {

   for ($chant = 1; $chant <= $the_number; $chant++) {
       echo $the_sound . "... ";
   }

}

function Mood_Chant($my_mood, $chant_number = 10){

   if ($my_mood == "happy") {

      mantra("OM",$chant_number);
      $after_chant = "<br>I feel serene now.";

   }
   else if ($my_mood == "sad") {

      mantra("okay",$chant_number);
      $after_chant = "<br>I feel better now.";

   }
   else if ($my_mood == "angry") {

      mantra("mississippi",$chant_number);
      $after_chant = "<br>I've calmed down now.";

   }
   else if ($my_mood == "indifferent") {

      mantra("Wake up",$chant_number);
      $after_chant = "<br>I'm awake now.";

   }
   else {

      mantra("Try harder",$chant_number);
      $after_chant = "<br>I'll try harder now.";

   }

   return $after_chant;
}

$my_mood = $_GET["my_mood"];
$chant_number = $_GET["my_number"];

$after_chant_mood = Mood_Chant($my_mood, $chant_number);

echo $after_chant_mood;

?>
```

Save this as **mantra.php**, open **moodinput.html**, and Preview.

In this program we let the user select a number. Then inside of the mood_chant function we call Mantra(first parameter, second paramter) where the second parameter is the number the end user chose on the form in the first place. Notice we changed the function Mantra() to accept two parameters.

By adding a **default value** to the parameter **$the_number**, you made that parameter completely *optional* when you call **Mantra**. To see this in action, try changing the program so that one of the calls to Mantra() has only one parameter being set.

Note You can have as many parameters and default values as you want in a function. But you have to make sure that the default-valued parameters are at the *end* of the parameter list. Any idea why? Experiment to find out!

Be sure to save your work and hand in your assignments. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Arrays

Have you ever used one of those weekly pill containers? You know, the ones that keep your vitamins or medicine organized for each day? Surely you've at least *seen* one:

This is an excellent representation of this entire lesson—that box is just an **array** of containers with objects in them. Let's get started with a fresh file and take a breather from the monster we've created.

Creating an Array

Open a new PHP file and type this code into CodeRunner:

```
<?php

$names = array("scott","kendell","Trish","Tony","Mike","Debra","Curt");

echo "<pre>";
print_r($names);
echo "</pre>";


?>
```

Save and preview the file:

```
Array
(
    [0] => scott
    [1] => kendell
    [2] => Trish
    [3] => Tony
    [4] => Mike
    [5] => Debra
    [6] => Curt
)
```

You've just *defined* an **array** named **names**, by passing the seven names as parameters to the built-in PHP **array()** construct. If you think in terms of the pill box above—a huge, people-sized pill box—it would look like this:

Now, the great thing about arrays is that you can access and mess with any one of the **elements**—names, pills, whatever values are in the boxes—by using the array *keys*. Let's give Mike a call:

Type the following into CodeRunner:

```
<?php

$names = array("scott","kendell","Trish","Tony","Mike","Debra","Curt");

echo "Who is it?  ...".$names[4]."<br/>";

echo "<pre>";
print_r($names);
echo "</pre>";

?>
```

Preview this. Did you see Mike's name? All you did here is retrieve the value of the array **element** at the 4th position, or **index**. In this case, you used the index **4** as the **key**. For kicks, let's replace Mike:

Type the following into CodeRunner:

```
<?php

$names = array("scott","kendell","Trish","Tony","Mike","Debra","Curt");

echo $names[4];

$names[4] = "Josh";
echo "Who is it?  ...".$names[4]."<br/>";

echo "<pre>";
print_r($names);
echo "</pre>";

?>
```

See, this is why we love arrays. No scope to contend with, just a simple organization of values that we can mess with at will. So now the **$names** array looks like this:

Note Notice the new, super-handy, built-in function called **print_r**, which prints out an array in a really nice, readable format. With a little experimentation, you can figure out why we used the <pre> and </pre> tags, too. You can find out more about this function at php.net.

Associative Arrays

If we wanted to represent the pill box in PHP, it would make sense to use the labels that already exist to mark each box for our purposes as well. Here's one way to do it:

Type the following into CodeRunner:

```
<?php

$weekly_pills = array("S" => "vitamin C",
                      "M" => "Echinacea",
                      "T" => "antibiotic",
                      "W" => "calcium",
                      "Th" => "zinc",
                      "F" => "multivitamin",
                      "Sa" => "alka seltzer");

echo "<pre>";
print_r($weekly_pills);
echo "</pre>";

?>
```

Save it as **pills.php** and preview:

```
Array
(
    [S] => vitamin C
    [M] => Echinacea
    [T] => antibiotic
    [W] => calcium
    [Th] => zinc
    [F] => multivitamin
    [Sa] => alka seltzer
)
```

You've just defined an *associative array*. By using the **=>** operator, you've *associated* each array element value to its own index, or *key*, so that you can access it more intuitively. In other words, an associative array is a way of naming each slot of the array. In this case, the slots are named **S**,**M**, **T**, **W**, **Th**, **F**, and **Sa**, respectively. So now we can store and access values in an array based on these names instead of using indices. Experiment with this:

Type the following into CodeRunner:

```
<?php

$weekly_pills = array("S" => "vitamin C",
                      "M" => "Echinacea",
                      "T" => "antibiotic",
                      "W" => "calcium",
                      "Th" => "zinc",
                      "F" => "multivitamin",
                      "Sa" => "alka seltzer");

echo "<pre>";
print_r($weekly_pills);
echo "</pre>";


//assign a new pill to Thursday
$weekly_pills["Th"] = 'aspirin';

//Does Thursday correspond to index 4?  Let's see...
$weekly_pills[4] = 'garlic';

//Let's be lazy and see what happens...
$weekly_pills[] = 'glucose';

echo "<pre>";
print_r($weekly_pills);
echo "</pre>";

?>
```

Save and preview this:

```
Array
(
    [S] => vitamin C
    [M] => Echinacea
    [T] => antibiotic
    [W] => calcium
    [Th] => zinc
    [F] => multivitamin
    [Sa] => alka seltzer
)

Array
(
    [S] => vitamin C
    [M] => Echinacea
    [T] => antibiotic
    [W] => calcium
    [Th] => aspirin
    [F] => multivitamin
    [Sa] => alka seltzer
    [4] => garlic
    [5] => glucose
)
```

By the way, *all* arrays in PHP are associative. Every array value is assigned to a key index, regardless of whether we defined it. When you *don't* define a key index for an element value, PHP automatically assigns a default index to that value for you. Specifically, it assigns the next increment after the highest integer index used. That's why 'glucose' was assigned to the index **5**—we'd already used **4**.

Type the following into CodeRunner:

```
<?php

$months_of_the_year = array(1 => "January", "February", 4 => "April", 3 => "March",
                            "May", "June", "July", "August", "September", 12 => "Decemb
er",
                            10 => "October", 11 => "November");

echo "<pre>";
print_r($months_of_the_year);
echo "</pre>";

?>
```

Save it as **months.php** and preview it. Play around with it. Become one with array elements and keys. Oh, and don't forget to study your book or php.net for more fun examples.

Creating Multi-Dimensional Arrays

A multi-dimensional array is simply an array of arrays. That is, we can put arrays in for the values of an array which would be a *two-dimensional array*. A three-dimensional array would be an array of arrays of arrays. Ah, nesting. One of PHP's little joys. Let's modify our **pills.php** to see how it works.

Type the following into CodeRunner:

```
<?php

$weekly_pills = array("S" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "M" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "T" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "W" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Th" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "F" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Sa" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"));

echo "<pre>";
print_r($weekly_pills);
echo "</pre>";

echo "What pill should I pop right now? ...".$weekly_pills["Th"]["6pm"];

?>
```

Save and preview this code. Wow. That's a lot of pills! But it seems that there are enough people taking enough pills that a container indeed exists that represents this *matrix* of dosages:

Creating a *multi-dimensional array* is as simple as nesting the **array()** construct to your heart's content, to create useful representations of just about anything.

Traversing and Manipulating Arrays

Let's have some fun and send a shout-out to everyone in the **$names** array. Modify **array.php** as shown

We're feeling friendly. Type the following into CodeRunner:

```
<?php

$names = array("scott","kendell","Trish","Tony","Mike","Debra","Curt");

echo "There are ".count($names)." names in the \$names array.<br/>";
for ($i = 0; $i < count($names); $i++) {
    echo "Dialing index ".$i."...";
    echo "Hey there, ".$names[$i]."!!<br/>";
}

?>
```

Note Yet another excellent built-in PHP function is **count()**. We're sure you can guess what it does, but we still encourage you to check it out at php.net.

Preview this code and feel the love:

There are 7 names in the $names array.
Dialing index 0...Hey there, scott!!
Dialing index 1...Hey there, kendell!!
Dialing index 2...Hey there, Trish!!
Dialing index 3...Hey there, Tony!!
Dialing index 4...Hey there, Mike!!
Dialing index 5...Hey there, Debra!!
Dialing index 6...Hey there, Curt!!

Just by being friendly, you've *traversed* an array. Traversing simply requires that you hopscotch through all the elements of your array and do something with each value. "For" and "while" loops are great for that, especially when you use numerical indices.

Traversing Associative Arrays with list() and each()

Here's one guarantee: you're going to use arrays a *lot*. You can create, access, traverse, and manipulate arrays fairly easily IF you know exactly *what* is going into them, *how many* elements they have, and *how deep* the nesting goes in every case. But most of the time, you won't know all that. You'll need to work around any gaps with some nifty programming or some great built-in PHP array functions, like **count()**.

For instance, how would you traverse the associative **$weekly_pills** array? Using numerical counters won't help. But don't worry, you have options. Here's our recommended way to do it:

Type the following into CodeRunner:

```
<?php

$weekly_pills = array("Sunday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Monday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Tuesday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Wednesday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Thursday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Friday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"),
                      "Saturday" => array("8am" => "vitamin C",
       "1pm" => "antibiotic",
       "6pm" => "zinc",
       "11pm" => "alka seltzer"));

while (list($key, $value) = each($weekly_pills)) {
   echo "Here's what you should take on ".$key.":<br/>";

   echo "<pre>";
   print_r($value);
   echo "</pre>";
}

?>
```

Save and preview it:

Here's what you should take on Sunday:

```
Array
(
    [8am] => vitamin C
    [1pm] => antibiotic
    [6pm] => zinc
    [11pm] => alka seltzer
)
```

How did you get all that output? Well, there are two built-in functions working together here.

Let's break it down:

```
while (list($key, $value) = each($weekly_pills)) {
   echo "Here's what you should take on ".$key.":<br/>";
.
.
.
}
```

list() is not really considered a *function*, but a *language construct*, because it doesn't follow the normal "Parameter in/Return value out" function rule. **list()** is simply a shortcut which, when used with the assignment operator (**=**) and an **array**, assigns each value of that array to the parameter variables within **list()**.

In other words, this:

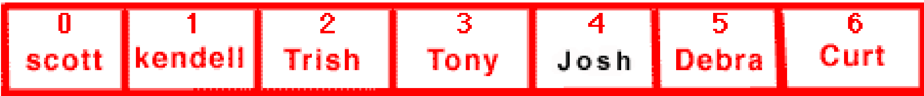
```
list($parameter1, $parameter2, $parameter3) = array("value1", "value2", "value3"
);
```

...is the same as this:

```
$parameter1 = "value1";
$parameter2 = "value2";
$parameter3 = "value3";
```

Now let's go on to **each()**, which may be even trickier than **list()**. Trickier, because it introduces an aspect of arrays that we haven't discussed until now: the *array cursor*.

Take a look at the graphical representation of **$names** again:



Now, take your mouse cursor and *point* to each box, one by one, starting with the first entry. You've just demonstrated the way an array cursor works: it *points* to array elements. The array cursor always begins by pointing to an array's first element, and stays where it is until moved by a built-in PHP function.

Here's where **each()** comes in:

Type the following into CodeRunner:

```
<?
$test_array = array("key1" => "value1",
      "key2" => "value2",
      "key3" => "value3");

//start with the beginning
$new_array1 = each($test_array);

echo "<pre>";
print_r($new_array1);
echo "</pre>";


?>
```

Save it as **each.php** and preview it:

```
Array
(
    [1] => value1
    [value] => value1
    [0] => key1
    [key] => key1
)
```

As you may have guessed, **each()** takes an array as its parameter. But what you may *not* have guessed is that it also has an array as its return value. Only the array returned is different from the array passed in.

each() uses the array cursor to access the element currently being pointed to by that cursor. This is called the *current element*. In our above example, the current element is the first element of **$test_array**. After accessing the element, **each()** creates a *new* array with four elements—using the key and value from the current element of the parameter array—and *returns* that array. In our example, we assigned that array to **$new_array1**. Finally, **each()** *increments* the array cursor so it points to the next element in the array.

Why four elements in the return array? So that the new array can be accessed both numerically AND associatively. The **key** of the parameter array's current element becomes the **value** for two of the new array's elements, accessed by the keys **0** and **"key"**. The **value** of the parameter array's current element becomes the **value** for the other two elements of the new array, accessed by the keys **1** and **"value"**.

Since **list()** can only deal with numerical keys (it ignores associative keys), the four-element return array is especially handy.

Let's put it all together:

```
while (list($key, $value) = each($weekly_pills)) {
   echo "Here's what you should take on ".$key.":<br/>";
.
.
.
}
```

In this example, the "while" loop is monitoring the cursor of our **$weekly_pills** array. We can trust that the loop won't be infinite because of **each()**. The array cursor will eventually reach the end of the array and point to **null**, but each time it loops, the current element's key (in this case, the day of the week) would be assigned to the variable **$key**. Similarly, the current element's value (in this case, another array containing that day's pills) would be assigned to the variable **$value**.

Yikes! That's not just tricky, that's downright eye-crossing. When you do get the hang of it though, this little PHP concoction will serve you well, not only with arrays, but with SQL commands and lots of other programming.

Note As an alternative to using list() and each() inside the condition of a while loop, check out foreach() loops at php.net.

More built-in functions

How do you know if an element exists in an array? What if you need distinct array elements? How about sorting and merging? All these questions can be answered with built-in PHP functions. Like we said earlier, it would take ages to go through them all, but we should definitely go over some of the major ones.

To cap off our intensive array workout, we leave you with a montage of fun PHP functions. Play, experiment, and refer back to your book or to php.net often. Think about how the functions work. Are array cursors used? What are the parameters? What is the function returning?

Finally, think about how you would write your own PHP functions to perform the same tasks. Would you make the same choices as the PHP folks?

Type the following into CodeRunner:

```
<?php

$scotts_phonebook = array("kendell" => "555-1234",
    "Trish" => "555-2345",
    "Tony" => "555-3456",
    "Mike" => "555-4567",
    "Debra" => "555-5678",
    "Curt" => "555-6789");

$kendells_phonebook = array("scott" => "555-7890",
    "Trish" => "555-2345",
    "Tony" => "555-3456",
    "Debra" => "555-5678",
    "Kate" => "555-8901");

//here's a phonebook combining both Scott's and Kendell's contacts, no duplicates

$combined_phonebook = array_unique(array_merge($scotts_phonebook, $kendells_phonebook))
;

echo "<pre> Combined Phonebook:";
print_r($combined_phonebook);
echo "</pre>";


//sort by name - why do you suppose we aren't assigning the return value to anything?

ksort($combined_phonebook);

echo "<pre>Sorted Phonebook:";
print_r($combined_phonebook);
echo "</pre>";


//here's a phonebook containing only mutual friends of Scott and Kendell

$mutual_friends = array_intersect($scotts_phonebook, $kendells_phonebook);

echo "<pre>Mutual Friends:";
print_r($mutual_friends);
echo "</pre>";


//in this custom function called "invite_friend," a phone number is
//called and that friend is invited to a party.

function invite_friend($phone_number, $name) {
   echo "Calling phone number $phone_number...";
   echo "Hello $name! You're invited to a party!<br/>";
}


//Here's a REALLY tricky built-in function we can use to invite ALL friends to the part
y.
//Careful, this one has lots of rules regarding the second parameter.

array_walk($combined_phonebook, 'invite_friend');

//Finally, generate a random phone number and see if it's in the phonebook.

$random_phonenumber = "555-".strval(rand(1000,9999));

if (in_array($random_phonenumber, $combined_phonebook)) {
   echo "Phone number ".$random_phonenumber." is in the phonebook.";
}
else {
```

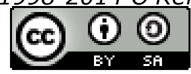
```
    echo "Phone number ".$random_phonenumber." is not in the phonebook.";
}

?>
```

Save it as **phonebooks.php** and preview.

Were you able to figure them out? If not, give yourself some time and don't stress—remember, these are functions built by someone else to save time. If any built-in function doesn't suit your purpose, look for another one...or just write one yourself.

Don't forget to Save your work! And be sure to work on the assignments in your syllabus when you're done here. See you at the next lesson...

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Strings

Welcome back. So, you already know that **strings** are one type of PHP **variable**. And you've been using strings throughout the last five lessons with **echo**, the concat operator (**.**), and in all kinds of functions and loops. You've got strings down, baby.

So why spend an entire lesson on the letters, numbers, and symbols that make up strings?

The truth is, we've only explored the tip of the proverbial iceberg when it comes to strings. In fact, they are the cornerstones of many a web-based, database-driven application. Like piranha, you should never underestimate the feisty little critters.

So get your waist-high galoshes on, fire up **PHP** in CodeRunner, and let's get cracking.

What's a String Anyway?

And what is it hiding from us? String *is* its real name, right? Let's see what's going on here. Remember our LAMP acronym?:

Type the following into a new PHP file in CodeRunner:

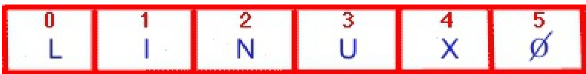
```
<?php

$lamp_l = "Linux";
$lamp_a = "Apache";
$lamp_m = "MySQL";
$lamp_p = "PHP";

echo "<br/>The stack begins with ".$lamp_l.", and goes on to include "
    .$lamp_a.", ".$lamp_m.", and ".$lamp_p."!<br/>";


//These supposedly simple strings are hiding something...

echo "Gimme an L!  ".$lamp_l[0]."!<br/>";
echo "Gimme an A!  ".$lamp_a[0]."!<br/>";
echo "Gimme an M!  ".$lamp_m[0]."!<br/>";
echo "Gimme a P!  ".$lamp_p[0]."!<br/>";



?>
```

Save it as **strings.php**, then click Preview. You should see this:

The stack begins with Linux, and goes on to include Apache, MySQL, and PHP!
Gimme an L! L!
Gimme an A! A!
Gimme an M! M!
Gimme a P! P!

Wait a minute. Why were we just able to use the array operator **[]** to access the first letters of the LAMP acronym? Aha, now the truth comes forth.

That sneaky string doesn't want you to know it has a secret identity. You see, a **string** is a special type of *array*, one where each **character** --letter, number, symbol, newline, whatever takes up one byte of space -- is assigned a numerical key index. Here's what the string "Linux" would look like in our pillbox representation from the arrays lesson:



Note The last box you see contains simply the **NULL** character, which has always been used to *terminate* strings in the C language - the language PHP is based upon. (Check out the history of PHP.)

Manipulating Strings

Let's explore strings further. We're going to make a new PHP file called **bologna.php**.

Type the following into CodeRunner:

```
<?php


function spell_me($mystring) {
   $i = 0;
   while ($mystring[$i] != null) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
      $i++;
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';


//Sing along if you remember the commercial!

echo "My ".$string_1." has a first name, it's ";
spell_me($string_2);
echo "<br/>";


echo "My ".$string_1." has a second name, it's ";
spell_me($string_3);
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

echo "That ".$string_2." ".$string_3." has a way<br/>
      With ";
spell_me($string_1);
echo "!";



?>
```

Note For reference here's the Oscar Mayer bologna song

Preview for the lyrics of the song:

My bologna has a first name, it's o - s - c - a - r
My bologna has a second name, it's m - a - y - e - r
Oh I love to eat it every day,
And if you ask me, why I'll say...
That oscar mayer has a way
With b - o - l - o - g - n - a!

Through the power of commercial jingles, we're able to uncover two more truths about strings: not only can we *access* the characters within a string using the **[]** operator, but we can use the same operator to *traverse* and *manipulate* strings, just like arrays.

Take another look:

```
function spell_me($mystring) {
   $i = 0;
   while ($mystring[$i] != null) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
      $i++;
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';

.
.
.
```

In our function **spell_me()**, we used a **while loop** to *traverse* the string parameter, stopping when we reached the null character. Then we *manipulated* **$string_3** by *assigning* new characters to the indices we wanted to change. In no time, "oscar" turned to "mayer," and all were spelled correctly.

Go ahead, keep humming the tune - we don't mind.

Other nifty string shortcuts

Type the BLUE stuff into your document in CodeRunner:

```
<?php

function spell_me($mystring) {
   $i = 0;
   while ($mystring[$i] != null) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
      $i++;
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';


//Sing along if you remember the commercial!

echo "My $string_1 has a first name, it's ";  //we took out the concat operators
spell_me($string_2);
echo "<br/>";


echo "My $string_1 has a second name, it's ";
spell_me($string_3);
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

echo "That $string_2 $string_3 has a way<br/>
      With ";
spell_me($string_1);
echo "!";


?>
```

Preview this. Nothing changed, right? This is a cool shortcut created especially for strings in PHP, called *embedding variables*. Since you'll most likely use PHP in web pages, you can thank us later for showing you this shortcut. It provides a more intuitive method of creating and outputting dynamic strings without the need for all those annoying concat operators and quotation marks.

There's only one small complication with this shortcut. What happens if you want to display an actual dollar sign (**$**) along with all the embedded variables?

Type the following into CodeRunner:

```
<?php

function spell_me($mystring) {
   $i = 0;
   while ($mystring[$i] != null) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
      $i++;
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';


//Sing along if you remember the commercial!

echo "My $string_1 has a first name, it's ";  //we took out the concat operators
spell_me($string_2);
echo "<br/>";


echo "My $string_1 has a second name, it's ";
spell_me($string_3);
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

echo "That $string_2 $string_3 has a way<br/>
      With ";
spell_me($string_1);
echo "!";

echo "<br/>As you can see, $string_1 was passed into the parameter $mystring.";


?>
```

You should get something like this:

My bologna has a first name, it's o - s - c - a - r
My bologna has a second name, it's m - a - y - e - r
Oh I love to eat it every day,
And if you ask me, why I'll say...
That oscar mayer has a way
With b - o - l - o - g - n - a!
As you can see, bologna was passed into the parameter .

Well, that's a bunch of bologna. While we wanted to output the actual variable names, the **echo** command tried to replace them with their values instead. This happens anytime echo sees a dollar sign (**$**) followed by something that could pass as a variable name.

How can we stop it? *Escape* it.

Type the following into CodeRunner:

```
<?php

function spell_me($mystring) {
   $i = 0;
   while ($mystring[$i] != null) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
      $i++;
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';


//Sing along if you remember the commercial!

echo "My $string_1 has a first name, it's ";  //we took out the concat operators
spell_me($string_2);
echo "<br/>";


echo "My $string_1 has a second name, it's ";
spell_me($string_3);
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

echo "That $string_2 $string_3 has a way<br/>
      With ";
spell_me($string_1);
echo "!";

echo "<br/>As you can see, \$string_1 was passed into the parameter \$mystring."
;


?>
```

You should get this:

My bologna has a first name, it's o - s - c - a - r
My bologna has a second name, it's m - a - y - e - r
Oh I love to eat it every day,
And if you ask me, why I'll say...
That oscar mayer has a way
With b - o - l - o - g - n - a!
As you can see, $string_1 was passed into the parameter $mystring.

Ah, much better. Just by adding a little backslash (\) before each dollar sign (**$**), we're able to tell PHP that we really do want the name itself displayed, not the value.

That backslash is handy for *escaping* several other characters too. Refer to your book or to php.net to embed them all into your subconscious.

Built-in String Functions

"String functions?" you say, "I don't need no stinking string functions. I could use all the built-in array functions on strings too!"

While that may be true in C, PHP treats strings as a different **type** with its own set of built-in functions, generally keeping their secret identity under wraps. Try using an array function to traverse a string:

Type the following into CodeRunner:

```
<?php

function spell_me($mystring) {
   for ($i = 0; $i < count($mystring); $i++) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';


//Sing along if you remember the commercial!

echo "My $string_1 has a first name, it's ";  //we took out the concat operators
spell_me($string_2);
echo "<br/>";


echo "My $string_1 has a second name, it's ";
spell_me($string_3);
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

echo "That $string_2 $string_3 has a way<br/>
      With ";
spell_me($string_1);
echo "!";

echo "<br/>As you can see, \$string_1 was passed into the parameter \$mystring.";


?>
```

Preview it and you should get this:

My bologna has a first name, it's o
My bologna has a second name, it's m
Oh I love to eat it every day,
And if you ask me, why I'll say...
That oscar mayer has a way
With b!
As you can see, $string_1 was passed into the parameter $mystring.

Not exactly the catchiest lyrics anymore. Now try it with a built-in string function.

Type the following into CodeRunner:

```
<?php

function spell_me($mystring) {
   for ($i = 0; $i < strlen($mystring); $i++) {
      if ($i == 0) {
         echo $mystring[$i];
      }
      else {
         echo " - ".$mystring[$i];
      }
   }
}

$string_1 = "bologna";
$string_2 = "oscar";
$string_3 = $string_2;

$string_3[0] = 'm';
$string_3[1] = 'a';
$string_3[2] = 'y';
$string_3[3] = 'e';


//Sing along if you remember the commercial!

echo "My $string_1 has a first name, it's ";  //we took out the concat operators
spell_me($string_2);
echo "<br/>";


echo "My $string_1 has a second name, it's ";
spell_me($string_3);
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

echo "That $string_2 $string_3 has a way<br/>
      With ";
spell_me($string_1);
echo "!";

echo "<br/>As you can see, \$string_1 was passed into the parameter \$mystring.";


?>
```

You should get this:

My bologna has a first name, it's o - s - c - a - r
My bologna has a second name, it's m - a - y - e - r
Oh I love to eat it every day,
And if you ask me, why I'll say...
That oscar mayer has a way
With b - o - l - o - g - n - a!
As you can see, $string_1 was passed into the parameter $mystring.

See, it's not such a bad thing. Having specialized string functions means they'll be faster, easier, and more intuitive to use.

Soon we'll be working with HTML forms and dynamic inputs, which really flex the muscles of built-in PHP string functions. However, even without forms, string functions have thousands of uses. Here we have peppered our oscar mayer song with a plethora of useful string functions. Play, experiment, and refer back to your book or to php.net as much as you need. Try out the code below. Can you figure out how they all work?

Type the following into CodeRunner:

```
<?php

function spell_me($mystring) {
   for ($i = 0; $i < strlen($mystring); $i++) {
      if ($i == 0) {
         echo strtoupper($mystring[$i]);
      }
      else {
         echo " - ".strtoupper($mystring[$i]);
      }
   }
}

$string_1 = "bologna";
$string_2 = "oscar mayer";
$space_index = strpos($string_2, " ");

//let's spell boloney how we really say it...
echo "My ".str_replace('gna','ney',$string_1)." has a first name, it's ";
spell_me(substr($string_2,0,$space_index));
echo "<br/>";


echo "My $string_1 has a second name, it's ";
spell_me(substr($string_2,$space_index+1)); //notice this has only two parameters
echo "<br/>";


?>

Oh I love to eat it every day, <br/>
And if you ask me, why I'll say...<br/>


<?

//we're tired of putting in the HTML <br/> tags...
echo "That $string_2 ".nl2br("has a way
      With ");
spell_me($string_1);
echo "!";

$sale_price = 1;  //a dollar
echo "<br/>On sale for ".number_format($sale_price, 2)."!";


?>
```

Before we move on, experiment with these questions in CodeRunner:

- Does it matter whether you use single quotation marks (') or double quotation marks (") with strings?
- Can you mix the two types of quotation marks? Do you have to escape them if you do?
- Are there any built-in array functions that *do* work with strings?
- Would you have built the **substr()** function differently?

Regular Expressions

Not many subjects can make a programmer groan like that of **regular expressions**. They are immensely useful, yes - they are used to create "wildcard" strings so that you can, say, verify that someone has entered a valid email address or a correct phone number format. However, learning "Regex" patterns can sometimes feel as though you're deciphering the Rosetta Stone. Even the ever-helpful php.net pawns you off to a cryptic "man page" when dealing with Regex rules. Aargh.

But hey! We've got "learning by doing" on our side. And when we learn by doing, we can accomplish anything.

Type the following into a new PHP file in CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}

$regex_1 = "log";

$string_1 = "bologna";
check_regex($regex_1, $string_1);



?>
```

You should get:

The pattern 'log' is found in bologna.

Here we have an example of a **regular expression** -- a simple string: "log." And by using the built-in PHP function **preg_match()**, we are simply checking to see if a "log" is found in "bologna." Of course it is. Notice the quotes and forward slashes needed around the $myregex variable. These are needed because preg_match is a PERL style regex matching function and regex's in PERL must have forward slashes. See what happens if you remove the slashes.

So, you may wonder why we didn't just use the built-in string function **strpos()**. We could have. But here's where it gets interesting...

The plot thickens. Type the following into CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}


$regex_1 = "log$";

$string_1 = "bologna";
check_regex($regex_1, $string_1);


$string_2 = "catalog";
check_regex($regex_1, $string_2);


?>
```

Preview this:

The pattern 'log$' is NOT found in bologna.
The pattern 'log$' is found in catalog.

Now THIS result we could not get from **strpos**. Since when is a dollar sign (**$**) found in the word "catalog"?

As it turns out, the dollar sign has a special meaning in regular expressions, and it's different from the meaning it usually has for PHP variables. In regular expressions, placing a dollar sign (**$**) after a string means *"at the end of the string"*.

Take another look:

```
<?php

.
.
.


$regex_1 = "log$";

$string_1 = "bologna";
check_regex($regex_1, $string_1);

$string_2 = "catalog";
check_regex($regex_1, $string_2);


?>
```

Because we specified **$regex_1** to be **"log$"** and not just **"log"**, our function **check_regex()** now checks to see if **"log"** is found at the ***end*** of each of our strings. This is why it returned **true** for "catalog," but **false** for "bologna."

This is the key to regular expressions. More than just a random string tool, regular expressions are an entirely different language for creating and comparing strings with very specific patterns in mind: the presence of specific characters, the number of occurrences of each character, and their location in the string. In this case, we were concerned with the location of the string "log." Let's try another one...

Type the following into CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}


$regex_1 = "^cat";

$string_1 = "concatenate";
check_regex($regex_1, $string_1);

$string_2 = "catalog";
check_regex($regex_1, $string_2);

?>
```

The pattern '^cat' is NOT found in concatenate.
The pattern '^cat' is found in catalog.

You guessed it. Placing a carat (**^**) in front of your Regex string means *"at the beginning of the string."*

Character Ranges and Number of Occurrences

Type the following into CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}


$regex_1 = "cat.*a";

$string_1 = "concatenate";
check_regex($regex_1, $string_1);

$string_2 = "catalog";
check_regex($regex_1, $string_2);

?>
```

Preview this:

The pattern 'cat.*a' is found in concatenate.
The pattern 'cat.*a' is found in catalog.

Whoa. That is a crazy Regex pattern. Yet it was found in the strings "concatenate" AND "catalog." What gives?

And speaking of concatenate, isn't that period (**.**) the concatenate operator in PHP? Not this time. Just like the dollar sign (**$**), the period (**.**) has a much different meaning when it comes to regular expressions. In this case, it represents *any* character, like a wildcard.

As for the asterisk (*****), that means *"zero or more"*. Put it all together, and the regular expression **"cat.*a"** means *"the string 'cat,' followed by zero or more characters, followed by an 'a'"*.

Is that found in **"concatenate"**? Yes: the string **'cat'** is found, followed by two characters **'e'** and **'n'**, followed by an **'a'**. How about **"catalog"**? Yep: **'cat'** is followed by zero characters, followed by an **'a'**. Let's try a REALLY crazy Regex:

Type the following into CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}


$regex_1 = "cat(a|e)+[a-z]{2,5}";

$string_1 = "concatenate";
check_regex($regex_1, $string_1);

$string_2 = "catalog";
check_regex($regex_1, $string_2);

$string_3 = "catamaran";
check_regex($regex_1, $string_3);

$string_4 = "scathing";
check_regex($regex_1, $string_4);

$string_5 = "pontificates";
check_regex($regex_1, $string_5);


?>
```

Preview this:

The pattern 'cat(a|e)+[a-z]{2,5}' is found in concatenate.
The pattern 'cat(a|e)+[a-z]{2,5}' is found in catalog.
The pattern 'cat(a|e)+[a-z]{2,5}' is found in catamaran.
The pattern 'cat(a|e)+[a-z]{2,5}' is NOT found in scathing.
The pattern 'cat(a|e)+[a-z]{2,5}' is NOT found in pontificates.

Now, this may seem overwhelming, but it's actually just a series of simple Regex patterns. Let's break them down:

cat(a|e)+[a-z]{2,5}

- **(a|e)**: The pipe character (**|**) in regular expressions means ***OR***, so in this case we're looking for *"either an 'a' or an 'e'"*. Parentheses(**()**) are used to separate out expressions when we are *nesting* them, just like always.
- **+**: The plus sign (**+**) is just like the asterisk (*****), except it's looking for **one or more** of the characters it follows. Since we preceded it with the expression **(a|e)**, in this case it means *"one or more of either 'a' or 'e'"*.
- **[a-z]**: To allow entire ranges of characters as a shortcut, we use square brackets (**[]**) and the dash (**-**). So in this case, we're looking for *"any lowercase letter from 'a' to 'z'"*.
- **{2,5}**: Curly braces (**{}**) are used like the plus sign and asterisk, indicating a range of occurrences of the preceding expression. In this case, because **{2,5}** follows **[a-z]**, we're looking for *"2 to 5 occurrences of any lowercase letter from 'a' to 'z'"*.

Put it all together, and we find that the pattern **cat(a|e)+[a-z]{2,5}** in Regex-speak means *"The string **'cat'**, followed by **one or more 'a**'s or **'e'**s, followed by **at least 2, but not more than 5** lowercase letters, from **'a'** to **'z'**."*

Can you figure out why it's not found in **"scathing"** or **"pontificates"**?

Excluding Characters

Now, if you thought THAT was confusing, consider this: What if you had the all-important task of, say, *removing* funky characters from a file name and *replacing* them with something benign? Here's where things REALLY get screwy.

Type the following into CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}


//here's a function that takes in a file name, and replaces all funky characters
 with an underscore "_"

function clean_filename($file_name) {
   $bad_characters = "[^a-zA-Z0-9.]";
   $new_filename = preg_replace("/$bad_characters/", "_", $file_name);
   return $new_filename;
}

$bad_filename = "file[3*1 name.doc";

$good_filename = clean_filename($bad_filename);

echo "'$bad_filename' has been changed to '$good_filename'.";


?>
```

Preview this:

'file[3*1 name.doc' has been changed to 'file_3_1_name.doc'.

We know, we know, this makes no sense at all. First of all, the carat (**^**) is supposed to mean *"at the beginning of the string."* The period (**.**) is supposed to represent a wildcard character. And what's with the ranges of characters -- **a-z**, **A-Z**, and **0-9** -- stuck together like that? Groan.

Well, as it turns out, when it comes to whatever's in the square brackets (**[]**), the rules change. Let's take a closer look at brackets in regular expressions.

[^a-zA-Z0-9.]

- **^**: When used within square brackets, the carat (**^**) *negates* everything after it - just like the exclamation point(**!**) in PHP. So in this case it's looking for characters that DON'T match what's inside the brackets.
- **a-zA-Z0-9**: Everything within square brackets comes together to represent only one character. Therefore, characters placed within the brackets are treated as if a pipe character (**|**) was inserted in between each one. For instance, **[abcd]** is the same as **(a|b|c|d)**, and in this case, **a-zA-Z0-9** is the same as **([a-z]|[A-Z]|[0-9])**, or more simply, *"any alphanumeric character"*.
- **.**: Within square brackets, every character except for the carat(**^**), the dash(**-**), and the right bracket itself(**]**) is taken as a literal character - including the period(**.**) that would normally be considered a wildcard character.

Put it all together, and we find that the pattern **[^a-zA-Z0-9.]** actually means *"any character which is NOT an alphanumeric character or a period."*.

Escaping Characters

Regular Expressions are extremely useful in PHP - especially since you'll be doing a lot of HTML form processing. For instance, how can you ensure that someone's entered their phone number properly?

Type the following into CodeRunner:

```
<?php

//here's a simple function to check Regex patterns against string parameters

function check_regex($myregex, $mystring) {
   if (preg_match("/$myregex/", $mystring)) {
      echo "The pattern '$myregex' is found in $mystring.<br/>";
   }
   else {
      echo "The pattern '$myregex' is NOT found in $mystring.<br/>";
   }
}

//here's a function that takes in a file name, and replaces all funky characters
 with an underscore "_"

function clean_filename($file_name) {
   $bad_characters = "[^a-zA-Z0-9.]";
   $new_filename = preg_replace("/$bad_characters/", "_", $file_name);
   return $new_filename;
}


//here's a function which validates an American phone number

function validate_phone($phone_number) {
   $good_phone = "^\(?[0-9]{3}\)?( |-|\.)[0-9]{3}(-|\.)[0-9]{4}$";

   if (preg_match("/$good_phone/", $phone_number)) {
   echo "$phone_number is valid.<br/>";
   }
   else echo "$phone_number is NOT valid.<br/>";
}

$phone_number1 = "34x.d98.1123";
validate_phone($phone_number1);

$phone_number2 = "(217) 555-1212";
validate_phone($phone_number2);


?>
```

Preview this:

34x.d98.1123 is NOT valid.
(217) 555-1212 is valid.

Let's break this down: **^\(?[0-9]{3}\)?(|-|\.)[0-9]{3}(-|\.)[0-9]{4}$**

- **^**: Since we're outside any square brackets, the carat (**^**) takes on its original meaning—*"at the beginning of the string."* By the same token, we use the dollar sign (**$**) to mean "*at the end of the string,*" so that we have an exact match.
- **\(?**: Some Americans use parentheses (**()**) to enclose the 3-digit area code of their phone numbers. To allow this possibility, we use the question mark (**?**) much like the asterisk or plus sign, only this time to denote *"zero or one"* of the leading parenthesis (**(**). However, because parentheses usually mean *nesting*, we use a backslash (****) to *escape* the character. This must always be done when not within the square brackets. The same is true with **\)?**.
- **[0-9]{3}**: Since the area code of the American phone number system uses exactly 3 digits, we use **{3}** to require exactly **3** of any digit, denoted by **[0-9]**. We use the same logic with the 3-digit prefix, as well as the ending 4 digits of the phone number.

- **(|-|\.)** Usually between the area code and the prefix of a phone number, folks use either a space (" "), a dash (**-**), or a period(**.**). Therefore, we use the parentheses(**()**) along with the pipe character (**|**) to say *"either a space or a dash or a period."* We put a backslash before the period because we must escape it. An unescaped period matches a single character without caring what that character is. So since we want a literal period, we add the backslash to "escape" the character having that special meaning.

In case your eyes are crossing right now, remember that **regular expressions** take a lot of patience, practice, and trial-and-error to get right. Refer back to this lesson, to books you may have, or to the web, often. Here's a great article on regular expressions in PHP.

Whew! We've covered a lot of ground. Don't forget to **Save** your work and hand in your **assignments** from your syllabus. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Fixing Broken PHP

This lesson is all about frustration. Frustration that comes in the form of **parse errors**, **infinite loops**, **unmatched brackets**, and **logical mistakes**. Frustration that makes your face grimace and your fingers type furiously, pounding as if sheer force could will the keys into assembling proper PHP code from the mangled mess that is your own program. Ah yes, we know this feeling well.

In previous lessons we focused on the basics of PHP, keeping examples and projects to finite blocks of code. We're sure you've handled the frustrating errors like a trooper so far. In the upcoming lessons, we'll begin constructing more complex programs to solve real-world problems, which means the threat of frustration looms larger than ever. You're going to need some serious ammo for creating and debugging scalable programs. Your sanity just may depend on it.

So let's take a break from new PHP concepts and focus on technique for a while. Got CodeRunner in **PHP** syntax? Good - let's get going.

Things Professors Don't Talk About Enough

We're guilty of it too. We introduce you to concepts that theoretically work just fine, assuming that everything typed in just so, and that we've explained the concept perfectly. So of *course* these concepts will work perfectly for you, every time you apply them, right?

Let's find out using the following silly program we assembled from concepts covered in previous lessons.

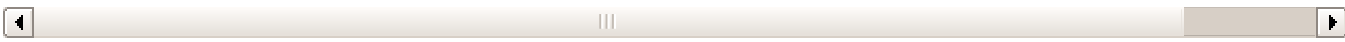
It's okay to copy and paste, JUST THIS ONCE! Paste this into CodeRunner:

```
<?
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number;) {
       echo $the_sound."... ";
   }
}

function Mood_Chant($my_mood){
   if ($my_mood == "happy") {
      mantra("OM");
      $after_chant = "<br/>I feel serene now.";
   }
   else if ($my_mood == "sad") {
      mantra("okay");
      $after_chant = "<br/>I feel better now.";
   }
   else if ($my_mood == "angry") {
      mantra("mississippi");
      $after_chant = "<br/>Ahhh, much better.  I've calmed down now.";
   }
   else if ($my_outlook == "indifferent") {
      mantra("Wake up");
      $after_chant = "<br/>I'm awake now.";
   }
   else {
      mantra("Try harder");
      $after_chant = "<br/>I'll try harder now.";
   }
   return $after_chant;
}

function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $my_cash) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $my_cash ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $my_cash && $cereal_prices['Fruit_Loops'
] < $my_cash ){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
<h3>The emotional roller-coaster of buying breakfast cereal</h3>
<?
$cereal_prices = array('Captain_Crunch' => 5, 'Fruit_Loops' => 3);
$my_cash = 4;
?>
I have $<? echo $my_cash; ?> in my pocket, and I want to buy some Captain Crunch!<br/>
<ul>
<li>Captain Crunch costs $<? echo $cereal_prices['Captain_Crunch']; ?></li>
<li>Fruit Loops costs $<? echo $cereal_prices['Fruit_Loops']; ?></li>
</ul>

$my_mood = whats_my_emotion($cereal_prices, $my_cash);
if (!($my_mood == "sad")) {
```

```
    $after_chant_mood = Mood_Chant($my_mood, $chant_number);
}
echo "<br/>".$after_chant_mood;

?>
```

Preview this:

Parse error: syntax error, unexpected ';' in **/users/certjosh/useractivepreviewtmp123.ph**
line **4**



Okay, WRONG. Even a benign-looking program like the above works beautifully in theory, but *never* in practice -- at least for the first hundred times you try it. Trust us. Do you think our examples worked perfectly the first time we wrote them? Hardly.

But even with this relatively small amount of code, where do you begin to debug it? Here is where you're usually on your own...

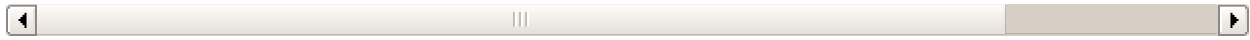
...but hey, not in this lesson! We're not too far removed from our humble beginnings to know how hard it is to master debugging. Consider this a support group for frustrated coders, and you're invited.

Debugging Tips

Utilizing Error Messages

Let's Preview again:

Parse error: syntax error, unexpected ';' in **/users/certjosh/useractivepreviewtmp**
line **4**



If you're lucky, you'll get an easy to see error message right away, like you're getting now. In other situations you may have to ask your system administrator where she keeps the PHP error logs. In any case, the First Rule of Debugging is: *check the error messages first.* They may seem cryptic at times, but they almost always give you the information you need. In particular, the **line number** where the problem is located.

Since our error message indicated **line 4**, go to that line. What do you see?

Suddenly, our **parse error** is as loud as a mariachi band. There shouldn't be a semicolon (**;**) inside the parentheses (**()**) of a **while loop**! This is easy enough to fix: just remove the semicolon (**;**).

Good for you! You fixed the error, and now everything should work perfectly, right?

Riddle-Me-This Error Messages

Cross your fingers and Preview again:

Parse error: syntax error, unexpected '<' in **/users/certjosh/useractivepreviewtm**
line **53**

Yikes, another error message! Mild frustration ensues. Well, no problem, we'll just do the same thing we did before. But this time try the Go To Line button. Type in line **53**:

Hmm, that's strange. This isn't even PHP code, it's HTML code -- and *perfect* HTML code, at that. Why would PHP single out a line of good HTML code in its error message?

We're going to have to look around for more clues, which brings us to the Second Rule of Debugging: *Check lines closest to the error message second.* Let's give that a shot, by looking at line **52**:

And there you have it - a tiny curly bracket (**}**), indicative of PHP code. Were you able to solve the riddle of the error message? We forgot to *delimit* the PHP with a **?>** between the PHP code and the HTML code, so the PHP parser was attempting to read the HTML code as PHP! Hence the message: "unexpected '<' on line **53**". It didn't know any better.

Go ahead and add the delimiter (**?>**), and you have squashed another bug in our program. Let's hope that's the last one.

Errors without Error Messages

Chant 'no error messages' three times, then Preview again:

The emotional roller-coaster of buying breakfast cereal

I have $4 in my pocket, and I want to buy some Captain Crunch!

- Captain Crunch costs $5
- Fruit Loops costs $3

$my_mood = whats_my_emotion($cereal_prices, $my_money); if (!($my_mood == "
$after_chant_mood = Mood_Chant($my_mood, $chant_number); } echo "
".$after_chant_mood; ?>

Hey, the chant worked - no error messages! But wait - there's still an error. Looks like "no error messages" is not the same as "no errors". Which brings us to the Third Rule of Debugging: *When there are no error messages, check your output.* Or, just work on your chant.

Look at your Preview again. It seems that the trouble starts right after the statement: *"Fruit Loops costs $3"*. After that, chaos erupts. So let's take a look at our code now, and try to pinpoint the problem.

Pay attention to the errors we already fixed, and where the new error seems to be happening:

```
<?
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number) { //first we took out the rogue semicolon...
       echo $the_sound."... ";
   }
}

function Mood_Chant($my_mood){
   if ($my_mood == "happy") {
      mantra("OM");
      $after_chant = "<br/>I feel serene now.";
   }
   else if ($my_mood == "sad") {
      mantra("okay");
      $after_chant = "<br/>I feel better now.";
   }
   else if ($my_mood == "angry") {
      mantra("mississippi");
      $after_chant = "<br/>Ahhh, much better.  I've calmed down now.";
   }
   else if ($my_outlook == "indifferent") {
      mantra("Wake up");
      $after_chant = "<br/>I'm awake now.";
   }
   else {
      mantra("Try harder");
      $after_chant = "<br/>I'll try harder now.";
   }
   return $after_chant;
}

function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $my_cash) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $my_cash ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $my_cash && $cereal_prices['Fruit
_Loops'] < $my_cash ){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
//then we added the delimiter here...
?>
<h3>The emotional roller-coaster of buying breakfast cereal</h3>
<?
$cereal_prices = array('Captain_Crunch' => 5, 'Fruit_Loops' => 3);
$my_cash = 4;
?>
I have $<? echo $my_cash; ?> in my pocket, and I want to buy some Captain Crunch
!<br/>
<ul>
<li>Captain Crunch costs $<? echo $cereal_prices['Captain_Crunch']; ?></li>
<li>Fruit Loops costs $<? echo $cereal_prices['Fruit_Loops']; ?></li>
</ul>
```

```
//BUT NOW THE PROBLEM APPEARS TO BE HERE

$my_mood = whats_my_emotion($cereal_prices, $my_cash);
if (!($my_mood == "sad")) {
  $after_chant_mood = Mood_Chant($my_mood, $chant_number);
}
echo "<br/>".$after_chant_mood;

?>
```

Looking at the code, we see now that we've done it again - we've forgotten a **delimiter**, this time an opening delimiter (**<?**). Why didn't we get an error message like before? Because this time the code went from HTML to PHP - so it was HTML attempting to render the PHP code, not the other way around. HTML is more forgiving in this sense, and simply prints out the code.

Be sure to add the delimiter **<?**. Are we done now?

Logical Errors

Signs point to no:

The emotional roller-coaster of buying breakfast cereal

I have $4 in my pocket, and I want to buy some Captain Crunch!

- Captain Crunch costs $5
- Fruit Loops costs $3

Oh well, I'm going home.

At first glance, everything appears to be correct. No error messages, no garbled output. But before you breathe that sigh of relief, remember that this silly program is supposed to determine our mood and purchasing behavior, based on cereal prices and how much money we have.

Take another look at the code:

```
function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $my_cash) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $my_cash ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $my_cash && $cereal_prices['Fruit
_Loops'] < $my_cash){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
```

From our output, we see that we have $4 -- not enough to buy Captain Crunch for $5, but enough to buy Fruit Loops for $3. In our program, that's supposed to make us **angry**, but we'd rather invoke a calming mantra chant and buy Fruit Loops anyway. So why are we dejected and going home?

This is called a **logical error**, and unfortunately it seems that the output isn't helping us much in the way of

clues to fix it. Which brings us to the Fourth Rule of Debugging: *When the output doesn't show the error, create strategic output that does.*

Type the following green code into CodeRunner:

```
<?
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number) { //first we took out the rogue semicolon...
       echo $the_sound."... ";
   }
}

function Mood_Chant($my_mood){
   if ($my_mood == "happy") {
      mantra("OM");
      $after_chant = "<br/>I feel serene now.";
   }
   else if ($my_mood == "sad") {
      mantra("okay");
      $after_chant = "<br/>I feel better now.";
   }
   else if ($my_mood == "angry") {
      mantra("mississippi");
      $after_chant = "<br/>Ahhh, much better.  I've calmed down now.";
   }
   else if ($my_outlook == "indifferent") {
      mantra("Wake up");
      $after_chant = "<br/>I'm awake now.";
   }
   else {
      mantra("Try harder");
      $after_chant = "<br/>I'll try harder now.";
   }
   return $after_chant;
}

function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $my_cash) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $my_cash ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $my_cash && $cereal_prices['Fruit
_Loops'] < $my_cash ){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
//then we added the delimiter here...
?>
<h3>The emotional roller-coaster of buying breakfast cereal</h3>
<?
$cereal_prices = array('Captain_Crunch' => 5, 'Fruit_Loops' => 3);
$my_cash = 4;
?>
I have $<?
//We know it's not here, because the output has been correct
echo $my_cash; ?> in my pocket, and I want to buy some Captain Crunch!<br/>
<ul>
<li>Captain Crunch costs $<? echo $cereal_prices['Captain_Crunch']; ?></li>
<li>Fruit Loops costs $<? echo $cereal_prices['Fruit_Loops']; ?></li>
```

```
</ul>

<?
//then we added the delimiter here...

$my_mood = whats_my_emotion($cereal_prices, $my_cash);
//Let's add some echo statements to figure out our logic.
echo "\$my_mood is $my_mood";

if (!($my_mood == "sad")) {
  $after_chant_mood = Mood_Chant($my_mood, $chant_number);
}
echo "<br/>".$after_chant_mood;

?>
```

Preview this:

The emotional roller-coaster of buying breakfast cereal

I have $4 in my pocket, and I want to buy some Captain Crunch!

- Captain Crunch costs $5
- Fruit Loops costs $3

Oh well, I'm going home.$my_mood is sad

So we find that when the function **whats_my_emotion()** returns, its value is **sad**, not angry. Since we know the values of **$my_cash** and **$cereal_prices** are correct, it looks like we've narrowed the problem down to **whats_my_emotion()**. Now what do we do?

Let's add some more echo statements - but this time use them *within* **whats_my_emotion()**, just to see what happens.

Add the following green code into CodeRunner:

```
<?
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number) { //first we took out the rogue semicolon...
       echo $the_sound."... ";
   }
}

function Mood_Chant($my_mood){
   if ($my_mood == "happy") {
      mantra("OM");
      $after_chant = "<br/>I feel serene now.";
   }
   else if ($my_mood == "sad") {
      mantra("okay");
      $after_chant = "<br/>I feel better now.";
   }
   else if ($my_mood == "angry") {
      mantra("mississippi");
      $after_chant = "<br/>Ahhh, much better.  I've calmed down now.";
   }
   else if ($my_outlook == "indifferent") {
      mantra("Wake up");
      $after_chant = "<br/>I'm awake now.";
   }
   else {
      mantra("Try harder");
      $after_chant = "<br/>I'll try harder now.";
   }
   return $after_chant;
}

function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $my_cash) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $my_cash ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $my_cash && $cereal_prices['Fruit
_Loops'] < $my_cash ){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      //We know the output is coming from here, so let's add echo statements:
      echo "Within whats_my_emotion, \$cereal_prices:<pre>";
      print_r($cereal_prices);
      echo "\$my_cash = ".$my_cash."</pre>";


      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
//then we added the delimiter here...
?>
<h3>The emotional roller-coaster of buying breakfast cereal</h3>
<?
$cereal_prices = array('Captain_Crunch' => 5, 'Fruit_Loops' => 3);
$my_cash = 4;
?>
```

```
I have $<?
//We know it's not here, because the output has been correct
echo $my_cash; ?> in my pocket, and I want to buy some Captain Crunch!<br/>
<ul>
<li>Captain Crunch costs $<? echo $cereal_prices['Captain_Crunch']; ?></li>
<li>Fruit Loops costs $<? echo $cereal_prices['Fruit_Loops']; ?></li>
</ul>

<?
//then we added the delimiter here...

$my_mood = whats_my_emotion($cereal_prices, $my_cash);
//Let's add some echo statements to figure out our logic.
echo "\$my_mood is $my_mood";

if (!($my_mood == "sad")) {
  $after_chant_mood = Mood_Chant($my_mood, $chant_number);
}
echo "<br/>".$after_chant_mood;

?>
```

Preview this:

The emotional roller-coaster of buying breakfast cereal

I have $4 in my pocket, and I want to buy some Captain Crunch!

- Captain Crunch costs $5
- Fruit Loops costs $3

Within whats_my_emotion, $cereal_prices:

```
Array
(
    [Captain_Crunch] => 5
    [Fruit_Loops] => 3
)
$my_cash =
```

Oh well, I'm going home.$my_mood is sad

This is starting to look pretty messy, but it does tell us everything we need to know. Through our echo and print_r statements, we can see that the **logical error** is definitely *within* **whats_my_emotion()**. Why? Because the parameter **$my_cash** was never properly passed in -- causing the resulting mood to be **sad** instead of **angry**.

Take a closer look at whats_my_emotion():

```
function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $my_cash) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $my_cash ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $my_cash && $cereal_prices['Fruit
_Loops'] < $my_cash ){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      //We know the output is coming from here, so let's add echo statements:
      echo "Within whats_my_emotion, \$cereal_prices:<pre>";
      print_r($cereal_prices);
      echo "\$my_cash = ".$my_cash."</pre>";

      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
```

Looking at our **if/else statements**, we can see that we make all kinds of comparisons between **$my_cash** and the various cereal prices, yet no matter what we set **$my_cash** to before we pass it to **whats_my_emotion()**, it comes up blank *within* **whats_my_emotion()**. And then the error becomes clear: *within* **whats_my_emotion()**, the parameter should be called **$cash_money**, NOT $my_cash!

And so, Sherlock, it looks like we have solved the mystery of the **logical error**. We can now *remove* the extraneous echo and print_r statements and fix the problem, once and for all.

Type the following green code into CodeRunner:

```
<?
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number) { //first we took out the rogue semicolon...
       echo $the_sound."... ";
   }
}

function Mood_Chant($my_mood){
   if ($my_mood == "happy") {
      mantra("OM");
      $after_chant = "<br/>I feel serene now.";
   }
   else if ($my_mood == "sad") {
      mantra("okay");
      $after_chant = "<br/>I feel better now.";
   }
   else if ($my_mood == "angry") {
      mantra("mississippi");
      $after_chant = "<br/>Ahhh, much better.  I've calmed down now.";
   }
   else if ($my_outlook == "indifferent") {
      mantra("Wake up");
      $after_chant = "<br/>I'm awake now.";
   }
   else {
      mantra("Try harder");
      $after_chant = "<br/>I'll try harder now.";
   }
   return $after_chant;
}

function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $cash_money) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $cash_money ) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $cash_money && $cereal_prices['Fr
uit_Loops'] < $cash_money ){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
//then we added the delimiter here...
?>
<h3>The emotional roller-coaster of buying breakfast cereal</h3>
<?
$cereal_prices = array('Captain_Crunch' => 5, 'Fruit_Loops' => 3);
$my_cash = 4;
?>
I have $<? echo $my_cash; ?> in my pocket, and I want to buy some Captain Crunch
!<br/>
<ul>
<li>Captain Crunch costs $<? echo $cereal_prices['Captain_Crunch']; ?></li>
<li>Fruit Loops costs $<? echo $cereal_prices['Fruit_Loops']; ?></li>
</ul>
```

```
<?
//then we added the delimiter here...

$my_mood = whats_my_emotion($cereal_prices, $my_cash);
if (!($my_mood == "sad")) {
  $after_chant_mood = Mood_Chant($my_mood, $chant_number);
}
echo "<br/>".$after_chant_mood;

?>
```

Infinite Loops, Infinite Headaches

Preview this:

The emotional roller-coaster of buying breakfast cereal

I have $4 in my pocket, and I want to buy some Captain Crunch!

- Captain Crunch costs $5
- Fruit Loops costs $3

Fine! I'll get some Fruit Loops.mississippi... mississippi... mississippi... mississippi... mis
mississippi... mississippi... mississippi... mississippi... mississippi... mississippi... mississip
mississippi... mississippi... mississippi... mississippi... mississippi... mississippi... mississip
mississippi... mississippi... mississippi... mississippi... mississippi... mississippi... mississip
mississippi... mississippi... mississippi... mississippi... mississippi... mississippi... mississip

YIKES, MAKE IT STOP! This is one of the worst errors of all: **infinite loops**. It causes memory leaks in your computer, aching in your head, and it may very well have crashed your entire browser...we hope that wasn't the case.

This brings us to the Fifth Rule of Debugging: *Always end your loops!*

Take a look at our while loop:

```
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number) {
       echo $the_sound."... ";
   }
}
```

The good news is, in our case it's easy to see what went wrong. The while loop is set to end when **$chant** is **greater than $the_number**, which defaults to 10. But we never increased **$chant**. Let's fix it!

Type the following into CodeRunner:

```
<?
function mantra($the_sound,$the_number = 10) {
   $chant = 1;
   while ($chant <= $the_number) { //first we took out the rogue semicolon...
       echo $the_sound."... ";
       $chant++;
   }
}

function Mood_Chant($my_mood){
   if ($my_mood == "happy") {
      mantra("OM");
      $after_chant = "<br/>I feel serene now.";
   }
   else if ($my_mood == "sad") {
      mantra("okay");
      $after_chant = "<br/>I feel better now.";
   }
   else if ($my_mood == "angry") {
      mantra("mississippi");
      $after_chant = "<br/>Ahhh, much better.  I've calmed down now.";
   }
   else if ($my_outlook == "indifferent") {
      mantra("Wake up");
      $after_chant = "<br/>I'm awake now.";
   }
   else {
      mantra("Try harder");
      $after_chant = "<br/>I'll try harder now.";
   }
   return $after_chant;
}

function whats_my_emotion($cereal_prices, $cash_money) {
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $cash_money) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $cash_money) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $cash_money && $cereal_prices['Fr
uit_Loops'] < $cash_money){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
//then we added the delimiter here...
?>
<h3>The emotional roller-coaster of buying breakfast cereal</h3>
<?
$cereal_prices = array('Captain_Crunch' => 5, 'Fruit_Loops' => 3);
$my_cash = 4;
?>
I have $<? echo $my_cash; ?> in my pocket, and I want to buy some Captain Crunch
!<br/>
<ul>
<li>Captain Crunch costs $<? echo $cereal_prices['Captain_Crunch']; ?></li>
<li>Fruit Loops costs $<? echo $cereal_prices['Fruit_Loops']; ?></li>
```

```
</ul>

<?
//then we added the delimiter here...

$my_mood = whats_my_emotion($cereal_prices, $my_cash);
if (!($my_mood == "sad")) {
  $after_chant_mood = Mood_Chant($my_mood, $chant_number);
}
echo "<br/>".$after_chant_mood;

?>
```

Preview this:

The emotional roller-coaster of buying breakfast cereal

I have $4 in my pocket, and I want to buy some Captain Crunch!

- Captain Crunch costs $5
- Fruit Loops costs $3

Fine! I'll get some Fruit Loops.mississippi... mississippi... mississippi... mississippi... mississippi mississippi... mississippi... mississippi... mississippi... mississippi...

Ahhh, much better. I've calmed down now.

Ahh, much better. And although it was quite the ordeal, we're all the better for it. Pat yourself on the back and raise your glass to stress relief through Debugging!

Notes on Scalable Programming

Now that you know the Rules of Debugging, you're almost ready to put them to the test by building some large-scale projects. However, before you go on, it's important to stress some very important points to help reduce *your* stress.

Before you Code, Pseudocode

What's **pseudocode**? Just a little jot-down of your program logic in English (or whatever your native language may be). Like this:

Here's how we might pseudocode whats_my_emotion():

```
If I have enough money to buy both cereals,
   My mood is happy.
Otherwise, if I have enough money to buy Captain Crunch,
   My mood is indifferent.
Otherwise, if I can't buy Captain Crunch, but can buy Fruit Loops,
   My mood is angry.
Otherwise, I'm sad no matter what.
```

Pseudocode is a way for you to organize your thoughts and design your logic before you start coding your program. Think of it as a blueprint for your software development. Using pseudocode, you can take a look at the big picture and catch any flaws in your design--*before* they cause you a week's worth of debugging. Plus, you can refer back to it as you go to ensure that you're sticking to your original design.

Make your Program Readable

What if we had coded whats_my_emotion() like this?

```
function what_is_it($a, $b = false) {
   if ($a >= $b) {
      $c = "happy";
   }
   else if ($d[0] < $b) {
      $c = "indifferent";
   }
   else if ($d[0] > $b && $d[1] < $b) {
      $c = "angry";
   }
   else {
      $c = "sad";
   }
   return $c;
}
```

Sure, we know exactly what it means at the time we write it, but when we go back later, we might not have a clue what any of it means, rendering it essentially worthless. And by the way, if you write code like this, forget ever getting promoted - you won't find anyone who can decipher your code enough to take over your lower position. You'd be stuck with it, buddy.

So just be sure to use readable, intuitive variable, and function names all the time, every time. If you find yourself slipping into using vague names, just remember what we told you about promotion. That should snap you back into shape.

Comment Until You're Blue in the Face

By the same token, you can kick your program's readability up a notch by using comments whenever you can. Use them to help recall what you've done, or to indicate to other programmers what your functions do. That's why they're there after all.

In particular, it's imperative that you start off each program, and every function within it, with a synopsis of the functions it performs, parameters it takes, and what it returns.

Like this:

```
function whats_my_emotion($cereal_prices, $cash_money) {
   #whats_my_emotion returns an emotion of happy, indifferent, angry or sad base
d upon
   #two parameters, $cereal_prices -- an array of floats -- and float $cash_mone
y.
   $total = array_sum($cereal_prices);  //array_sum is a built-in PHP function
   if ($total < $cash_money) {
      $mood = "happy";
      echo "I'll buy both Captain Crunch and Fruit Loops!";
   }
   else if ($cereal_prices['Captain_Crunch'] < $cash_money) {
      $mood = "indifferent";
      echo "I'll buy Captain Crunch.";
   }
   else if ($cereal_prices['Captain_Crunch'] > $cash_money && $cereal_prices['Fr
uit_Loops'] < $cash_money){
      $mood = "angry";
      echo "Fine!  I'll get some Fruit Loops.";
   }
   else {
      $mood = "sad";
      echo "Oh well, I'm going home.";
   }
   return $mood;
}
```

This will become more and more important as you build more reusable code, and even libraries which can be

shared by others--either within your company, or within the **open-source community**.

Code in Bite-Size Chunks

You'll notice throughout these lessons that our chosen process of learning to build programs is extremely repetitious - we typed a bit of code, *Previewed* it, added a little more code, *Previewed* it, and so on.

If you program one small part of your code at a time, you'll be much less likely to be overwhelmed with bugs and logical errors when it comes time to test. This is where your pseudocode can help as well, by showing you where you can divide your large program into smaller, "bite-size" chunks to make it more manageable.

Debug as You Work

There's nothing worse than writing a HUGE amount of code, only to find it's a complete mess. As long as you *Preview* often, you'll catch bugs as you go along, which will make your life much easier in the long run.

Reuse Functions as Much as Possible

What if we had written several different functions instead of one **whats_my_emotion()**, or simply copied and pasted the same code throughout our program? Instead of fixing the code once, we would have had to deal with it over and over again.

The biggest argument for creating **functions** for anything and everything: if something goes wrong with the code, you only have to debug it **once**, then every time it's called, it works.

Always create a function for *any finite task*, even if you're not sure you'll use it more than once. You'll be surprised at how useful it will become as you continue programming.

Utilize Available Resources

As if we haven't been preaching it enough: we live in an age where information is *always* available. There are reference books, Safari accounts, and web sites like PHP.net. Use them. And if you can't find your answer, there are communities of millions of PHP programmers just like yourself you can consult. Don't be afraid to ask questions!

Can you believe how far you've come? So far you've learned all the basics of PHP you need to get going with some meaty web projects. And from now on, that's exactly what you're going to do.

Don't forget to **Save** your work and hand in your **assignments** from your syllabus. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Forms in PHP

As promised, you're about to put the material you've learned into robust, real-world applications. Until now, there has been only one thing missing from your skillset preventing this: **user input**.

PHP was created specifically to work with the internet - to make the web surfer's life easier by customizing his experience, and to make the programmer's life easier by making that customization convenient for her. But without a way to gather information from the web surfer, all the convenience and power of PHP is worthless. What good is customization if the user's needs aren't met?

We are able to gather user input through a little HTML tag called **<form>**. Since we'll be using HTML and PHP in tandem, be prepared to use **both HTML and PHP syntax**. Let's go!

Forms Review

Start with an HTML form. Make sure you're using **HTML** syntax, and TYPE the following:

CODE TO TYPE:

```
<body>
<h3>Contact ACME Corporation</h3>
<form method="POST" action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<input type="text" size="25" name="name" value="">
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<input type="text" size="25" name="email" value="">
</td>
</tr>
<tr>
<td align="right">
Type of Request:
</td>
<td align="left">
<select name="whoami">
<option value="" />Please choose...
<option value="newcustomer" />I am interested in becoming a customer.
<option value="customer" />I am a customer with a general question.
<option value="support" />I need technical help using the website.
<option value="billing" />I have a billing question.
</select>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="">
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
<textarea name="message" cols="50" rows="8">
</textarea>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth" />Word of Mouth<br/>
<input type="radio" name="found" value="search" />Online Search<br/>
<input type="radio" name="found" value="article" />Printed publication/article<br/>
<input type="radio" name="found" value="website" />Online link/article<br/>
<input type="radio" name="found" value="other" />Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked="checked">Please email me updates about y
```

```
our products.<br/>
<input type="checkbox" name="update2">Please email me updates about products from third
-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT" />
</td></tr>
</table>
</form>
</body>
```

You'll see this:

Contact ACME Corporation

Name:

Email:

Type of Request: Please choose...

Subject:

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

☑ Please email me updates about your products.

☐ Please email me updates about products from third-party partners.

SUBMIT

Look familiar? This is a simple contact form, where a user can inquire about the services on a website and give a little information about him/herself. Also noteworthy is that it contains all the major form types: **text**, **textarea**, **select**, **radio**, **checkbox**, and **submit**.

Each form element has a **name** attribute and a **value** attribute, except for **textarea**, which has an ending tag instead of a **value** attribute. Furthermore, **radio** buttons all have the same name to ensure only one is checked, while **checkbox**es have different names so that any of them can be checked. **select** tags contain their names and values within separate **option** tags, for that nice drop-down-menu effect.

Note You mean it *doesn't* look familiar? We're assuming this is review for you - if you're completely lost, you may want to take a look at our HTML and CSS course.

It's a nice-looking form, but if you want something done with that information, you're going to have to create a PHP script to process the input. Go ahead and **Save your form** -- you can name it **contact.html**.

Using Superglobals to Read Form Inputs

Now, let's switch CodeRunner to **PHP** and start a new file.

In PHP, type the following:

```
<?php

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$_POST['name']."<br/>";
echo "Email: ".$_POST['email']."<br/>";
echo "Type of Request: ".$_POST['whoami']."<br/>";
echo "Subject: ".$_POST['subject']."<br/>";
echo "Message: ".$_POST['message']."<br/>";
echo "How you heard about us: ".$_POST['found']."<br/>";
echo "Update you about our products: ".$_POST['update1']."<br/>";
echo "Update you about partners' products: ".$_POST['update2']."<br/>";

?>
```

Preview this:

Thank You!

Here is a copy of your request:

Name:
Email:
Type of Request:
Subject:
Message:
How you heard about us:
Update you about our products:
Update you about partners' products:

Well, that didn't do much good. And what's this **$_POST[]** array anyway??

But wait, there's more. ***Save*** this PHP file and call it **contact.php**. Now, switch back to **HTML** in CodeRunner, where you should still have your **contact.html** file ready.

Preview this, and fill in the form:

Contact ACME Corporation

Name: Trish

Email: trish@myemail.com

Type of Request: I need technical help using the website.

Subject: Please help!

Message: I can't get the darn thing to work!

How did you hear about us?

- ○ Word of Mouth
- ○ Online Search
- ○ Printed publication/article
- ◉ Online link/article
- ○ Other

☑ Please email me updates about your products.
☐ Please email me updates about products from third-party partners.

SUBMIT

Now, within your Preview window, click SUBMIT. What did you get?

Hopefully you got something like this:

Thank You!

Here is a copy of your request:

Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
Update you about our products: on
Update you about partners' products:

So why did it work this time? Here's where the magic of that **$_POST[]** array is revealed.

Take another look at the form tag in contact.html:

```
<form method="POST" action="contact.php">
```

If you remember, forms themselves can be submitted using several different methods - two of the most important methods are **GET** and **POST**. If you've ever programmed a web application in a different language - say Perl or C - you might also remember using complicated CGI libraries to extract form data from either the **query string** in the case of the GET method, or from the **environment variables** in the case of the POST method.

However, because PHP was created with the web in mind, this process has been greatly simplified, using special variables called **superglobals**.

Let's look at contact.php again:

```
<?php

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$_POST['name']."<br/>";
echo "Email: ".$_POST['email']."<br/>";
echo "Type of Request: ".$_POST['whoami']."<br/>";
echo "Subject: ".$_POST['subject']."<br/>";
echo "Message: ".$_POST['message']."<br/>";
echo "How you heard about us: ".$_POST['found']."<br/>";
echo "Update you about our products: ".$_POST['update1']."<br/>";
echo "Update you about partners' products: ".$_POST['update2']."<br/>";

?>
```

Did you notice something familiar about the key indices of **$_POST[]** -- **name**, **email**, **whoami**, etc.? You see, PHP does all the work for you here - it processes the form input and places all the values into the **superglobal array** **$_POST[]**, an associative array with the key indices corresponding to the form element names. This is done automatically, whenever a form is submitted using the **POST method**, and the array works in any scope - that's why it's called a **superglobal** variable.

Note By convention, we don't normally use the underscore at the beginning of variable names (as in **$_POST**). However, they are used in **superglobals** to prevent any clashing with your own variable names.

What do you do if you use the **GET method** in your form? Experiment with this and find out. If you need help, check out php.net.

Extracting Superglobals into Variables

As if the **superglobal** variables weren't easy enough, PHP goes even further to make reading form inputs easy for you.

In PHP, change contact.php with the following blue code:

```
<?php

extract($_POST, EXTR_PREFIX_SAME, "post");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";
echo "Update you about our products: ".$update1."<br/>";
echo "Update you about partners' products: ".$update2."<br/>";

?>
```

Save contact.php again, then go back to **contact.html** and Preview. What did you get?

Preview contact.html and submit the form like before:

Thank You!

Here is a copy of your request:

Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
Update you about our products: on
Update you about partners' products:

Wow - it worked even without the **$_POST[]** array! This is yet another simplification in the process that can be done through the built-in function **extract()**. In this case, the form elements passed through the **$_POST[]** array have been extracted into PHP variables, accessible by anything within the program. We indicated to **extract()** that we wanted the PHP variable names to correspond to the form element names by passing in the flag **EXTR_PREFIX_SAME** as a parameter. You can read more about **extract()** here: http://www.php.net/manual/en/function.extract.php.

Note In previous versions of PHP, a php config directive called **register globals** automatically created global variables from GET and POST form elements. However, many dangers arose in using register globals, and as a result, PHP has removed them from PHP 5 and newer versions.

Superglobals are brilliant innovations in web programming - all built into PHP to make your world an easier place to live. Not to mention *our* world - did you notice just how *short* this lesson is? Exactly.

Nesting Variable Names

Just one more cool feature before we move on...

In PHP, change contact.php with the following, in blue:

```
<?php

extract($_POST, EXTR_PREFIX_SAME, "post");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";


for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

?>
```

Preview contact.html and submit the form like before:

Thank You!

Here is a copy of your request:

Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:

Did you see what happened there? We were able to dynamically construct the name of our "update#" form elements through a **for loop**, and then *access the value* of that element through the variables we created with **extract()**. This was done by ***nesting variable names.***

Take another look:

```
for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";  << evaluates to "update1" or "update2"
   echo $$element_name;  << evaluates to $update1 or $update2, whose values are "on" or
 "off"
   echo "<br/>";
}
```

Nesting variable names is just like all the nesting we did in previous lessons - first **$element_name** is evaluated, and then that value is used to evaluate the nested **$($element_name)**. Name nesting can be done with ALL variables, however, it's especially useful when you create dynamic form names and then need to read them with the variables passed in through **superglobals** and **extract()**. It's definitely worth learning this handy trick.

We're just getting warmed up with forms. Don't forget to **Save** your work and hand in your **assignments** from your syllabus. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Utilizing Internet Tools

In the last lesson, we created a contact form for customers to communicate with the customer support department of a corporation. But it's not quite ready for prime time yet. So far, we have no way of knowing what kind of computer or browser the customer is using, no way to catch incomplete form entries, and no way to, well, send the message out.

It's time to fix this! Fire up CodeRunner and open the two files we were working on before: **contact.html** and **contact.php**.

Environment and Server Variables

Preview contact.html and submit the form like before:

Thank You!

Here is a copy of your request:

Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:

Have you ever received a customer support request like this? We have. It's more common than you may think, and it can leave you scratching your head—this customer can't get the *darn* website to work, yet leaves the details of the problem to your mind-reading skills.

Let's look into our crystal ball...

In PHP, change contact.php as shown in blue:

```
<?php

extract($_POST, EXTR_PREFIX_SAME, "post");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}


echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


?>
```

Switch to contact.html, Preview, and submit the form like before:

Thank You!

Here is a copy of your request:

Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/41
(KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

When it comes to customer support, just as important as the customer's request is knowing where the customer is coming from—perhaps geographically, but more importantly in the sense of what operating system (Windows, Mac) and browser (Safari, Internet Explorer, Firefox) they're using.

Luckily, the folks who worked on our very first web browsers way back in the day, already thought of this. They created something called **CGI *(Common Gateway Interface)* Environment Variables**, which tell us a lot about both the **client**—that's the customer's computer—as well as the **server** -- that's the computer where your PHP script resides (in our case, it's sitting in Champaign, Illinois).

Take another look at this code:

```
echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];
```

You guessed it—**$_SERVER[]** is another **superglobal array** in PHP. That underscore(_) at the beginning tends to give it away. The information that **$_SERVER[]** holds? **Environment variables** like **HTTP_USER_AGENT** and **HTTP_X_FORWARDED_FOR**. But what do they mean?

Now take another look at the output:

```
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/4
18 (KHTML, like Gecko)
Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74
```

Not so luckily, the folks who created the environment variables didn't make them easy to decipher. Here's a little translation for the two we're using:

- **HTTP_USER_AGENT** gives you information about the computer and web software your customer is using. Since you're testing your own form, it should be telling you what computer and web software *you're* using. The format is something like this: **operating system/version (more info) web library/version (more info) web browser/version (more info)**.

In our case, we got the information **Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/418 (KHTML, like Gecko) Safari/417.9.2**. Very cryptically, this tells us that we are on a Macintosh computer with a Mac OS X operating system, using the Safari web browser. Obviously, *your* result will most likely be different from this—you might be on a Windows XP computer, for instance, using Internet Explorer (MSIE). Here is a list of more browsers than you'll ever care to know, and their HTTP_USER_AGENT translations. Can you find yours?

- **HTTP_X_FORWARDED_FOR** gives you the **IP address** of either your customer's computer, or if your customer is using an Internet Service Provider like AOL, the IP address of one of its servers. What's an **IP (Internet Protocol) address**? Every computer on the internet has one -- a unique identifier, chosen within the Internet Protocol Standard. It's useful to know, because it can indicate the customer's country of origin, through any Whois tool. More importantly, if it has been determined that a particular customer is a fraud, there are ways to block the IP address from ever getting to your site—something you will learn in a later course.

There are lots of useful **environment variables**. Here is a very useful list to reference.

Using HTTP Headers

Another important issue in customer support—and really any interface that requires form input—is ensuring that all fields are properly filled in. How can you help a customer if he doesn't include his email address or contact info? But of course he'll include it, right? You'd be surprised.

In PHP, change contact.php with the following blue code:

```
<?php

#We used the superglobal $_POST here
if (!($_POST['name'] && $_POST['email'] && $_POST['whoami']
      && $_POST['subject'] && $_POST['message'])) {
   echo "Please make sure you've filled in all required information.";
   exit();
}

extract($_POST, EXTR_PREFIX_SAME, "post");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];

?>
```

Note Notice the new built-in PHP function we used here: **exit()**. This essentially stops the program in its tracks. How's that for lack of commitment?

Switch to contact.html, Preview, and submit the form like before—only this time, try leaving something blank:

Please make sure you've filled in all required information.

So essentially when someone leaves something blank, we're letting them know about it. But now the customer has to go back to the form and find out what's wrong. What if we could take them back to the form automatically? Let's give it a shot:

In PHP, change contact.php with the following, in blue:

```
<?php

#We used the superglobal $_POST here
if (!($_POST['name'] && $_POST['email'] && $_POST['whoami']
      && $_POST['subject'] && $_POST['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $url = "http://".$_SERVER['HTTP_HOST']."/contact.html";
   header("Location: ".$url);
   exit();
}

extract($_POST, EXTR_PREFIX_SAME, "post");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


?>
```

Switch to contact.html, Preview your form, and submit it, leaving something blank. You should get this:

Contact ACME Corporation

Name:

Email:

Type of Request: Please choose...

Subject:

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

- [x] Please email me updates about your products.
- [ ] Please email me updates about products from third-party partners.

SUBMIT

Whoa! What just happened? If you left something blank on your form and submitted it, you just got the same form back, blank again.

Let's take another look at this code:

```
$url = "http://".$_SERVER['HTTP_HOST']."/contact.html";
header("Location: ".$url);
exit();
```

You may already know that all HTML-based web pages use the **HyperText Transfer Protocol (HTTP)** to render properly in your web browser—that's why you always see **http://** at the beginning of every web address. But what you may NOT know is that before any HTML is rendered on your web browser, a series of invisible **headers** are passed so that your browser knows exactly what to do with the code. Most of these headers are pretty obscure, but a few are extremely useful. Click here for a reference.

Of course, since PHP *embeds* HTML within its code, it can also manipulate **HTTP headers** through the built-in function **header()**. In this case, we were able to set the header **"Location: "** with the URL of the contact form **contact.html** we created. As a result of sending that header, the browser *redirected* the user back to the form.

Note Any headers that are sent using **header()** must come BEFORE any PHP or HTML output. Otherwise, the browser will get confused, and next thing you know, you're debugging.

You'll also notice we used another **environment variable**, called **HTTP_HOST**. This variable returns the **domain name** of the web address where your **contact.php** script resides.

In our case, our domain name, or HTTP_HOST, is **josh.onza.net**. It is a live web site, on the internet for everyone to see: http://josh.onza.net/. Pretty lame web site, huh? Keep this in mind when you create your website using your own domain name—you could have a lame web site like us, or you could have a professional online portfolio to show to all your friends, colleagues, and potential employers when you apply for your first LAMP-based programming job.

Manipulating Query Strings

But we digress. And in the meantime, simply redirecting our poor customer to a blank form is a horrible way to treat someone who's already frustrated with the website. There has to be a more user-friendly way to ask the customer to fix a form field before we submit it.

The problem is, since **contact.html** is a static HTML page, we can't dynamically add anything to it—that's why it's blank. And simply giving the error message "Please fix this" to the customer, like we did before, isn't user-friendly either. What we need is a way to show the customer, nicely, exactly what he needs to fix on the form, without losing any of the answers he's already filled in.

We can do this by *converting* the HTML form into a PHP script of its own. What you need to do is ***Save* contact.html in PHP syntax, but sure to call it "contact_form.php"**. Or if you'd rather, just copy and paste the HTML code into a new PHP file.

Be sure you're in PHP, and add the following blue code to contact_form.php:

```
<body>
<h3>Contact ACME Corporation</h3>
<form method="GET" action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>" />
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<input type="text" size="25" name="email" value="<? echo $_GET['email']; ?>" />
</td>
</tr>
<tr>
<td align="right">
Type of Request:
</td>
<td align="left">
<select name="whoami">
<option value="" />Please choose...
<option value="newcustomer"<?
if ($_GET['whoami'] == "newcustomer") {
   echo " selected";
}
?> />I am interested in becoming a customer.
<option value="customer"<?
if ($_GET['whoami'] == "customer") {
   echo " selected";
}
?> />I am a customer with a general question.
<option value="support"<?
if ($_GET['whoami'] == "support") {
   echo " selected";
}
?> />I need technical help using the website.
<option value="billing"<?
if ($_GET['whoami'] == "billing") {
   echo " selected";
}
?> />I have a billing question.
</select>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="<? echo $_GET['subject']; ?
>" />
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
<textarea name="message" cols="50" rows="8">
<? echo $_GET['message']; ?>
```

```
</textarea>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth" />Word of Mouth<br/>
<input type="radio" name="found" value="search" />Online Search<br/>
<input type="radio" name="found" value="article" />Printed publication/article<br/>
<input type="radio" name="found" value="website" />Online link/article<br/>
<input type="radio" name="found" value="other" />Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked="checked" />Please email me updates about
 your products.<br/>
<input type="checkbox" name="update2" />Please email me updates about products from thi
rd-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT" />
</td></tr>
</table>
</form>
</body>
```

Now, Preview contact_form.php:

Contact ACME Corporation

Name:

Email:

Type of Request: Please choose...

Subject:

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

- [x] Please email me updates about your products.
- [ ] Please email me updates about products from third-party partners.

SUBMIT

You'll notice here that we've switched the **method** attribute in the HTML form tag from **POST** to **GET**, and we've introduced some PHP echo statements using the **$_GET[] superglobal**. But so far, no changes have taken place—we still get the same blank form with **contact_form.php** as we did with **contact.html**.

Note

The **htmlspecialchars()** function can be used when obtaining the values for the input tags. For example:

```
<input type="text" size="25" name="email" value="<? echo
htmlspecialchars($_GET['email'],ENT_QUOTES, 'UTF-8'); ?>" /> </td>
```

This function converts some predefined characters to HTML entities and will help to protect your code against cross site scripting. A detailed discussion of web application security in beyond the scope of this course, but please check out the following links for additional information:

link
link.

Be sure to ***Save* contact_form.php**, since the **Location:** header will redirect you back to the *saved* version of **contact_form.php**, NOT the Preview version.

Switch to contact.php, and make the following changes in blue:

```
<?php


#We used the superglobal $_GET here
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";


   $query_string = $_SERVER['QUERY_STRING'];
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string;

   header("Location: ".$url);
   exit();

}

extract($_GET, EXTR_PREFIX_SAME, "get");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


?>
```

Remember to save contact.php, then switch back to contact_form.php and Preview.

When you submit the form, be sure that you leave one field blank to see what happens:

Contact ACME Corporation

Name: Trish

Email: trish@myemail.com

Type of Request: I need technical help using the website.

Subject: Woops, I left the message field blank!

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

- [x] Please email me updates about your products.
- [ ] Please email me updates about products from third-party partners.

SUBMIT

Now here's some real progress. When you submit the form with a field or two blank, the form still comes back—but this time, all the fields at the top have been filled in. This is much better, because now the user doesn't have to redo everything.

Let's take another look at the code we used in contact.php:

```
#We used the superglobal $_GET here
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";


   $query_string = $_SERVER['QUERY_STRING']
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string;
   header("Location: ".$url);
   exit();

}

extract($_GET, EXTR_PREFIX_SAME, "get");
```

We switched our form to have **method=GET** so that our data will come through to our script from the **query string**. The **query string** consists of all the encoded data you see after the question mark (**?**) in your URL when you submit

the form:

http://josh.onza.net/contact_form.php?name=Trish&email=trish%40myemail.com&whoami=support&subject=Woops%2C+I+left+the+message+field+blank%21

And, since we have the handy **environment variable QUERY_STRING**, we can simply use the **$_SERVER[]** superglobal to grab it and send it back to **contact_form.php**.

And if you look again at contact_form.php:

```
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>" />
```

You'll see that we were able to harness the **query string** yet again—through the superglobal **$_GET[]**—to fill in the input tags with the customer's original data.

Customizing specific error messages

Now it's time to use our newly-formed script **contact_form.php** to tell the customer exactly what needs to be done. To do this, however, we first need to *manipulate* the **query string** a bit:

In PHP, switch to contact.php, and make the following changes, in blue:

```
<?php

#We used the superglobal $_GET here
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixe
d
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&e
rror=1";
   header("Location: ".$url);
   exit();

}

extract($_GET, EXTR_PREFIX_SAME, "get");

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_
X_FORWARDED_FOR'];

?>
```

Make sure you Save contact.php, and switch to **contact_form.php**:

In PHP, add the following to contact_form.php, in blue:

```
<?php

if ($_GET['error'] == "1") {
   $error_code = 1;  //this means that there's been an error and we need to noti
fy the customer
} else {
   $error_code = 0;
}

?>

<body>
<h3>Contact ACME Corporation</h3>
<?
if ($error_code) {
   echo "<div style='color:red'>Please help us with the following:</div>";
}
?>
<form method="GET" action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>" />
<?
if ($error_code && !($_GET['name'])) {
   echo "<b>Please include your name.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<input type="text" size="25" name="email" value="<? echo $_GET['email']; ?>" />
<?
if ($error_code && !($_GET['email'])) {
   echo "<b>Please include your email address.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Type of Request:
</td>
<td align="left">
<select name="whoami">
<option value="" />Please choose...
<option value="newcustomer"<?
if ($_GET['whoami'] == "newcustomer") {
   echo " selected";
}
?> />I am interested in becoming a customer.
<option value="customer"<?
if ($_GET['whoami'] == "customer") {
   echo " selected";
}
?> />I am a customer with a general question.
<option value="support"<?
if ($_GET['whoami'] == "support") {
   echo " selected";
}
```

```
?> />I need technical help using the website.
<option value="billing"<?
if ($_GET['whoami'] == "billing") {
   echo " selected";
}
?> />I have a billing question.
</select>
<?
if ($error_code && !($_GET['whoami'])) {
   echo "<b>Please choose a request type.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="<? echo $_GET['subje
ct']; ?>" />
<?
if ($error_code && !($_GET['subject'])) {
   echo "<b>Please add a subject for your request.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
<textarea name="message" cols="50" rows="8">
<? echo $_GET['message']; ?>
</textarea>
<?
if ($error_code && !($_GET['message'])) {
   echo "<b>Please fill in a message for us.</b>";
}
?>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth" />Word of Mouth<br/>
<input type="radio" name="found" value="search" />Online Search<br/>
<input type="radio" name="found" value="article" />Printed publication/article<b
r/>
<input type="radio" name="found" value="website" />Online link/article<br/>
<input type="radio" name="found" value="other" />Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked="checked" />Please email me update
s about your products.<br/>
<input type="checkbox" name="update2" />Please email me updates about products f
rom third-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT" />
</td></tr>
```

```
</table>
</form>
</body>
```

Again, be sure to Save contact_form.php, and then Preview, leaving one field blank. What did you get?

We get something like this:

Contact ACME Corporation

Please help us with the following:

Name: [ ] **Please include your name.**

Email: trish@myemail.com

Type of Request: I need technical help using the website.

Subject: Woops, I left the message field blank!

Message: **Please fill i**
us.

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

[x] Please email me updates about your products.

MUCH better. Now the customer knows exactly what's wrong, he can fix it, and submit the support request easily.

Sending Emails

Finally, we can do what we wanted to do all along: send the support request via email. Never one to let us down, PHP has just the function for us: **mail()**. Let's try it:

In PHP, switch to contact.php, and make the following changes, in blue:

```
<?php


#We used the superglobal $_GET here
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

extract($_GET, EXTR_PREFIX_SAME, "get");

#construct email message
$email_message = "Name: ".$name."
 Email: ".$email."
 Type of Request: ".$whoami."
 Subject: ".$subject."
 Message: ".$message."
 How you heard about us: ".$found."
 User Agent: ".$_SERVER['HTTP_USER_AGENT']."
 IP Address: ".$_SERVER['REMOTE_ADDR'];

#construct the email headers
$to = "support@example.com";  //for testing purposes, this should be YOUR email address
.
//We will send the emails from our own server
$from = "anything@yourlogin.oreillystudent.com";

$email_subject = $_GET['subject'];

#now mail
mail($to, $email_subject, $email_message, "From: ".$from);

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";

echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


?>
```

***Save* contact.php**, switch to contact_form.php, Preview, and submit the form.

If you filled in all the required fields, you should get something like before:

Thank You!

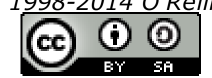
Here is a copy of your request:

Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/41
(KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

However, *this* time, if you included your own email in the **$to** variable, you should have a brand new customer support message in your email inbox.

Don't forget to **Save** your work and hand in the **assignments** from your syllabus. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Date and Time

You'll find that Date and Time play a huge part in programming - they are useful for timestamps, logs, and are needed in just about every database entry you'll create. And although they're somewhat tricky to harness, PHP has done well in simplifying the process.

Open the two files we were working on before: **contact_form.php** and **contact.php**.

Date and Time Standards

Switch to contact.php, and make the following changes, in green:

```
<?php


#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

extract($_GET, EXTR_PREFIX_SAME, "get");

#construct email message
$email_message = "Name: ".$name."
 Email: ".$email."
 Type of Request: ".$whoami."
 Subject: ".$subject."
 Message: ".$message."
 How you heard about us: ".$found."
 User Agent: ".$_SERVER['HTTP_USER_AGENT']."
 IP Address: ".$_SERVER['HTTP_X_FORWARDED_FOR'];

#construct the email headers
$to = "support@example.com";  //for testing purposes, this should be YOUR email address
.
//We will send the emails from our own server
$from = "anything@yourlogin.oreillystudent.com";

$email_subject = "CONTACT #".time().": ".$_GET['subject'];

#now mail
mail($to, $email_subject, $email_message, "From: ".$from);

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


?>
```

Note Notice we're breaking one of our own cardinal rules here - doubling up on code that could be taken care of with one function. Feel free to punish us within your own code.

***Save* contact.php**, switch to contact_form.php, Preview, and submit the form.

If you filled in all the required fields, you should get something like this:

Thank You!

Here is a copy of your request:

CONTACT #1148955473:
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/4
(KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

Although we're using it as a **timestamp** here, the number we got actually measures how many seconds have passed since **Unix Epoch** -- that's a fancy name for January 1st, 1970, at midnight (00:00:00) GMT. Why is that time the **Unix Epoch**? No good reason really, except that some early computer scientists agreed on it a long time ago as a **date and time standard**.

Sounds nerdy, but it's really a good thing - it enables us to harness date and time, not only in PHP, but also in mySQL and lots of other technology languages. For instance, you'll be using PHP functions to process SQL timestamps in later courses.

Date and Time Functions

Obviously, **date and time standards** weren't created for us to use merely as a unique identification number - although that's handy. What else can they do for us? Enter the built-in PHP functions.

Switch to contact.php, and make the following changes, in green:

```
<?php


#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

extract($_GET, EXTR_PREFIX_SAME, "get");

#construct email message
$email_message = "Message Date: ".date("F d, Y h:i a")."
 Name: ".$name."
 Email: ".$email."
 Type of Request: ".$whoami."
 Subject: ".$subject."
 Message: ".$message."
 How you heard about us: ".$found."
 User Agent: ".$_SERVER['HTTP_USER_AGENT']."
 IP Address: ".$_SERVER['HTTP_X_FORWARDED_FOR'];

#construct the email headers
$to = "support@example.com";  //for testing purposes, this should be YOUR email address
.
//We will send the emails from our own server
$from = "anything@yourlogin.oreillystudent.com";

$email_subject = "CONTACT #".time().": ".$_GET['subject'];

#now mail
mail($to, $email_subject, $email_message, "From: ".$from);

echo "<h3>Thank you!</h3>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];
```

```
?>
```

Again, if you ***Save contact.php***, then Preview contact_form.php, you might get something like this:

Thank You!

Here is a copy of your request:

CONTACT #1148955473:
Message Date: May 29, 2006 10:25 pm
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/41
(KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

This time, instead of a cryptic timestamp, the date of the message has been nicely formatted for us through the **date()** function.

Let's take another look:

```
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
```

The **parameter** for the **date()** function is a special coded **format** that PHP replaces with the proper time/date data. For instance, **"F"** is replaced with the name of the month - in our case, **May**, and **"a"** is replaced with **am or pm**, depending on the time - in our case, **pm**. Here is the php.net reference for time/date formats.

format character	Description	Example returned values
Day	---	---
d	Day of the month, 2 digits with leading zeros	*01* to *31*
D	A textual representation of a day, three letters	*Mon* through *Sun*
j	Day of the month without leading zeros	*1* to *31*
l (lowercase 'L')	A full textual representation of the day of the week	*Sunday* through *Saturday*
N	ISO-8601 numeric representation of the day of the week (added in PHP 5.1.0)	*1* (for Monday) through *7* (for Sunday)
S	English ordinal suffix for the day of the month, 2 characters	*st*, *nd*, *rd* or *th*. Works well with *j*
w	Numeric representation of the day of the week	*0* (for Sunday) through *6* (for Saturday)
z	The day of the year (starting from 0)	*0* through *365*
Week	---	---

W	ISO-8601 week number of year, weeks starting on Monday (added in PHP 4.1.0)	Example: *42* (the 42nd week in the year)
Month	---	---
F	A full textual representation of a month, such as January or March	*January* through *December*
m	Numeric representation of a month, with leading zeros	*01* through *12*
M	A short textual representation of a month, three letters	*Jan* through *Dec*
n	Numeric representation of a month, without leading zeros	*1* through *12*
t	Number of days in the given month	*28* through *31*
Year	---	---
L	Whether it's a leap year	*1* if it is a leap year, *0* otherwise.
o	ISO-8601 year number. This has the same value as *Y*, except that if the ISO week number (*W*) belongs to the previous or next year, that year is used instead. (added in PHP 5.1.0)	Examples: *1999* or *2003*
Y	A full numeric representation of a year, 4 digits	Examples: *1999* or *2003*
y	A two digit representation of a year	Examples: *99* or *03*
Time	---	---
a	Lowercase Ante meridiem and Post meridiem	*am* or *pm*
A	Uppercase Ante meridiem and Post meridiem	*AM* or *PM*
B	Swatch Internet time	*000* through *999*
g	12-hour format of an hour without leading zeros	*1* through *12*
G	24-hour format of an hour without leading zeros	*0* through *23*
h	12-hour format of an hour with leading zeros	*01* through *12*
H	24-hour format of an hour with leading zeros	*00* through *23*
i	Minutes with leading zeros	*00* to *59*
s	Seconds, with leading zeros	*00* through *59*
Timezone	---	---
e	Timezone identifier (added in PHP 5.1.0)	Examples: *UTC*, *GMT*, *Atlantic/Azores*
I (capital i)	Whether or not the date is in daylights savings time	*1* if Daylight Savings Time, *0* otherwise.
O	Difference to Greenwich time (GMT) in hours	Example: *+0200*
P	Difference to Greenwich time (GMT) with colon between hours and minutes (added in PHP 5.1.3)	Example: *+02:00*
T	Timezone setting of this machine	Examples: *EST*, *MDT* ...
Z	Timezone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UTC is always positive.	*-43200* through *43200*
Full Date/Time	---	---
c	ISO 8601 date (added in PHP 5)	2004-02-12T15:19:21+00:00
r	RFC 2822 formatted date	Example: *Thu, 21 Dec 2000 16:01:07 +0200*

U	Seconds since the Unix Epoch (January 1 1970 00:00:00 GMT)	

Constructing Dates and Times

Now, suppose Acme, Inc. had a customer service policy claiming "We'll get back to you in 48 hours." You'll want to use the date of the message to give the customer support representative an idea of the deadline she has.

Make sure you have contact.php, and make the following changes, in green:

```
<?php

#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixe
d
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&e
rror=1";
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

extract($_GET, EXTR_PREFIX_SAME, "get");

#construct email message

#we want a deadline 2 days after the message date.
$deadline_array = getdate();
$deadline_day = $deadline_array['mday'] + 2;
$deadline_str = $deadline_array['month']." ".$deadline_day." ".$deadline_array['
year'];

$email_message = "Message Date: ".date("F d, Y h:i a")."
 Please reply by: ".$deadline_str."
 Name: ".$name."
 Email: ".$email."
 Type of Request: ".$whoami."
 Subject: ".$subject."
 Message: ".$message."
 How you heard about us: ".$found."
 User Agent: ".$_SERVER['HTTP_USER_AGENT']."
 IP Address: ".$_SERVER['HTTP_X_FORWARDED_FOR'];

#construct the email headers
$to = "support@example.com";  //for testing purposes, this should be YOUR email
address.
//We will send the emails from our own server
$from = "anything@yourlogin.oreillystudent.com";

$email_subject = "CONTACT #".time().": ".$_GET['subject'];

#now mail
mail($to, $email_subject, $email_message, "From: ".$from);

echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br/>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
```

```
    $element_name = "update".$i;
    echo $element_name.": ";
    echo $$element_name;
    echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_
X_FORWARDED_FOR'];


?>
```

Save contact.php, switch to contact_form.php and Preview:

Thank You!

We'll get back to you by May 31 2006.
Here is a copy of your request:

CONTACT #1148955473:
Message Date: May 29, 2006 10:25 pm
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWe
(KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

Take another look:

```
#we want a deadline 2 days after the message date.
$deadline_array = getdate();
$deadline_day = $deadline_array['mday'] + 2;
$deadline_str = $deadline_array['month']." ".$deadline_day." ".$deadline_array['
year'];
```

Here, the function **getdate()**, like **time()**, gets a stamp of the current time. However, instead of just an integer, **getdate()** extracts the data and outputs an **associative array** that looks a bit like this:

Here's what a getdate() output array might look like:

```
Array
(
    [seconds] => 40
    [minutes] => 58
    [hours]   => 21
    [mday]    => 29
    [wday]    => 1
    [mon]     => 5
    [year]    => 2006
    [yday]    => 160
    [weekday] => Monday
    [month]   => May
    [0]       => 1055901520
)
```

This array makes it easy to construct a new date relative to the current date - all we have to do is add 2 to the **'mday'** array value, and suddenly we have a deadline for the customer support representative. Fast service means happy customers.

But wait a minute - what if today was, say, the 31st of May? Just adding 2 to that will give you an invalid date. We could do a series of **if** statements to fix this, but that's a lot of unwieldy code. Luckily, PHP has yet another handy function to help us.

In PHP, try adding the following green code to contact.php:

```
<?php

#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixe
d
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&e
rror=1";
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

extract($_GET, EXTR_PREFIX_SAME, "get");

#construct email message
#we want a deadline 2 days after the message date.
$deadline_array = getdate();
$deadline_day = $deadline_array['mday'] + 2;

$deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$de
adline_array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
$deadline_str = date("F d, Y", $deadline_stamp);

$email_message = "Message Date: ".date("F d, Y h:i a")."
 Please reply by: ".$deadline_str."
 Name: ".$name."
 Email: ".$email."
 Type of Request: ".$whoami."
 Subject: ".$subject."
 Message: ".$message."
 How you heard about us: ".$found."
 User Agent: ".$_SERVER['HTTP_USER_AGENT']."
 IP Address: ".$_SERVER['HTTP_X_FORWARDED_FOR'];

#construct the email headers
$to = "support@example.com";  //for testing purposes, this should be YOUR email
address.
//We will send the emails from our own server
$from = "anything@yourlogin.oreillystudent.com";

$email_subject = "CONTACT #".time().": ".$_GET['subject'];

#now mail
mail($to, $email_subject, $email_message, "From: ".$from);

echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br/>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";
```

```
for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_
X_FORWARDED_FOR'];


?>
```

Save contact.php, switch to contact_form.php and Preview:

Thank You!

We'll get back to you by June 2 2006.
Here is a copy of your request:

CONTACT #1148962509:
Message Date: May 31, 2006 12:20 pm
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can't get the darn thing to work!
How you heard about us: website
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWe
(KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

Take one more look:

```
$deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$de
adline_array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
$deadline_str = date("F d, Y", $deadline_stamp);
```

The function **mktime()** fixes all those pesky date problems. It takes in the parameter data of the date you want to format, and creates the original **timestamp**, which we then plug into **date()** to format properly. Problem solved!

Don't forget to **Save** your work and hand in your **assignments** from your syllabus. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.*

Using Files

So far, we've created a simple corporate contact form that accounts for user error and friendly reminders, successful output, and sending of the proper message. Pretty robust, but at the same time, you have to admit, it's pretty ugly. And it's not the most professional-looking interface on the web either.

In a perfect world, we'd have lots of time to keep tweaking the PHP script to make every page look just so. But in most situations, you won't have the luxury of extra time, and you probably won't even be allowed to dictate how the page looks. Not when there's a graphic designer down the hall. Just the same, you don't want the graphic designer down the hall messing with your PHP scripts either. Here's where **file templates** come in real handy.

Including and Requiring Files

Fire up CodeRunner and open up the two files we were working on before: **contact_form.php** and **contact.php**. After you do this, switch CodeRunner to **HTML syntax**. For just a moment, we're going to pretend that we are the graphic designers down the hall.

In HTML, type the following, in blue:

```
<html>
<head>
<title>Acme, Inc.</title>
<link rel="stylesheet" href="http://students.oreillyschool.com/resource/php_lesson.css"
 type="text/css" />
</head>
<body>
<div class="topbar">
ACME, INC.
</div>
<table>
<tr><td class="sidebar" valign="top">
links go here
</td><td class="content">
Content goes here

</td></tr></table>
<div class="bottombar">
</div>
</body>
</html>
```

Preview this:

links go here

Content goes here

What we have here is a basic "C-Clamp" design template for a corporate web page: logo on top, links on the side, something on the bottom to wrap the content nicely, and a CSS file to add a little style (here we've provided one for you). This will make our contact form look slightly better than it did before.

But how do you most easily place our content within this C-Clamp? You could simply *embed* the HTML into the PHP script itself, but this creates a big problem - if the graphic designer decides to make a change, you're stuck making that

same change in every PHP script you've written. And if you work for a large corporation, this could mean dozens, even hundreds of files.

It would be great is if you could *reuse* the code, like when you create PHP functions.

In HTML, remove the second half of our C-Clamp:

```
<html>
<head>
<title>Acme, Inc.</title>
<link rel="stylesheet" href="http://students.oreillyschool.com/resource/php_lesson.css"
 type="text/css" />
</head>
<body>
<div class="topbar">
ACME, INC.
</div>
<table>
<tr><td class="sidebar" valign=top>
links go here
</td><td class="content">
```

Now, ***Save*** this file and name it **template_top.inc**.

Note Why use **.inc**? Just for clarity - this isn't a complete HTML file, so no need to name it with a .html suffix.

In HTML, create a NEW file, containing the second half of our C-Clamp:

```
</td></tr></table>
<div class="bottombar">
</div>
</body>
</html>
```

Save this file and name it **template_bottom.inc**.

Add the following to contact_form.php, in green:

```
<?php

   require($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


if ($_GET['error'] == "1") {
   $error_code = 1;  //this means that there's been an error and we need to notify the
customer
}

?>
<body>
<h3>Contact ACME Corporation</h3>
<?
if ($error_code) {
   echo "<font color=red>Please help us with the following:</font>";
}
?>
<form method=GET action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>">
<?
if ($error_code && !($_GET['name'])) {
   echo "<b>Please include your name.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<input type="text" size="25" name="email" value="<? echo $_GET['email']; ?>">
<?
if ($error_code && !($_GET['email'])) {
   echo "<b>Please include your email address.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Type of Request:
</td>
<td align="left">
<select name="whoami">
<option value="">Please choose...
<option value="newcustomer"<?
if ($_GET['whoami'] == "newcustomer") {
   echo " selected";
}
?>>I am interested in becoming a customer.
<option value="customer"<?
if ($_GET['whoami'] == "customer") {
   echo " selected";
}
?>>I am a customer with a general question.
<option value="support"<?
if ($_GET['whoami'] == "support") {
   echo " selected";
```

```
}
?>>I need technical help using the website.
<option value="billing"<?
if ($_GET['whoami'] == "billing") {
   echo " selected";
}
?>>I have a billing question.
</select>
<?
if ($error_code && !($_GET['whoami'])) {
   echo "<b>Please choose a request type.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="<? echo $_GET['subject']; ?
>">
<?
if ($error_code && !($_GET['subject'])) {
   echo "<b>Please add a subject for your request.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
<textarea name="message" cols=50 rows=8>
<? echo $_GET['message']; ?>
</textarea>
<?
if ($error_code && !($_GET['message'])) {
   echo "<b>Please fill in a message for us.</b>";
}
?>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth">Word of Mouth<br>
<input type="radio" name="found" value="search">Online Search<br>
<input type="radio" name="found" value="article">Printed publication/article<br>
<input type="radio" name="found" value="website">Online link/article<br>
<input type="radio" name="found" value="other">Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked>Please email me updates about your produc
ts.<br>
<input type="checkbox" name="update2">Please email me updates about products from third
-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT">
</td></tr>
```

```
</table>
</form>


<?
   require($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");
?>

</body>
```

Be sure and ***Save*** **contact_form.php**.

Now PREVEIW:

ACME, INC.

links go here

Contact ACME Corporation

Name:

Email:

Type of Request: Please choose...

Subject:

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

☑ Please email me updates about your products.

☐ Please email me updates about products from third-party partners.

SUBMIT

Even with the extreme simplicity of our C-Clamp template, this looks much better than it did before.

Take another look at the code:

```
   require($_SERVER['DOCUMENT_ROOT']."/template_top.inc");
```

The PHP built-in function **require()** takes a filename as its **parameter**, and imports all the data from that filename into that exact place within the PHP code. It's as if you had written the code right in.

We can do this with **contact.php** as well.

Switch to contact.php and add the following green code:

```
<?php

#Remember, if you place any output before a header() call, you'll get an error.
#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}
extract($_GET, EXTR_PREFIX_SAME, "get");
#construct email message
#we want a deadline 2 days after the message date.
$deadline_array = getdate();
$deadline_day = $deadline_array['mday'] + 2;

$deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$deadline_
array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
$deadline_str = date("F d, Y", $deadline_stamp);

$email_message = "Message Date: ".date("F d, Y h:i a")."<br>
 Please reply by: ".$deadline_str."<br>
 Name: ".$name."<br>
 Email: ".$email."<br>
 Type of Request: ".$whoami."<br>
 Subject: ".$subject."<br>
 Message: ".$message."<br>
 How you heard about us: ".$found."<br>
 User Agent: ".$_SERVER['HTTP_USER_AGENT']."<br>
 IP Address: ".$_SERVER['HTTP_X_FORWARDED_FOR'];

#construct the email headers
$to = "support@example.com";  //for testing purposes, this should be YOUR email address
.
//We will send the emails from our own server
$from = "anything@yourlogin.oreillystudent.com";

$email_subject = "CONTACT #".time().": ".$_GET['subject'];

$headers  = "From: " . $from . "\r\n";
$headers .= 'MIME-Version: 1.0' . "\n";  //these headers will allow our HTML tags to be
 displayed in the email
$headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";

#now mail
mail($to, $email_subject, $email_message, $headers);


   include($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br>";
echo "Here is a copy of your request:<br><br>";
echo "CONTACT #".time().":<br>";
echo "Message Date: ".date("F d, Y h:i a")."<br>";
```

```
echo "Name: ".$name."<br>";
echo "Email: ".$email."<br>";
echo "Type of Request: ".$whoami."<br>";
echo "Subject: ".$subject."<br>";
echo "Message: ".$message."<br>";
echo "How you heard about us: ".$found."<br>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWAR
DED_FOR'];


   include($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");


?>
```

***Save* contact.php**. Now when you view **contact_form.php** and submit the form, you should see something like this:

ACME, INC.

links go here

Thank you!

We'll get back to you by June 07, 2006.
Here is a copy of your request:

CONTACT #1149489921:
Message Date: June 05, 2006 01:45 pm
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can/'t get the darn thing to work!
How you heard about us: other
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; er
AppleWebKit/418 (KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

Note This time, instead of **require()** we used **include()**. What's the difference? If for some reason the URL doesn't exist, **require()** will give you a PHP error, whereas **include()** will just skip that URL.

Reading and Writing Files

Now that we've got the web interface looking better, let's work on the email. In this case, there's no clear-cut beginning and ending template, rather, the data is peppered throughout the email. So if we want to use a template with this, we'll have to find a way to insert the data into the template, instead of the other way around.

First, let's see how we want the template to look. Switch to **HTML**, and create a **text-only file** that looks something like below.

Make sure you're in HTML, and type the following into a new file:

```
You have just received a customer email.  Please respond to this email by #DEADLINE#.
Details are below:

<table>
<tr><td width="100" align="right">Message Type: </td><td>#WHOAMI#</td></tr>
<tr><td width="100" align="right">Message Date: </td><td>#DATE#</td></tr>
<tr><td width="100" align="right">Name: </td><td>#NAME#</td></tr>
<tr><td width="100" align="right">Email: </td><td>#EMAIL#</td></tr>
<tr><td width="100" align="right">IP Address: </td><td>#IP#</td></tr>
<tr><td width="100" align="right">Platform: </td><td>#AGENT#</td></tr>
</table>

<b>Subject: #SUBJECT#
<br>
#MESSAGE#</b>
<br><br>
This customer found us through #FOUND#.<br>
#CONTACT#
```

Save this text file, and call it **email_template.txt**. Now let's go back to **contact.php**.

Switch to contact.php and make the following changes, in green:

```
<?php

function mail_message($data_array, $template_file, $deadline_str) {

   #get template contents, and replace variables with data
   $email_message = file_get_contents($template_file);
   $email_message = str_replace("#DEADLINE#", $deadline_str, $email_message);
   $email_message = str_replace("#WHOAMI#", $data_array['whoami'], $email_message);
   $email_message = str_replace("#DATE#", date("F d, Y h:i a"), $email_message);
   $email_message = str_replace("#NAME#", $data_array['name'], $email_message);
   $email_message = str_replace("#EMAIL#", $data_array['email'], $email_message);
   $email_message = str_replace("#IP#", $_SERVER['HTTP_X_FORWARDED_FOR'], $email_messag
e);
   $email_message = str_replace("#AGENT#", $_SERVER['HTTP_USER_AGENT'], $email_message)
;
   $email_message = str_replace("#SUBJECT#", $data_array['subject'], $email_message);
   $email_message = str_replace("#MESSAGE#", $data_array['message'], $email_message);
   $email_message = str_replace("#FOUND#", $data_array['found'], $email_message);

   #include whether or not to contact the customer with offers in the future
   $contact = "";
   if (isset($data_array['update1'])) {
      $contact = $contact." Please email updates about your products.<br/>";
   }
   if (isset($data_array['update2'])) {
      $contact = $contact." Please email updates about products from third-party partne
rs.<br/>";
   }
   $email_message = str_replace("#CONTACT#", $contact, $email_message);


   #construct the email headers
   $to = "support@example.com";  //for testing purposes, this should be YOUR email addr
ess.
   $from = $data_array['email'];
   $email_subject = "CONTACT #".time().": ".$data_array['subject'];

   $headers  = "From: " . $from . "\r\n";
   $headers .= 'MIME-Version: 1.0' . "\n";  //these headers will allow our HTML tags to
 be displayed in the email
   $headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";

   #now mail
   mail($to, $email_subject, $email_message, $headers);

}


#Remember, if you place any output before a header() call, you'll get an error.
#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening
```

```
}

extract($_GET, EXTR_PREFIX_SAME, "get");

#we want a deadline 2 days after the message date.
   $deadline_array = getdate();
   $deadline_day = $deadline_array['mday'] + 2;

   $deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$deadli
ne_array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
   $deadline_str = date("F d, Y", $deadline_stamp);


//DOCUMENT_ROOT is the file path leading up to the template name.
mail_message($_GET, $_SERVER['DOCUMENT_ROOT']."/email_template.txt", $deadline_str);


include($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br/>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


include($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");


?>
```

Save **contact.php**, then view and submit the form in **contact_form.php**. If you used your own email address as the **$to** variable, you should have received an email in your INBOX like before. However, this time it should look a little better.

You should have received an email like this:

```
Date: Thu, 8 Jun 2006 17:03:11 -0500
From: trish@myemail.com
To: support@acmeinc.com
Subject: CONTACT #1149804191: Please help!

You have just received a customer email.  Please respond to this email by June 10,
Details are below:

                Message Type: support
                Message Date: June 08, 2006 05:03 pm
                        Name: Trish
                       Email: trish@myemail.com
                  IP Address: 12.149.132.162
                    Platform: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWe
```

Subject: Please help!

I can't get the darn thing to work!

```
This customer found us through wordofmouth.
Please email updates about your products.
```

Take another look at the code:

```
function mail_message($data_array, $template_file, $deadline_str) {

   #get template contents, and replace variables with data
   $email_message = file_get_contents($template_file);
   $email_message = str_replace("#DEADLINE#", $deadline_str, $email_message);
   $email_message = str_replace("#WHOAMI#", $data_array['whoami'], $email_message);

.
.
.
}
.
.
.
mail_message($_GET, $_SERVER['DOCUMENT_ROOT']."/email_template.txt", $deadline_str);
```

Here, we created a function called **mail_message()**, which takes three parameters -- **$data_array**, **$template_file**, and **$deadline_str**. **$data_array** contains all the form data, because we pass the **$_GET** superglobal array into it. **$template_file** is the full path to the template file we want to use - in our case, we passed in the path to "**email_template.txt**" that we created earlier. And **$deadline_str** is the formatted string of the date by which we want the message answered.

We used the built-in PHP function **file_get_contents()** to import our email template file into a string, **$email_message**. Then, one by one, we replace each of our template variables with the corresponding form data, using the built-in function **str_replace()**. Go to php.net to read more about file_get_contents() or str_replace().

By making the support email easier to read -- and obtaining as much user information as possible -- you've improved efficiency in Acme's customer support process. Go ahead, demand a raise. You deserve it.

Allowing Users to Download Files

To make things a little more interesting, it turns out that Acme wants every customer who sends in a support email to

be allowed to download its informational brochure, a PDF document.

Now, technically you could just include a link to the PDF document itself, if the document is in a web-accessible directory. However, most of the time corporations don't want their downloadable files to be in a public area for anyone and everyone to download. This is especially true when electronic documents are for purchase, like marketing reports or copyrighted materials.

In your case, we've placed the brochure, called **acme_brochure.pdf**, in a hidden directory called **.php_files/** within your account. You can't view this file through the web, but you need to allow web users of your choosing to download it. What do you do?

In PHP, create a new file, called download.php:

```
<?php

$filepath = $_SERVER['DOCUMENT_ROOT']."/.php_files/acme_brochure.pdf";
if (file_exists($filepath)) {
   header("Content-Type: application/force-download");
   header("Content-Disposition:filename=\"brochure.pdf\"");
   $fd = fopen($filepath,'rb');
   fpassthru($fd);
   fclose($fd);
}

?>
```

Save **download.php**.

Now, switch back to contact.php and make the following changes, in green and blue:

```
<?php

function mail_message($data_array, $template_file, $deadline_str) {

   #get template contents, and replace variables with data
   $email_message = file_get_contents($template_file);
   $email_message = str_replace("#DEADLINE#", $deadline_str, $email_message);
   $email_message = str_replace("#WHOAMI#", $data_array['whoami'], $email_message);
   $email_message = str_replace("#DATE#", date("F d, Y h:i a"), $email_message);
   $email_message = str_replace("#NAME#", $data_array['name'], $email_message);
   $email_message = str_replace("#EMAIL#", $data_array['email'], $email_message);
   $email_message = str_replace("#IP#", $_SERVER['HTTP_X_FORWARDED_FOR'], $email_messag
e);
   $email_message = str_replace("#AGENT#", $_SERVER['HTTP_USER_AGENT'], $email_message)
;
   $email_message = str_replace("#SUBJECT#", $data_array['subject'], $email_message);
   $email_message = str_replace("#MESSAGE#", $data_array['message'], $email_message);
   $email_message = str_replace("#FOUND#", $data_array['found'], $email_message);

   #include whether or not to contact the customer with offers in the future
   $contact = "";
   if (isset($data_array['update1'])) {
      $contact = $contact." Please email updates about your products.<br/>";
   }
   if (isset($data_array['update2'])) {
      $contact = $contact." Please email updates about products from third-party partne
rs.<br/>";
   }
   $email_message = str_replace("#CONTACT#", $contact, $email_message);


   #construct the email headers
   $to = "support@example.com";  //for testing purposes, this should be YOUR email addr
ess.
   $from = $data_array['email'];
   $email_subject = "CONTACT #".time().": ".$data_array['subject'];

   $headers  = "From: " . $from . "\r\n";
   $headers .= 'MIME-Version: 1.0' . "\n";  //these headers will allow our HTML tags to
 be displayed in the email
   $headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";

   #now mail
   mail($to, $email_subject, $email_message, $headers);

}


#Remember, if you place any output before a header() call, you'll get an error.
#We used the superglobal $_GET here instead of the register globals, for safety
if (!($_GET['name'] && $_GET['email'] && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

extract($_GET, EXTR_PREFIX_SAME, "get");
```

```
#we want a deadline 2 days after the message date.
   $deadline_array = getdate();
   $deadline_day = $deadline_array['mday'] + 2;

   $deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$deadli
ne_array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
   $deadline_str = date("F d, Y", $deadline_stamp);

//DOCUMENT_ROOT is the file path leading up to the template name.
mail_message($_GET, $_SERVER['DOCUMENT_ROOT']."/email_template.txt", $deadline_str);

include($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br/>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$name."<br/>";
echo "Email: ".$email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];


?>
<br/><br/><a href="download.php"><b>Download our PDF brochure!</b></a>
<?


include($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");


?>
```

Save **contact.php**, then view **contact_form.php** and submit the form:

It should look something like this:

ACME, INC.

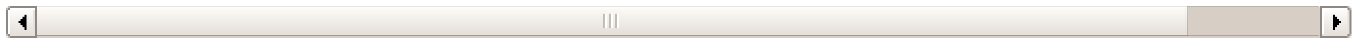
links go here

Thank you!

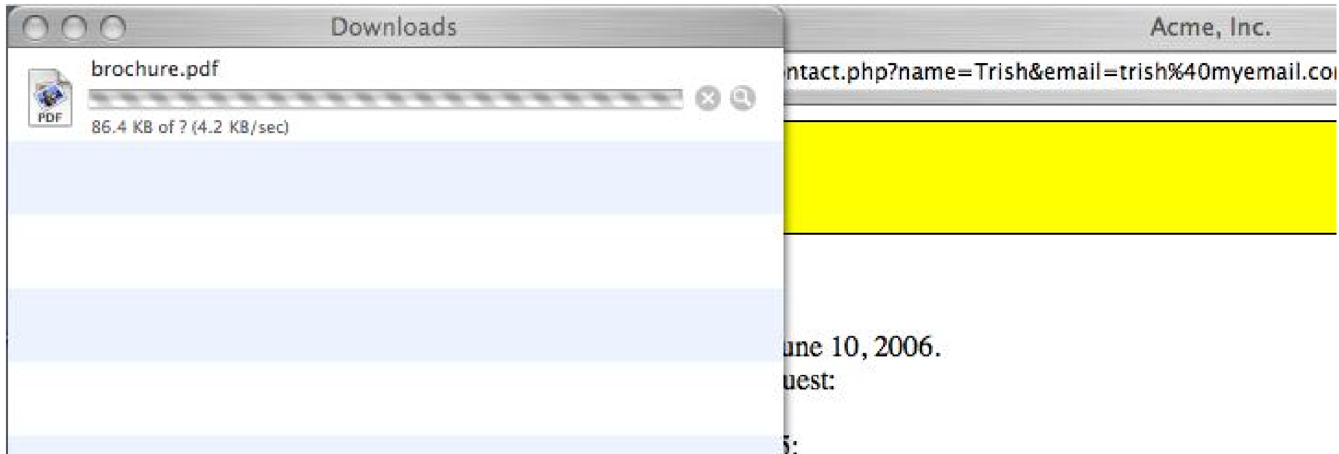
We'll get back to you by June 10, 2006.
Here is a copy of your request:

CONTACT #1149809625:
Message Date: June 08, 2006 06:33 pm
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please Help!
Message: I can/'t get the darn thing to work!
How you heard about us: wordofmouth
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X; er
AppleWebKit/418 (KHTML, like Gecko) Safari/417.9.2
The IP address of the computer you're working on is 63.171.219.74

Download our PDF brochure!



Now click the link. Did the PDF file download to your computer? You may have seen something like this:



How were we able to do that? Take another look at the code in download.php:

OBSERVE:

```
$filepath = $_SERVER['DOCUMENT_ROOT']."/.php_files/acme_brochure.pdf";
if (file_exists($filepath)) {
   header("Content-Type: application/force-download");
   header("Content-Disposition:filename=\"brochure.pdf\"");
   $fd = fopen($filepath,'rb');
   fpassthru($fd);
   fclose($fd);
}
```

First, the built-in function **file_exists()** does exactly what it says - it returns TRUE or FALSE based upon the existence of the parameter **$filepath**, which we set to the path of Acme's hidden PDF brochure in our account. Since it does exist, we use **header()** to output two **HTTP headers**. The header **"Content-Type"** is extremely important, as it tells the web browser that we are preparing to download data that is NOT in an HTML or text format, but in fact an application. Find out what happens if you leave this header out. The header **"Content-Disposition"** is optional, but we used it to create a generic name for the downloaded file.

Note In the case of PDF files, you can also use the header **"Content-Type: application/pdf"**. What's the difference? Some browsers allow PDF files to be opened within the browser itself, without having to download them to the computer's hard drive. Try it out and see what happens in your own browser.

Then, the built-in function **fopen()** creates a **file stream** pointing to our acme_brochure.pdf file, and *binds* it to the handle **$fd**. The parameter **'rb'** specifies that the file should be opened in **read-only, binary mode** -- binary, again, because it's not a text file. **fpassthru()** then sends all the file data through to the **output buffer** -- and because we specified through **header()** what the browser should do with that output, this launches your computer's download manager. **fclose()** simply closes the **file stream** **$fd**, to clean things up.

Don't forget to **Save** your work and hand in your **assignments** from your syllabus. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Cookies and Sessions

Learning about **cookies** and **sessions** is essential for programming PHP in the 21st century. You see, web users just aren't as patient as they used to be - they want websites that are incredibly easy for them to use and reuse, without having to repeat themselves over and over again. And their attention spans are shorter as well, meaning corporate websites in particular must compete by targeting the user as specifically as possible.

"Know Thy User", as they say. But how?

Using Cookies

Mmmm, cookies. Well, no, not *those* kinds of cookies. Although we would certainly revisit a web site for free cookies any day, unfortunately, downloading chocolate-chip goodness just hasn't been invented yet. *Sigh...*

Okay, so what *are* **browser cookies**? Let's find out. Fire up CodeRunner in **PHP**, and ***open*** your files **contact_form.php** and **contact.php**.

Add the following to contact_form.php, in green and blue:

```
<?php


require($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


if ($_GET['error'] == "1") {
   $error_code = 1;  //this means that there's been an error and we need to notify the
customer
}



?>

<h3>Contact ACME Corporation</h3>
<?
if ($error_code) {
echo "<div style='color:red'>Please help us with the following:</div>";
}
?>
<form method="GET" action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<?
if ($_COOKIE['name']) {
   echo $_COOKIE['name'];
}
else {
?>
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>" />
<input type="checkbox" name="remember" /> Remember me on this computer
<?
}
if ($error_code && !($_GET['name'] || $_COOKIE['name'])) {
   echo "<b>Please include your name.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<?
if ($_COOKIE['email']) {
   echo $_COOKIE['email'];
}
else {
?>
<input type="text" size="25" name="email" value="<? echo $_GET['email']; ?>" />
<?
}
if ($error_code && !($_GET['email'] || $_COOKIE['email'])) {
   echo "<b>Please include your email address.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Type of Request:
</td>
```

```
<td align="left">
<select name="whoami">
<option value="" />Please choose...
<option value="newcustomer"<?
if ($_GET['whoami'] == "newcustomer") {
   echo " selected";
}
?> />I am interested in becoming a customer.
<option value="customer"<?
if ($_GET['whoami'] == "customer") {
   echo " selected";
}
?> />I am a customer with a general question.
<option value="support"<?
if ($_GET['whoami'] == "support") {
   echo " selected";
}
?> />I need technical help using the website.
<option value="billing"<?
if ($_GET['whoami'] == "billing") {
   echo " selected";
}
?> />I have a billing question.
</select>
<?
if ($error_code && !($_GET['whoami'])) {
   echo "<b>Please choose a request type.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="<? echo $_GET['subject']; ?
>" />
<?
if ($error_code && !($_GET['subject'])) {
   echo "<b>Please add a subject for your request.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
<textarea name="message" cols="50" rows="8">
<? echo $_GET['message']; ?>
</textarea>
<?
if ($error_code && !($_GET['message'])) {
   echo "<b>Please fill in a message for us.</b>";
}
?>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth" />Word of Mouth<br/>
<input type="radio" name="found" value="search" />Online Search<br/>
<input type="radio" name="found" value="article" />Printed publication/article<br/>
<input type="radio" name="found" value="website" />Online link/article<br/>
```

```
<input type="radio" name="found" value="other" />Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked="checked" />Please email me updates about
 your products.<br/>
<input type="checkbox" name="update2" />Please email me updates about products from thi
rd-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT" />
</td></tr>
</table>
</form>

<?
   require($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");
?>
```

Be sure and ***Save*** **contact_form.php**, then Preview.

You should see something like this:

ACME, INC.

links go here

Contact ACME Corporation

Name: [ ] ☐ Remember me on th computer

Email: [ ]

Type of Request: Please choose...

Subject: [ ]

Message: [ ]

How did you hear about us?

- ○ Word of Mouth
- ○ Online Search
- ○ Printed publication/article
- ○ Online link/article
- ○ Other

☑ Please email me updates about your products.

☐ Please email me updates about products from third-party partners.

Now, switch back to contact.php and make the following changes, in green:

```
<?php

function mail_message($data_array, $template_file, $deadline_str, $myname, $myemail) {

   #get template contents, and replace variables with data
   $email_message = file_get_contents($template_file);
   $email_message = str_replace("#DEADLINE#", $deadline_str, $email_message);
   $email_message = str_replace("#WHOAMI#", $data_array['whoami'], $email_message);
   $email_message = str_replace("#DATE#", date("F d, Y h:i a"), $email_message);
   $email_message = str_replace("#NAME#", $myname, $email_message);
   $email_message = str_replace("#EMAIL#", $myemail, $email_message);
   $email_message = str_replace("#IP#", $_SERVER['HTTP_X_FORWARDED_FOR'], $email_messag
e);
   $email_message = str_replace("#AGENT#", $_SERVER['HTTP_USER_AGENT'], $email_message)
;
   $email_message = str_replace("#SUBJECT#", $data_array['subject'], $email_message);
   $email_message = str_replace("#MESSAGE#", $data_array['message'], $email_message);
   $email_message = str_replace("#FOUND#", $data_array['found'], $email_message);

   #include whether or not to contact the customer with offers in the future
   $contact = "";
   if (isset($data_array['update1'])) {
      $contact = $contact." Please email updates about your products.<br/>";
   }
   if (isset($data_array['update2'])) {
      $contact = $contact." Please email updates about products from third-party partne
rs.<br/>";
   }
   $email_message = str_replace("#CONTACT#", $contact, $email_message);


   #construct the email headers
   $to = " ReplaceWithYourOwnEmailAddress@oreillyschool.com";  //for testing purposes,
this should be YOUR email address.
   $from = $data_array['email'];
   $email_subject = "CONTACT #".time().": ".$data_array['subject'];

$headers  = "From: " . $from . "\r\n";
$headers .= 'MIME-Version: 1.0' . "\n";
$headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";   #now mail
   mail($to, $email_subject, $email_message, $headers);

}


$customer_name = $_COOKIE['name'];
if (!($customer_name)) {
   $customer_name = $_GET['name'];
}
$customer_email = $_COOKIE['email'];
if (!($customer_email)) {
   $customer_email = $_GET['email'];
}


#Remember, if you place any output before a header() call, you'll get an error.
#We used the superglobal $_GET here
if (!($customer_name && $customer_email && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";

   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
```

```
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

#we want a deadline 2 days after the message date.
   $deadline_array = getdate();
   $deadline_day = $deadline_array['mday'] + 2;

   $deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$deadli
ne_array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
   $deadline_str = date("F d, Y", $deadline_stamp);


if (isset($_GET['remember'])) {
   #the customer wants us to remember him/her for next time
   ### set errcode cookie
                /*
                cookie expires in one year
                365 days in a year
                24 hours in a day
                60 minutes in an hour
                60 seconds in a minute
                */
   $mytime = time() + (365 * 24 * 60 * 60);
   setcookie("name",$customer_name,$mytime);
   setcookie("email",$customer_email,$mytime);
}


//DOCUMENT_ROOT is the file path leading up to the template name.
mail_message($_GET, $_SERVER['DOCUMENT_ROOT']."/email_template.txt", $deadline_str, $cu
stomer_name, $customer_email);

include($_SERVER['DOCUMENT_ROOT']."/template_top.inc");

extract($_GET, EXTR_PREFIX_SAME, "get");

echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br/>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$customer_name."<br/>";
echo "Email: ".$customer_email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];

?>
<br/><br/><a href="download.php"><b>Download our PDF brochure!</b></a>
<?
```

```
include($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");

?>
```

Save **contact.php**, switch to **contact_form.php**, and Preview. This time, however, when you submit the form, be sure to check the box that says "Remember me on this computer."

ACME, INC.

links go here

Thank you!

We'll get back to you by June 11, 2006.
Here is a copy of your request:

CONTACT #1149880113:
Message Date: June 09, 2006 02:08 pm
Name: Trish
Email: trish@myemail.com
Type of Request: support
Subject: Please help!
Message: I can/'t get the darn thing to work!
How you heard about us: wordofmouth
update1: on
update2:
You are currently working on Mozilla/5.0 (Macintosh; U; PPC Mac OS X Ma
en-US; rv:1.7.12) Gecko/20050915 Firefox/1.0.7
The IP address of the computer you're working on is 63.171.219.74

Download our PDF brochure!

Looks pretty much the same as before. What's changed? To find out, now go back to **contact_form.php** and *RELOAD* the page:

ACME, INC.

links go here

Contact ACME Corporation

Name: Trish
Email: trish@myemail.com

Type of Request: Please choose...

Subject:

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

- [x] Please email me updates about your products.
- [ ] Please email me updates about products from third-party partners.

SUBMIT

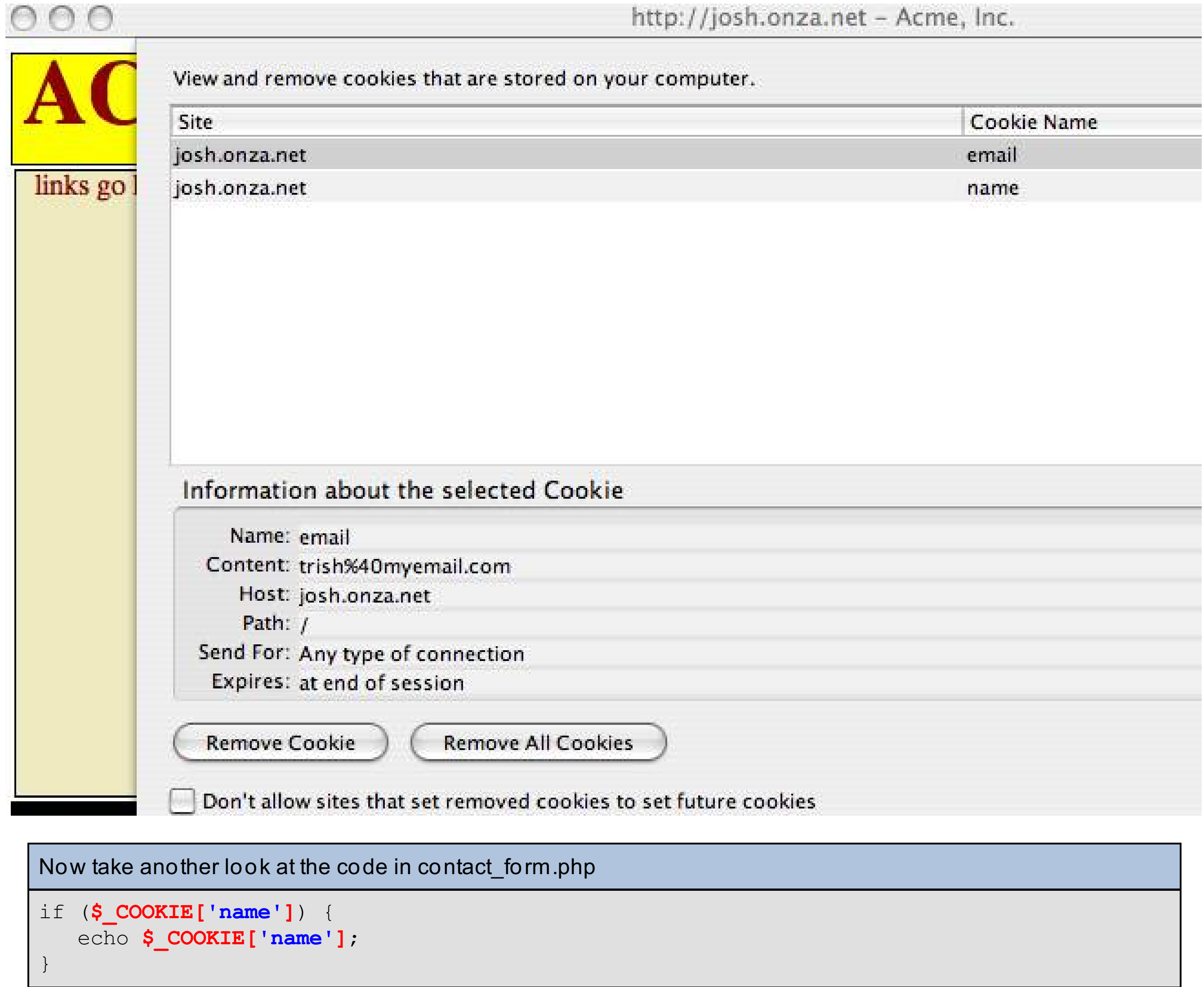
And there we are! The form is indeed remembering us, and even if you exit your browser entirely and come back, your name and email would still be there. But how were we able to do it? Using **cookies**.

Take another look at the code in contact.php:

```
if (isset($_GET['remember'])) {
   #the customer wants us to remember him/her for next time
   ### set errcode cookie
                /*
                cookie expires in one year
                365 days in a year
                24 hours in a day
                60 minutes in an hour
                60 seconds in a minute
                */
   $mytime = time() + (365 * 24 * 60 * 60);
   setcookie("name",$customer_name,$mytime);
   setcookie("email",$customer_email,$mytime);
}
```

Here, we're using the built-in PHP function **setcookie()** with three parameters: **"name"** and **"email"** are the names we're giving the respective cookies, and **$customer_name** and **$customer_email** are the values that we got from the **$_GET** superglobal. **$mytime** is the **timestamp** at which we want the cookies to expire - since it's measured in seconds, we simply took **time()** and added enough seconds to make 1 year.

Browser cookies are simply variables that are stored within the user's browser on his/her computer. If you look in your own browser preferences, you can actually view all the cookies that are set:



Now take another look at the code in contact_form.php

```
if ($_COOKIE['name']) {
   echo $_COOKIE['name'];
}
```

Just like **$_GET** and **$_POST** store values set by the user, and **$_SERVER** and **$_ENV** store values set by the environment, **$_COOKIE** is a **superglobal array** -- but this time the values being stored are set by you, the programmer.

Before cookies, once a user left a website, that site had no way recognizing that user when she came back. Basically, the user had to start from scratch every time. No shopping carts, personalized home pages, or pre-filled forms. So as you can see, introducing cookies opened up a world of power and convenience that have made them invaluable to web programming.

Knowing the User Through Sessions

Of course, there are a couple of downfalls to using cookies. One is that different browsers have different restrictions on the number and size of cookies - some allow unlimited numbers but small sizes, others allow large cookies but only up to 10.

But the main problem with cookies is privacy. Anyone who uses the same browser that you used - unless you deleted your cookies before you left - can now view your name and email in the browser cookie list. Think if that had been even more sensitive information, like usernames or financial information. Yikes! Let's try fixing this.

Add the following to contact_form.php, in green:

```
<?php

#start the session before any output
session_start();


require($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


if ($_GET['error'] == "1") {
   $error_code = 1;  //this means that there's been an error and we need to notify the
customer
}

?>

<h3>Contact ACME Corporation</h3>
<?
if ($error_code) {
echo "<div style='color:red'>Please help us with the following:</div>";
}
?>
<form method="GET" action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<?
if ($_SESSION['name']) {
   echo $_SESSION['name'];
}
else {
?>
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>" />
<input type="checkbox" name="remember" /> Remember me on this computer
<?
}
if ($error_code && !($_GET['name'] || $_SESSION['name'])) {
   echo "<b>Please include your name.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<?
if ($_SESSION['email']) {
   echo $_SESSION['email'];
}
else {
?>
<input type="text" size="25" name="email" value="<? echo $_GET['email']; ?>" />
<?
}
if ($error_code && !($_GET['email'] || $_SESSION['email'])) {
   echo "<b>Please include your email address.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
```

```
Type of Request:
</td>
<td align="left">
<select name="whoami">
<option value="" />Please choose...
<option value="newcustomer"<?
if ($_GET['whoami'] == "newcustomer") {
   echo " selected";
}
?> />I am interested in becoming a customer.
<option value="customer"<?
if ($_GET['whoami'] == "customer") {
   echo " selected";
}
?> />I am a customer with a general question.
<option value="support"<?
if ($_GET['whoami'] == "support") {
   echo " selected";
}
?> />I need technical help using the website.
<option value="billing"<?
if ($_GET['whoami'] == "billing") {
   echo " selected";
}
?> />I have a billing question.
</select>
<?
if ($error_code && !($_GET['whoami'])) {
   echo "<b>Please choose a request type.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="<? echo $_GET['subject']; ?
>" />
<?
if ($error_code && !($_GET['subject'])) {
   echo "<b>Please add a subject for your request.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
<textarea name="message" cols="50" rows="8">
<? echo $_GET['message']; ?>
</textarea>
<?
if ($error_code && !($_GET['message'])) {
   echo "<b>Please fill in a message for us.</b>";
}
?>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth" />Word of Mouth<br/>
<input type="radio" name="found" value="search" />Online Search<br/>
```

```
<input type="radio" name="found" value="article" />Printed publication/article<br/>
<input type="radio" name="found" value="website" />Online link/article<br/>
<input type="radio" name="found" value="other" />Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked="checked" />Please email me updates about
 your products.<br/>
<input type="checkbox" name="update2" />Please email me updates about products from thi
rd-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT" />
</td></tr>
</table>
</form>

<?
   require($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");
?>
```

Be sure to ***Save* contact_form.php.**

Now switch to contact.php and make the following changes, in green:

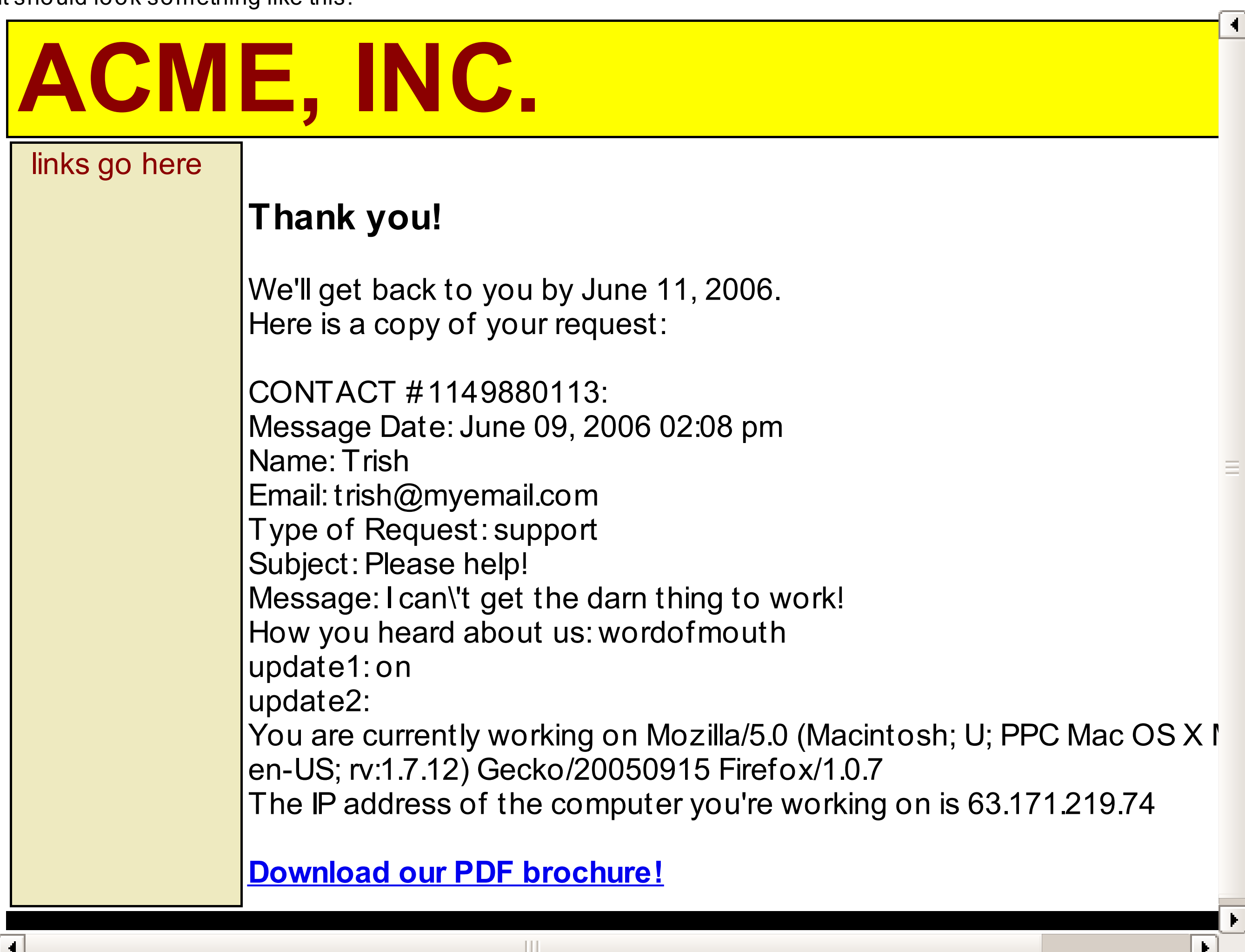
```
<?php

function mail_message($data_array, $template_file, $deadline_str, $myname, $myemail) {

   #get template contents, and replace variables with data
   $email_message = file_get_contents($template_file);
   $email_message = str_replace("#DEADLINE#", $deadline_str, $email_message);
   $email_message = str_replace("#WHOAMI#", $data_array['whoami'], $email_message);
   $email_message = str_replace("#DATE#", date("F d, Y h:i a"), $email_message);
   $email_message = str_replace("#NAME#", $myname, $email_message);
   $email_message = str_replace("#EMAIL#", $myemail, $email_message);
   $email_message = str_replace("#IP#", $_SERVER['HTTP_X_FORWARDED_FOR'], $email_messag
e);
   $email_message = str_replace("#AGENT#", $_SERVER['HTTP_USER_AGENT'], $email_message)
;
   $email_message = str_replace("#SUBJECT#", $data_array['subject'], $email_message);
   $email_message = str_replace("#MESSAGE#", $data_array['message'], $email_message);
   $email_message = str_replace("#FOUND#", $data_array['found'], $email_message);

   #include whether or not to contact the customer with offers in the future
   $contact = "";
   if (isset($data_array['update1'])) {
      $contact = $contact." Please email updates about your products.<br/>";
   }
   if (isset($data_array['update2'])) {
      $contact = $contact." Please email updates about products from third-party partne
rs.<br/>";
   }
   $email_message = str_replace("#CONTACT#", $contact, $email_message);


   #construct the email headers
   $to = " ReplaceWithYourOwnEmailAddress@oreillyschool.com";  //for testing purposes,
this should be YOUR email address.
   $from = $data_array['email'];
   $email_subject = "CONTACT #".time().": ".$data_array['subject'];

$headers  = "From: " . $from . "\r\n";
$headers .= 'MIME-Version: 1.0' . "\n";
$headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";    #now mail
   mail($to, $email_subject, $email_message, $headers);

}


#start the session
session_start();

$customer_name = $_SESSION['name'];
if (!($customer_name)) {
   $customer_name = $_GET['name'];
}

$customer_email = $_SESSION['email'];
if (!($customer_email)) {
   $customer_email = $_GET['email'];
}

#Remember, if you place any output before a header() call, you'll get an error.
#We used the superglobal $_GET here
if (!($customer_name && $customer_email && $_GET['whoami']
      && $_GET['subject'] && $_GET['message'])) {

   #with the header() function, no output can come before it.
   #echo "Please make sure you've filled in all required information.";
```

```
   $query_string = $_SERVER['QUERY_STRING'];
   #add a flag called "error" to tell contact_form.php that something needs fixed
   $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php?".$query_string."&error=1"
;
   header("Location: ".$url);
   exit();  //stop the rest of the program from happening

}

#we want a deadline 2 days after the message date.
   $deadline_array = getdate();
   $deadline_day = $deadline_array['mday'] + 2;

   $deadline_stamp = mktime($deadline_array['hours'],$deadline_array['minutes'],$deadli
ne_array['seconds'],
       $deadline_array['mon'],$deadline_day,$deadline_array['year']);
   $deadline_str = date("F d, Y", $deadline_stamp);


if (isset($_GET['remember'])) {
   #the customer wants us to remember him/her for next time
   $_SESSION['name'] = $customer_name;
   $_SESSION['email'] = $customer_email;
}


//DOCUMENT_ROOT is the file path leading up to the template name.
mail_message($_GET, $_SERVER['DOCUMENT_ROOT']."/email_template.txt", $deadline_str, $cu
stomer_name, $customer_email);

include($_SERVER['DOCUMENT_ROOT']."/template_top.inc");

extract($_GET, EXTR_PREFIX_SAME, "get");

echo "<h3>Thank you!</h3>";
echo "We'll get back to you by ".$deadline_str.".<br/>";
echo "Here is a copy of your request:<br/><br/>";
echo "CONTACT #".time().":<br/>";
echo "Message Date: ".date("F d, Y h:i a")."<br/>";
echo "Name: ".$customer_name."<br/>";
echo "Email: ".$customer_email."<br/>";
echo "Type of Request: ".$whoami."<br/>";
echo "Subject: ".$subject."<br/>";
echo "Message: ".$message."<br/>";
echo "How you heard about us: ".$found."<br/>";

for ($i = 1; $i <= 2; $i++) {
   $element_name = "update".$i;
   echo $element_name.": ";
   echo $$element_name;
   echo "<br/>";
}

echo "You are currently working on ".$_SERVER['HTTP_USER_AGENT'];
echo "<br/>The IP address of the computer you're working on is ".$_SERVER['HTTP_X_FORWA
RDED_FOR'];

?>
<br/><br/><a href="download.php"><b>Download our PDF brochure!</b></a>
<?


include($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");


?>
```

***Save* contact.php**, then switch to contact_form.php and Preview. You'll notice that you have to re-enter your name

and email address again, but not for long. Be sure to click the "Remember me on this computer" checkbox when you submit the form. What did you get?

It should look something like this:



Again, it looks exactly the same as always. But, if you go back to **contact_form.php** and *RELOAD*, you'll get:

Something like this:

ACME, INC.

links go here

Contact ACME Corporation

Name: Trish
Email: trish@myemail.com

Type of Request: Please choose...

Subject:

Message:

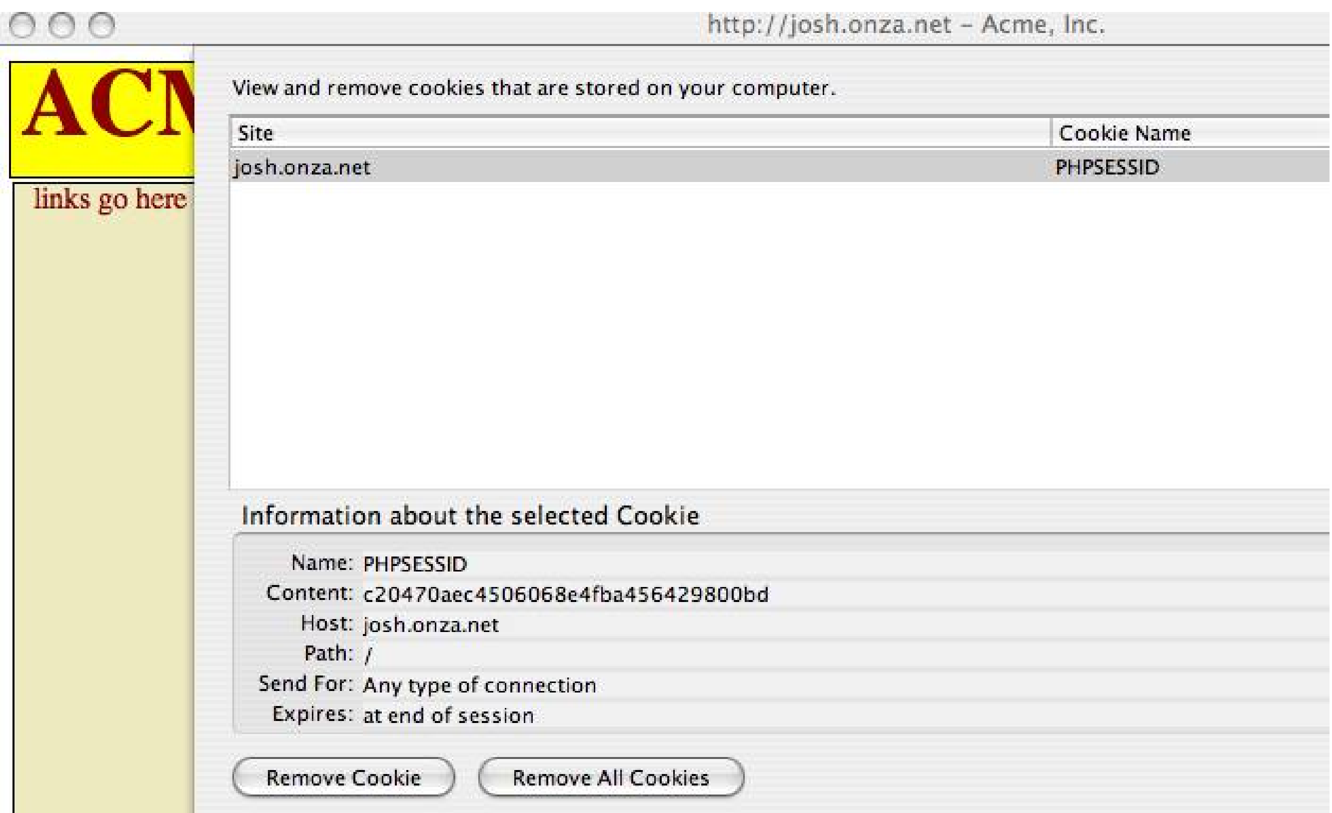
How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

- [x] Please email me updates about your products.
- [ ] Please email me updates about products from third-party partners.

Yes, it's exactly the same output as when you used **cookies** -- your name and email address are now magically saved within the browser.

So what's the difference? If you check out your browser's preferences and view the cookies stored there, you won't see your name and email address in there anymore. Instead, you'll see something like this:



Take another look at the code in contact.php:

```
#start the session
session_start();

$customer_name = $_SESSION['name'];
if (!($customer_name)) {
   $customer_name = $_GET['name'];
}

$customer_email = $_SESSION['email'];
if (!($customer_email)) {
   $customer_email = $_GET['email'];
}
.
.
.
if (isset($_GET['remember'])) {
   #the customer wants us to remember him/her for next time
   $_SESSION['name'] = $customer_name;
   $_SESSION['email'] = $customer_email;
}
```

Any time you want to use **sessions** in your PHP script, you must start the session first - using the PHP function **session_start()**. This way, the browser knows to pull up the **$_SESSION superglobal** using the **SESSION ID** that was set in your browser cookies. Once it's been pulled up, you can not only access the values using **$_SESSION**, you can *set* the values too.

Note It's important to stress that **session_start()** must be called **before** any output - much like **header()**.

Deleting Sessions

In case someone else visits our site using the same browser, we should give the user a way to end the session without waiting for it to expire.

Add the following to contact_form.php, in green and blue:

```
<?php

if (isset($_GET['delete_session'])) {
  session_start(); //must always use this command to access the session and its variables
  session_destroy(); //force the session to end

  //Add in a page reload so that the session_destroy() will take effect
  if($_SESSION && $_SESSION['name']){
    $url = "http://".$_SERVER['HTTP_HOST']."/contact_form.php";
    header("Location: ".$url);
  }
}
else {
   #start the session before any output
   session_start();
}

   require($_SERVER['DOCUMENT_ROOT']."/template_top.inc");


if ($_GET['error'] == "1") {
   $error_code = 1;  //this means that there's been an error and we need to notify the customer
}

?>

<h3>Contact ACME Corporation</h3>
<?
if ($error_code) {
   echo "<div style='color:red'>Please help us with the following:</div>";
}
?>
<form method="GET" action="contact.php">
<table>
<tr>
<td align="right">
Name:
</td>
<td align="left">
<?
if ($_SESSION['name']) {
   echo $_SESSION['name'];
?>
 <a href="contact_form.php?delete_session=1">Not <? echo $_SESSION['name']; ?>?</a>
<?
}
else {
?>
<input type="text" size="25" name="name" value="<? echo $_GET['name']; ?>" />
<input type="checkbox" name="remember" /> Remember me on this computer
<?
}
if ($error_code && !($_GET['name'] || $_SESSION['name'])) {
   echo "<b>Please include your name.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Email:
</td><td align="left">
<?
```

```
if ($_SESSION['email']) {
   echo $_SESSION['email'];
}
else {
?>
<input type="text" size="25" name="email" value="<? echo $_GET['email']; ?>" />
<?
}
if ($error_code && !($_GET['email'] || $_SESSION['email'])) {
   echo "<b>Please include your email address.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Type of Request:
</td>
<td align="left">
<select name="whoami">
<option value="" />Please choose...
<option value="newcustomer"<?
if ($_GET['whoami'] == "newcustomer") {
   echo " selected";
}
?> />I am interested in becoming a customer.
<option value="customer"<?
if ($_GET['whoami'] == "customer") {
   echo " selected";
}
?> />I am a customer with a general question.
<option value="support"<?
if ($_GET['whoami'] == "support") {
   echo " selected";
}
?> />I need technical help using the website.
<option value="billing"<?
if ($_GET['whoami'] == "billing") {
   echo " selected";
}
?> />I have a billing question.
</select>
<?
if ($error_code && !($_GET['whoami'])) {
   echo "<b>Please choose a request type.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right">
Subject:
</td>
<td align="left">
<input type="text" size="50" max="50" name="subject" value="<? echo $_GET['subje
ct']; ?>" />
<?
if ($error_code && !($_GET['subject'])) {
   echo "<b>Please add a subject for your request.</b>";
}
?>
</td>
</tr>
<tr>
<td align="right" valign="top">
Message:
</td>
<td align="left">
```

```
<textarea name="message" cols="50" rows="8">
<? echo $_GET['message']; ?>
</textarea>
<?
if ($error_code && !($_GET['message'])) {
   echo "<b>Please fill in a message for us.</b>";
}
?>
</td>
</tr>
<tr>
<td colspan="2" align="left">
How did you hear about us?
<ul>
<input type="radio" name="found" value="wordofmouth" />Word of Mouth<br/>
<input type="radio" name="found" value="search" />Online Search<br/>
<input type="radio" name="found" value="article" />Printed publication/article<b
r/>
<input type="radio" name="found" value="website" />Online link/article<br/>
<input type="radio" name="found" value="other" />Other
</ul>
</td>
</tr>
<tr>
<td colspan="2">
<input type="checkbox" name="update1" checked="checked" />Please email me update
s about your products.<br/>
<input type="checkbox" name="update2" />Please email me updates about products f
rom third-party partners.
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="SUBMIT" />
</td></tr>
</table>
</form>

<?
   require($_SERVER['DOCUMENT_ROOT']."/template_bottom.inc");
?>
```

Be sure to ***Save*** **contact_form.php**, then Preview.

It should look something like this:

ACME, INC.

links go here

Contact ACME Corporation

Name: Trish Not Trish?
Email: trish@myemail.com

Type of Request: Please choose...

Subject:

Message:

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

☑ Please email me updates about your products.

Try clicking the link to see what happens:

ACME, INC.

links go here

Contact ACME Corporation

Name: [ ] Remember me on this computer

Email: [ ]

Type of Request: Please choose...

Subject: [ ]

Message: [ ]

How did you hear about us?

- Word of Mouth
- Online Search
- Printed publication/article
- Online link/article
- Other

Please email me updates about your products.

Ending the session was pretty straightforward, because session_destroy() will destroy all the session data for a user. If we wanted to delete just one session variable, we would use **unset($_SESSION['some_var'])**.

Congratulations! You've now learned the PHP skills needed to make a vast range of robust, commercial applications for the web. Are you ready for those skills to be tested? Make sure you have ***Saved*** your work and handed in the **assignments** for this lesson. Then, it's time for your **final project**.

Good luck! We know you can do it.

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

Final Project

Final Project

The overall goal of this project is to create a shopping cart, with products, prices, registration, and a checkout area. You can make this shopping cart any way you wish.

For the sake of evaluation, try to include as many elements discussed in this course as you can. For instance, you should use arrays for products, functions for various program tasks, template files, form validation, and cookies/sessions for cart persistence. You are encouraged to observe good programming practices, with comments, code reusability and readability.

You can hand in up to five files, but you don't have to create that many if you don't want to.

Be creative and have fun! You want to present yourself in a professional yet friendly way, so feel free to express yourself!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.