

03 C0021  
KAL-0204-51

FRONTIERS WEB PROGRAMMING  
An Interactive Qualifying Project Report


Submitted to the Faculty  
Of the  
WORCESTER POLYTECHNIC INSTITUTE  
in partial fulfilment of the requirements for the  
Degree of Bachelor of Science

by

---

Ravi Patel

Date

  
Reid Ransom

1/3/03  
Date

Approved:

---

Professor Karen A. Lemone, Advisor

The Computer Science Program in Frontiers is an on-campus research and learning experience that teaches students the fundamentals of web programming and an introduction to object oriented programming. The goal of our project was to consider different learning styles when creating the course program. We gave the students a pre-test and a post-test to measure how much they learned during the program in the areas of HTML, JavaScript, Perl/CGI, and Java with applets.

1	Introduction.....	5
2	Background Research .....	7
2.1	Educational Considerations .....	7
2.1.1	Structure.....	7
2.1.2	Motivation.....	8
2.1.3	Active Engagement.....	9
2.1.4	Collaboration & Interdependence.....	10
2.1.5	Content.....	11
2.1.6	Assessment.....	12
2.2	Frontiers .....	13
2.3	Internet Programming .....	14
2.3.1	HTML .....	14
2.3.2	JavaScript.....	15
2.3.3	Perl/CGI .....	17
3	Methodology .....	19
3.1	Structure .....	19
3.2	Motivation.....	20
3.3	Active Engagement.....	20
3.4	Collaboration and Interdependence .....	21
3.5	Content.....	21
3.6	Assessment.....	21
4	Results and Analysis .....	23
4.1	Week One.....	23
4.1.1	General Lab Organization.....	24
4.1.2	Lab One: Creating Web Pages.....	25
4.1.3	Lab Two: JavaScript .....	26
4.1.4	Lab Three: Perl/CGI .....	28
4.1.5	Lab Four: Java Applets .....	29
4.2	Week Two .....	30
4.3	Pre-Test.....	31
4.4	Quizzes.....	32
4.3.1	Quiz One – HTML.....	32
4.3.2	Quiz Two – JavaScript.....	32
4.3.3	Quiz Three – Perl.....	32
4.3.4	Quiz Four – Java .....	33
4.5	Post-Test .....	33
4.6	Individual Student Progress .....	35
4.6.1	Stock Ticker Project.....	36
4.6.1.1	Student 1.1 .....	36
4.6.1.2	Student 1.2 .....	36
4.6.1.3	Student 1.3 .....	37
4.6.2	Web Browser Project.....	37
4.6.2.1	Student 2.1 .....	37
4.6.2.2	Student 2.2 .....	38
4.6.3	E-Commerce Site.....	38
4.6.3.1	Student 3.1 .....	38

4.6.3.2 Student 3.2 .....	39
4.6.3.3 Student 3.3 .....	39
4.6.4 Cryptography Project – Cipher text Decryption .....	39
4.6.4.1 Student 4.1 .....	39
4.6.4.2 Student 4.2 .....	40
4.6.4.3 Student 4.3 .....	40
4.6.4.4 Student 4.4 .....	41
4.6.5 Meet the Sticks.....	41
4.6.5.1 Student 5.1 .....	41
4.6.5.2 Student 5.2 .....	41
4.6.5.3 Student 5.3 .....	42
4.6.6 Create a Fractal Interface .....	42
4.6.6.1 Student 6.1 .....	42
4.6.6.2 Student 6.2 .....	42
4.6.6.3 Student 6.3 .....	43
5 Conclusion .....	44
5.1 Summary .....	44
5.2 Future Work.....	45
References.....	47
Appendix A - Background Material .....	50
Appendix B - Activities .....	136
Appendix C - Student Evaluations.....	147
Appendix D - Frontiers Qualifying Projects.....	155

# 1 Introduction

The WPI Frontiers program is a two-week academic program that is designed to enlighten high school juniors and seniors to the WPI college experience. The program illustrates the nature of college work characteristically completed in the field they are interested in studying. We developed a curriculum for the Computer Science section of the Frontiers program. The two-week program was taught by both WPI faculty and WPI college students and was separated into two different elements, Week One and Week Two. During the first week, the goal intended for students was to work autonomously and learn web programming and an object oriented approach to several types of computer language. During the second week of the program, the students were placed into groups to complete a project collectively.

The objective of this Interdisciplinary Qualifying Project (IQP) project was to create a program, which facilitates independent study in the subject of Computer Science. After much contemplation, we decided to utilize an educational method where each student learns the material given to them on an individual basis, working at their own pace, and utilizing the resources provided: the instructors and the online tutorial. A large advantage from this method was that the student can begin in whatever computer language he or she is most comfortable.

To achieve this goal, we split all of the information into four labs: HTML, JavaScript, Perl/CGI, and Java Applets. Each lab was then split into three sections, the beginner, "the more", and the advance section. "The more" section was utilized as an intermediate section for students who had some computer language experience. After the lab reading was completed, each individual student was required to finish the activity assigned for that section. It was important for each student to accomplish three sections

total. After the student had finished all three sections in each lab, he or she was required to pass the quiz at the conclusion of the lab work. This segment of the program contributed to the first week of the class.

During the second week of the program, students worked together in groups to complete a Frontiers Qualifying Project (FQP). The goal of this assignment was to demonstrate to students how the "WPI Plan" worked. Students were split into groups and chose projects pertaining to their computer science interests. A prearranged block of time was allotted, equal to one week in order to complete this project; working in groups of two to four members. Upon completion, each group gave a detailed presentation to the class explaining the goal of their project and what they had accomplished.

The conventional method of teaching is by lecture where each student learns the course material at the same pace. Due to the fact that each student in the program had a different background in computer science, the goal of our project was to use a different method of teaching; where by each student could begin where they felt comfortable and work at their own pace. We did this providing each student with a goal, a time allowance, initial instruction, and individual help. The students also had plenty of time together in the computer lab where they could easily communicate with their fellow students for additional assistance. Each student thrived with this educational technique, and succeeded not only in the first week of individual exertion but also in the second week of group endeavors.

## **2 Background Research**

### **2.1 Educational Considerations**

The focus of this project is the educational considerations for online teaching. To do this we needed to “orchestrate the whole learning environment” (Lander 1997) rather than merely arranging computer activities. “International conferences, journals, working groups, even discussions in the public news make us believe that traditional education is about to be replaced by computer-supported teaching and learning.” (Ausserhofer 1999) When designing an educational program, emphasis needs to be placed on the use of technology to develop the learning environment and to produce a better understanding of the material, rather than following a program of instruction.

#### **2.1.1 Structure**

It is important to have a well worked out structure of activities taking place in the classroom. According to Lander, having a structure of classroom activities is equally crucial when teaching online. There are several different types of structures that one can follow. Traditionally, teaching has been structured with reference to curriculum content. “However, where students have some freedom to determine content, structure can be achieved by establishing a learning cycle that orchestrates and relates the various activities being used to reach the goals of course or subject.” (Lander 1997) In online teaching, all of the course material is provided on the Internet. This is one way to give students the freedom to determine content and to work at their own pace.

There are many different kinds of structural considerations that we needed to consider within an educational environment. One of the main theories concerns the structure being rehearsed by the online teacher prior to utilization. A curriculum has to

be more than just a schedule of events; "...it needs to encompass the social interactions between participants (including their roles and goal interdependence), motivation and opportunities for reflection and evaluation." (Lander 1997) Lander returns to the notion that it is crucial for the entire learning environment to be structured within specific confines of the material.

### **2.1.2 Motivation**

The connection between the course activities and students long term goals of obtaining credentials may be plenty to ensure motivation, but this is hardly ever the case, "because such distant extrinsic motivation is in competition with much more immediate forms related to the current concerns in their lives."(Lander 1997) There are three key elements to motivation, according to Lander: first, motivation depends on encouragement and excitement, second, Lander harps on the meaning of materials and activities that are present for the students, and last, the relevance of all the material being presented.

As an illustration of motivational ideas that teachers may have, they are sometimes required to give a "micro-world classroom" example of their ideas with the task of planning a number of lessons in a subject. The reason for this is that they might have to accommodate for numerous concealed conditions that are part of a class they are teaching. These concealed conditions include varied comprehension levels, curriculum frameworks, school discipline policy, and the expectations of other educators.

Another factor that we take into consideration for motivation is the design of the software that we use to teach the students. "In order for educational software to maintain its innovative edge, instructional designers need to access models that recognize the



variety of proposed guidelines for developing technology supported learning environments which support a constructivist approach.” (Harper 1997) The design of the software and the educational process has to be appealing to the student. If the design is not appealing, the students will start losing interest in the learning process.

### **2.1.3 Active Engagement**

Most people understand and comprehend better when they are actively participating rather than listening to a lecture. “People learn best by doing things, not by being passive recipients.” (Lander 1997) Active engagement is one of the major parts of motivation; people need to be active in the classroom, even in a class that is taught online. “Web-based educational systems are asynchronous, that is, they do not require simultaneous presence of teacher and students.”(Ausserhofer 1999) Online teaching offers many benefits such that the student is able to learn from mistakes when he or she is carrying out an action or solving a problem. This is not possible when the student is listening to the teacher give a lecture in class.

By accomplishing tasks on their own, students gain procedural as well as declarative knowledge. One of the biggest advantages of teaching a class online is that by putting the student in control of their own tasks, they are also in control of what they want to do and learn from doing. “They must do the searching, make the decisions, interact with multimedia, contribute to conferences and solve problems.” (Lander 1997) One of the most important issues needed in technology today is how to encourage students to actively engage in the process of carrying out activities, answering questions and solving problems.

### **2.1.4 Collaboration & Interdependence**

“Computer Supported Cooperative Learning (CSCL) is an approach to online teaching based on the important idea that students learn more when collaborating with others than they do studying in isolation.”(Lander 1997) Education is a social process; hence it should be treated as that. Students should be allowed to actively help each other, and also share ideas and knowledge based on what they are doing. By doing this, they are gaining a greater abundance of knowledge, and in turn are passing on the knowledge that they already embody. “CSCL is one of the most promising innovations to improve teaching and learning with the help of modern information and communication technology.” (Lehtinen 1999) Collaboration can be made very useful in online teaching.

CSCL is the process in which the students are encouraged to work together on the learning process. This process is very different from the traditional “direct transfer” model, which implies that the instructor is the distributor of knowledge. It is believed that CSCL will take the place of traditional learning methods used today. We are able to visualize this gradual transformation, where everything becomes a networked, information society and everyone becomes dependent on an online connection. Schools are becoming more dependent on networks and the use of networks. “Educational institutions are being forced to find better pedagogical methods to cope with these new challenges.” (Lehtinen 1999) CSCL has become a major part of many new learning styles for various schools.

As reviewed by Lehtinen, there are two kinds of evidence supporting the educational value of CSCL. First, the quality of social interaction among students and also between students and teachers is improving because of the introduction of computer technology. By introducing computers into the classroom, students interact in the

learning process much better than the traditional way of teaching. Second, "...there is a reasonable amount of published experiments showing positive learning effects when CSCL systems have been applied to classroom learning." (Lehtinen 1999) Even though these studies have been rather limited in the duration of the experiment, the number of participants, and the share of curriculum covered there are some important qualities in the results which make them noteworthy.

### **2.1.5 Content**

Online teaching changes the role of the educator in many ways. In the classroom, the teacher is at the center of everything that is going on. He or she has control over the activities and all the other interactions between students during class time. When online teaching, it is not possible for the teacher to be in contact with everyone at the same time. The students interact more with the computer by working on a problem alone. "The technology removes the teacher from the center to the periphery of the learning situation. In this position it is more appropriate for the teacher to adopt the role of designer of the learning environment, facilitator of the activities, mentor and coach to the students."(Lander)

Online teaching demands a different treatment of subject matter. In a classroom setting, the curriculum can be "tightly" specified by the teacher. The teacher has control of what the student will learn, what information the student will search for and what information he or she will investigate. When online teaching, the teacher loses direct control over the specific subject matter. The teacher does not have control of what the student will search or investigate.

By working self-paced, the student must take on more responsibility. By giving students control over what they learn, we, as educators, have to ask ourselves; will the student learn enough information about the subject to make wise decisions for a solution? There are many ways to handle this issue. One solution is to raise issues and generate questions to be answered by the student during the online teaching process. This enables the student to follow the class more efficiently and learn the subject material that is presented. (Lander)

### **2.1.6 Assessment**

Pre and Post tests are a very easy way to define what a student has learned already and what he or she has learned at the end. “The idea of pre and post testing of students is often accepted as a viable method to assess the extent to which an educational intervention has had an impact on student ‘learning’”. (Newton 1999) The pre-test is given to the student at some point when a course begins to find out what the student has already learned in their past education. The Post Test is given at the end of a course to judge the student and see what the student has learned during a class. Newton explains that, the instructor, to define clearly “why the evaluation is being performed, what is being evaluated, when the evaluation will be performed, and how it will be performed.” These questions better assist us in answering and defining how to judge the student’s progress and what kind of a test to give them. If we are able to understand why we are doing this assessment, we will be able to study the students with a greater amount of efficiency.

## **2.2 Frontiers**

Frontiers is a summer program offered at WPI for high school students entering their junior and senior year. The program runs for two weeks every July. During the program, students attend classes, which are specific to their prospective majors. Students are also given an account on WPI's computer system for accessing campus PCs and UNIX workstations.

The computer science section of the program is designed to be self paced because the students, coming from many different high schools, have diverse backgrounds with respect to computer programming. In the past, students with little or no programming experience were advised to begin learning with the RoBOTL programming language. RoBOTL is designed to teach beginners the basic concepts of object-oriented programming. With some experience, students could learn JavaBOTL. JavaBOTL is regular java programming except libraries are included which have RoBOTL-like methods or commands. JavaBOTL is the next step for students after learning RoBOTL or a good starting place for students with a little programming experience. Students with more object-oriented programming experience are encouraged to begin programming with java. Beginning lessons in java assumes the student has an understanding of elementary java syntax. Material covered in java lessons provides information about writing java applets.

There are often a number of students who are familiar with java programming and are capable of completing the course material before the end of the program. The main objective of this project was to give the Frontiers students of 2002 an introduction to web programming. Subject matter includes creating web pages, an introduction to JavaScript, server side programming with Perl CGI, and implementing java applets. The amount of

course material a student covered merely depends on their individual background with computer programming.

## **2.3 Internet Programming**

### **2.3.1 HTML**

The World Wide Web is built on web pages. These web pages are created with the Hypertext Markup Language (HTML). “One of the original design goals of HTML was to be device independent.” (Richmond 2002) All computers, regardless of system structure require access to the World Wide Web, HTML needed to be device independent. This means that the language had to follow the same rules on every type of system.

While HTML is widely used to display web pages, its main purpose initially was to provide a way of structuring the data in a page. “So the basic HTML elements specify such things as headings, titles, and paragraphs - but not margins and fonts.” (Richmond 2002) While the structure is defined within an HTML file, the proper display or rendering of the web page was left up to the individual browser. While the original developers of HTML decided it was appropriate to only define structure, as the web became more commercialized, it was important for the image of businesses to have more functionality with the design or layout of pages. “...the original HTML offered very little support for layout and presentation, so the demand grew for extensions. Various browser manufacturers have introduced new HTML elements oriented towards presentation issues - notably Netscape - and some of these have been adopted into the HTML standards proposals.” (Richmond 2002) Style sheets are currently used to put display information into a web document.

The structural elements of an HTML document are elements. Elements are defined within tags. Tags are defined by left and right angle brackets. The first character string in the tag is the name of the tag and the name/value pairs which follow are attributes of the tag. "HTML documents are free-format - you can use spaces and tabs any way you like, and break lines anywhere. White space and line breaks will not affect the document appearance in a browser except when used inside certain special tags which we'll describe later." (Richmond 2002) The tags in an HTML document are properly nested, however, spaces, tabs, new lines, and character case does not affect the resulting display. HTML allows one to publish web documents, create links to related works from your document, include graphics and multimedia data within ones own document, and link to non-World Wide Web information resources on the Internet.

### **2.3.2 JavaScript**

"JavaScript was designed to provide an easy way for Web authors to create interactive Web pages." (Savetz 1999) While HTML is used to create static web pages, JavaScript is a scripting language designed to be an easy way for people with little or no real programming experience to create dynamic web pages. "Unlike Java, which is meant for experienced programmers with an understanding of C++, JavaScript is a simpler "scripting" language (like dBASE and AppleScript) aimed at those with less programming experience." (Savetz1999) JavaScript, which is embedded in HTML tags, is lightweight programming code which can be used for form validation, specific browser detection, and cookie manipulation. "JavaScript can be used to manage user input as well as to show text, play sounds, display images, or communicate with a plug-in in response to 'events' such as a mouse-click or exiting or entering a Web page." (Savetz1999)

While CGI and Java provide some interactivity to the web, JavaScript was not intended to replace those.

A common misconception is that JavaScript is similar to Java. Similarities can be found in the syntax and in the names of the languages. Otherwise, the two are distinct languages with separate functions and purposes. “JavaScript programs are interpreted and run entirely on the client side. This means fewer hits and less processing time on the server than with Java (where applets are compiled on the server before being executed on the client) or CGI (which requires the server to do the work and rack up hits).” (Savetz) Another major distinction is that Java source code is contained in files separate from the HTML and JavaScript is embedded between script tags within the HTML files.

There are currently many clients that support JavaScript, including Netscape Navigator 2.0x, Netscape Navigator 3.0x, Netscape Navigator 4.0x, Microsoft Internet Explorer 3.0x, and Microsoft Internet Explorer 4.0. Thus, when you write a script and embed it in your site, it will be interpreted by different interpreters with unequal capabilities and features. Unlike server-side scripts over which you have full control of their interpretation, client-side scripts are executed by the user’s browser. (Clark 1997)

While the advantage of this difference is reduced server traffic, because the workload is transferred to the client, there are issues with JavaScript compatibility between browsers.



### 2.3.3 Perl/CGI

Perl is an acronym for Practical Extraction and Report Language. It is a programming language which is in wide use on the World Wide Web in helping to create interactive web pages. The basic syntax of Perl is meant to be easy to understand.

Yet, the open-ended flexibility of Perl offers a seemingly endless possibility – which is where the exhilaration and/or intimidation usually set in. The good news is that you don't need to master Perl to make it useful. Web developers take heart: Perl is simply a hammer, with which you can build a birdhouse or a mansion. And you don't need to be Bob Villa to build a birdhouse. (Weiss 1999)

Beyond the general syntax of Perl, there are so many different possibilities that the language appears ambiguous. At the same time, that is what makes the language flexible and powerful especially in the context of web programming. The basics of the language include if statements, looping statements, functions, and regular expressions. Perl also has many features for generating HTML, receiving arguments from the Web, and setting/reading cookies. However, the possibilities certainly do not cease there.

For web programming, Perl is used in the context of CGI. CGI, which stands for Common Gateway Interface, is a standard by which Perl is used to interact with web pages. The basic concept of CGI is: “the user provides some information on the web page and the browser sends this information to the web server. The web server passes this information to a particular program; this program “does a bunch of stuff” with the information and returns some results to the web server, which passes the results back to the user's browser.” (Wiess 1999) While any programming language can be used with

CGI, Perl is the most common because of its ability for data manipulation and its general open-endedness and flexibility.

## **3 Methodology**

### **3.1 Structure**

The academic curriculum was structured around many activities with the purpose motivating the students. Each activity had a background section explaining everything needed for finishing it correctly and students were allotted plenty of time every day during lab sessions to complete each activity. During these sessions, the students were encouraged to work together in solving the problems and finishing the assignments. This lab period also provided the students with plenty of direct assistance from the teaching staff, when and if needed and offered the opportunity to explore related topics easily by searching online.

By encouraging the students to work together in order to solve problems, they had the freedom to determine their own content and were able to research different areas of what they wanted to learn. By also providing all the content online, the students were able to learn at their own pace. They did not need to refer to anyone for help, although help was readily available, and were able to teach themselves. During this process, the students were allowed to collaborate with other students and were allowed to talk while they were programming in order to better understand the process.

The goal of this structure was to teach the students how to learn productively. They were given background information and asked to complete each assignment during the class period, using the resources available to them. The material for the class was available to everyone when they needed it.

### **3.2 Motivation**

According to Lander, there are three key elements for motivation in a classroom. The first element is that there should be ample encouragement and excitement. This was achieved by giving the students three different levels of learning. Students who already knew the material were able to do the advanced section and learn more from their time. These levels gave all students the encouragement and excitement needed to excel in the class.

The second element was to teach the students the importance of the materials and activities. Each activity had a sufficient amount of background needed to complete the activity. The students were given enough information to be able to complete the activities.

The third element is that the materials being presented should be relevant to what is being taught in the class. The material that the student was asked to learn in each section had its relevance in what the student was asked to perform during the class. Each of the activity referred the student back to the following section which he completed reading.

### **3.3 Active Engagement**

The students were actively engaged by participating in the online labs. By completing labs they used the materials provided and they learned to find information on their own in an independent manner by researching, and going online in order to find better ways of implementing the language they learned. By doing this, the students were able to learn how to research on their own, and learn how to do things that were not taught in a traditional educational setting. Active engagement was a major part of the Frontiers program.

### **3.4 Collaboration and Interdependence**

Students worked together and helped each other find solutions to problems faster and easier than they would have been able to do alone. This encouraged interaction between students and offered an additional method to teaching separate from the traditional 'teacher teaches student' strategy. This enabled the students were able to learn from each other. As discussed in the Background Research in Section 2.1.4, education is a social process; hence it should be treated as that.

### **3.5 Content**

The content of the program was given to each student. Links were provided to related materials which would encourage students to expand on what they were given through independent research. The student were asked to, in the advanced section, go out and look for other websites that may help them build a better webpage. The content that was offered to them was also very important to the student to learn. This is the reason that the student had to complete each activity to show that they knew how to do that part of class. This also ensured that the student learned the content that was given to him/her, and also learned more by doing his/her own research over the World Wide Web.

### **3.6 Assessment**

The pre-test and the post-test are the same exact test. It is multiple-choice and includes one question out of each section from the labs. The advantage to giving students the same exact test before and after the course is in measuring what the students learned. The students were asked questions from all of the web programming languages they were going to learn during the two-week program. In each section, they had approximately three questions, each of these questions were exactly setup the way the labs were setup

up. The first question was the least challenging, the second was a bit more difficult, and the third one was from the advanced section.

## **4 Results and Analysis**

The Frontiers Program was structured to run over a span of two weeks. During the first week, students worked alone or collaboratively to cover the background material and complete the activities associated with each lab. All of the activities were done in the student's public\_html directory, so we were easily able to check on their progress during the first week.

They were able to get help from other students, the teaching assistant, the professor, or the undergraduate student assistants. The second week of the program was set up to be a group-oriented project. All of the students were put into groups of two or three. We polled the students to see who they wanted to work with and which projects they preferred to work on. There were ten projects to choose from. Three projects were done using Perl/CGI, three using java, and four were using java applets. The first week helped the students develop their skills and helped them learn more about web programming which, in turn, helped them when working on projects.

The formal assessment results in the instructors' decision-making process include a pre-test, a post-test, four quizzes, and a survey. Other information that is included is the individual students' progress through the four labs.

### **4.1 Week One**

The labs during the first week were divided into three parts: the introductory section, the "more" section, and the advanced section. The students were asked to review the material in each section. Each section had its own activity that the student was to complete and post on their web page. When the student finished one section, they were able to move onto the next section. We organized the class this way because the students

would be able to move at their own pace. Lander says where the student has freedom to determine content; structure can be achieved by the activities used in the class. After the students had completed work on the lab, they were given a written quiz, which included four multiple-choice questions and one short answer question. There would be only one multiple-choice question based on material covered in the optional advanced section. This allowed the students to skip the advanced sections and still pass the quizzes.

#### **4.1.1 General Lab Organization**

The course was divided into four separate labs to be reviewed, not necessarily completed, during the first week of the Frontiers program. The first lab covered the Client/Server model, file management in the UNIX operating system, and the Hypertext Markup Language (HTML). The second lab covered the basics of the JavaScript language. The next lab gives an introduction to Perl and how it is used in CGI. The last lab provides an introduction to the java programming language and covers the basics for creating java applets.

Each student in the Frontiers program has a different background with computer programming. With this in mind, the HTML, JavaScript, Perl/CGI, and java applet labs were divided into three sections. The first section for each topic is an introductory section, which all students were expected to understand regardless of past programming experience. This section generally covers background information or basic concepts. The second section of each lab was meant to provide the students with a functional understanding for the subject. The last section for each topic was the advanced section. Not all students are expected to understand the material covered in this section. This section is an extension on the basics covered in the previous sections. While the



advanced sections are not completely necessary for all students, those who breeze over the previous sections should find it useful if not challenging.

There is sample code provided throughout all of the labs. These sections of code are clearly marked with block quotes, courier font, and red lettering. If there is a full example provided, where the student can simply copy the entire sample into a file, the code is also given a white background. The sample code provides a template for students to manipulate or add to. Most of the sample code is also accompanied with examples, which illustrate exactly what the code is doing.

#### **4.1.2 Lab One: Creating Web Pages**

In the first lab, the Client/Server Model and the file management in UNIX are only given one section each. These sections are both necessary for any advancement in the labs. All students need to understand how the client/server model works and what the necessary UNIX commands are to start applying or testing anything they learn in the labs. The basics of the UNIX directory structure and relative and absolute pathnames were discussed in the UNIX section. After that, commands used in working with files and directories in UNIX are discussed. There are also subsections on getting information using `finger`, accessing man pages, understanding file permissions, and changing file permissions using `chmod`. Activity 1.1 is associated with these sections. It required that students first login to their UNIX accounts. Then they create a directory named “public\_html” in their home directory, and a dummy file in that directory named “index.html”. The permissions for the directory and file would then be set to read, write, and execute permission for the user, and read permission only for group and all. Students would then be instructed to use ‘`ls -l`’ to check those permissions.

The next section is an introduction to HTML, it explains what tags are, how they are implemented, and how they can be applied to creating web pages. The basic tags included were used for text display, creating hyper links, and displaying images. Activity 1.2 asked students to edit their “~public\_html/index.html” files to display information such as a title, a heading, and various hyper links.

The next section, entitled “More HTML”, discusses some of the various tag attributes and how to represent color in hexadecimal form. This section also goes over adding sound and creating ordered, unordered, and definition lists in web pages. Specifically, the body, basefont, font, and img tags and background, bgcolor, text, link, vlink, and alink attributes were explained. These topics cover the basics for adding style to web pages. Activity 1.3 required the students to add style to their web pages using the tags and attributes discussed. Each student had their picture taken so they could post it on their web page.

The final section of lab 1 was “Advanced HTML”, it was not a required section for all students because it would not be necessary in the following labs. The advanced HTML was divided in two subsections. The first dealt with creating tables and manipulating their structure, size, and style. The second subsection, on frames, explained the structure of basic frames and how to use the target and scrolling attributes. At the end of this lab, there was also a list of ‘Good Web Publishing Techniques’. Activity 1.4 had the students format their web page from previous activities using tables.

### **4.1.3 Lab Two: JavaScript**

In previous Frontier programs, the students would be learning object-oriented programming with Java. After JavaScript is included in HTML, it can be run, simply by

bringing up the web page. There is no compiling involved, which removes some complications for students just getting started. There are many snippets or examples of JavaScript code that can be found on the web. Students who complete the material could continue learning outside the course.

In the introductory section of this lab, some background information about JavaScript is given. We explained how JavaScript worked and some things it is commonly used for. There is an example of how to include scripts in an HTML file, and there is a type of “Hello World!” example using an alert box, which is provided, explained, and illustrated. Basic JavaScript syntax was explained and there was another example provided which uses the `document.write()` function. In the associated activity students are asked to create a page, which displays some alert when the page was loaded.

The next section in the JavaScript lab begins with an example, which illustrates how variables were used. Then there were examples of comparing variables and values using relational operators. The next subsection discusses the syntax of the if/else structure in general and in the context of a confirm box. Activity 2.2 requires students to display a confirm box on their web page.

The advanced JavaScript section explains how various events are defined and handled using functions. There were also subsections on using for and while loops and how they can be used to work with arrays. There was also a simple example, which explains how you can set and read cookies using JavaScript. The activity for this section asked students to write an HTML page which used JavaScript to work with text boxes and arrays.

#### **4.1.4 Lab Three: Perl/CGI**

The textbook for the course, “The Web Wizard’s Guide to Perl and CGI”, began with the basics of the Perl/CGI development process and then described the details of the Perl language that were useful for Web application programming. Further topics in the textbook include variable lists (arrays), looping statements, hash lists, subroutines, regular expressions, and working with files. There are basic UNIX commands discussed in the appendix and a quick reference to built in Perl functions and modules in the back cover.

The Perl/CGI lab was a good introduction to programming for beginners. While Perl is an interpreted programming language which can be used in conjunction with HTML, it will also be an extension on the student web pages. It does not require compilation and can be as simple as cut and paste.

The first section of the Perl/CGI lab does not cover any CGI. It simply covers various aspects of the Perl language. It explained how to work with variables, arrays, and hashes. There were also descriptions for commonly used functions, which were built into the Perl language. The activity required students to write a Perl script, which prompts the user for a name, course name, and four quiz scores. The script should then calculate and display the students average.

The next section, “More Perl/CGI”, explains how a Perl script can interpret form data passed by an HTML page. It explained how to use the `QUERY_STRING` and `CONTENT_LENGTH` environment variables. There was also a description on how a CGI program can send a response back to the user. The corresponding activity asked students to create an HTML file with the following text boxes: first name, middle initial, last name, street address, city, state, zip code, and email address. After a user submits the

form, the data should be sent to a CGI script, which simply formats the data using and HTML list and sends it back to the user browser.

The advanced Perl/CGI section gave a brief explanation of pattern matching and decoding form data. There was a reference to the book where students can find functions which make it easy to work with cookies using Perl/CGI. There is also a program described which automatically sends email. Activity 3.3 was divided into two parts. The first part asks the students to modify their previous program to also email the form data to the owner of the site (themselves). The next section of the activity was taken out of the textbook. It requires cookies to save the users preference for background color. When a user selected a color and then comes back to the page, the background color should take affect.

#### **4.1.5 Lab Four: Java Applets**

Lab four taught students how to work with Java applets. This subject was chosen because it fits in with the theme of “Web Programming” and because it serves as a very good introduction to ‘Object-Oriented Programming’. There are plenty of examples provided in the lab, which really helps beginning programmers get through the material. All of the activities in this lab ask the students to progressively build a working Tic-Tac-Toe applet.

The introduction to Java Applets section explained the basic applets structure: import statements, class declarations, init functions, and paint functions. There was a subsection at the end, which described how to compile a java program and embed into an HTML document. The activity for this section required students to set up the game using the init function and draw the initial board using the paint function.

The “More Java Applets” section described how to work with java variables, arrays, and two-dimensional arrays. Detailed examples were provided in each of these subsections. The activity for this section asks students to first declare a three by three array of integers representing blocks in the Tic-Tac-Toe board. The init function should set the values of that array to equal 0. Then it should set the value for the center square to 1 and the value for the upper left square to 2. The paint function should use two for loops to interpret those values and place X’s and O’s accordingly.

In the advanced section there were subsections on event handling, specifically the mouseUp function, logical operators, adding graphics, and adding sound. The activity associated with this section required students to declare a variable named “curPlayer” to keep track of whose turn it is. There should also be a mouseUp function defined which interprets the mouse position into squares on the board, assigns values accordingly, and calls the repaint function.

## **4.2 Week Two**

The second week for the Frontiers Program consisted of a Qualifying Project. They used all the material learned in the first week to do their major project. This project consisted of such things as an e-commerce website, a stock ticker program, and even a web browser. The students were asked to work in groups of 2, 3 or 4 and were split up into whichever project they wanted to do.

The reason for these projects was to show the students a “real world” experience in the Web Development field. They were required to work in groups, like they do in the real world, in order to complete the project. They had to set goals, and setup a schedule of things in which they would work on. This project is similar to the two projects, MQP

and IQP which WPI students are required to finish in order to graduate. In these two projects, WPI students have to work in groups to achieve their goals.

### 4.3 Pre-Test

Full results available in Appendices A.: The Pre-Test basically showed us what the students were capable of doing. The average score in each section are as follows in Figure 4.1.P:

Section (points possible)	Average score
HTML (5)	3.2
JavaScript (3)	1.667
Perl (4)	1.2778
Java (3)	0.444
<i>Total Pre-Test Average (15)</i>	6.0556

Figure 4.1.P

We were able to see that most students had some experience with HTML and JavaScript. From the test scores of the other two sections, Perl and Java, we are able to lead say that mostly no student in the class had experience of it.

The highest grade that was recorded on the pre-test was an eleven out of fifteen and the lowest recorded on was a three out of fifteen. Student 4.2 did not take the test.

Out of the eighteen students in the class, three students started the programming section of each lab with javaBOTL. The rest of the students did the Perl and Java Labs as they were given.

## **4.4 Quizzes**

There were four quizzes that were offered to the students. All of the students did not finish all the labs; hence all of the students did not take all the quizzes. The quiz grades and the students' progress through the activities are available in the Appendices B, C, D, and E.

### **4.3.1 Quiz One – HTML**

The HTML section of the program contained a quiz and also four activities. The students were asked to do all the activities they were able to do. All of the students were required to take the HTML quiz. Eleven out of the eighteen students finished all four activities in the section. The average for the quiz was 93.3. Four out of the eighteen students got the advanced question wrong on the quiz. This question was the one that most students had problems with.

### **4.3.2 Quiz Two – JavaScript**

All of the students were required to attempt this section of the program. This section contained a quiz and three activities. The average on the quiz for this section was 91.67. The problem that most students had a difficulty in was number three on the quiz. Six out of the eighteen students in the class finished all the activities available for this lab.

### **4.3.3 Quiz Three – Perl**

The students were not required to do this lab or take the quiz. Some of the students jumped from this to either Java or, some of the less experienced students, to javaBOTL. Three out of the eighteen people did not attempt to take the quiz. The quiz average for the rest of the class was 83.67. The problem that most students on had on the quiz was with number one. Nine out of the eighteen people completed all of the activities



for this section. One of the students took the quiz without doing any of the activities and got a perfect score. One student did not attempt either the lab or the quiz. Two of the students attempted the lab, but did not take the quiz.

#### **4.3.4 Quiz Four – Java**

The students were not required to do this lab or the quiz also. Ten out of the eighteen students attempted this section also took the quiz. The average for the quiz was 86.5. There wasn't one problem that gave a major problem to everyone, it was a wide variety of questions in the quiz that people had got wrong on. Seven out of those ten people who attempted this section, achieved what they were supposed to achieve, which was a working tic-tac-toe game. The other three students only finished the first two sections of the lab.

#### **4.5 Post-Test**

The post-test was given the last day of classes. The post-test was given too see if the students had learned anything through the program that we had setup. The following chart, Figure 4.2.P, shows the total for each section, and the average score for the pre and the post test.

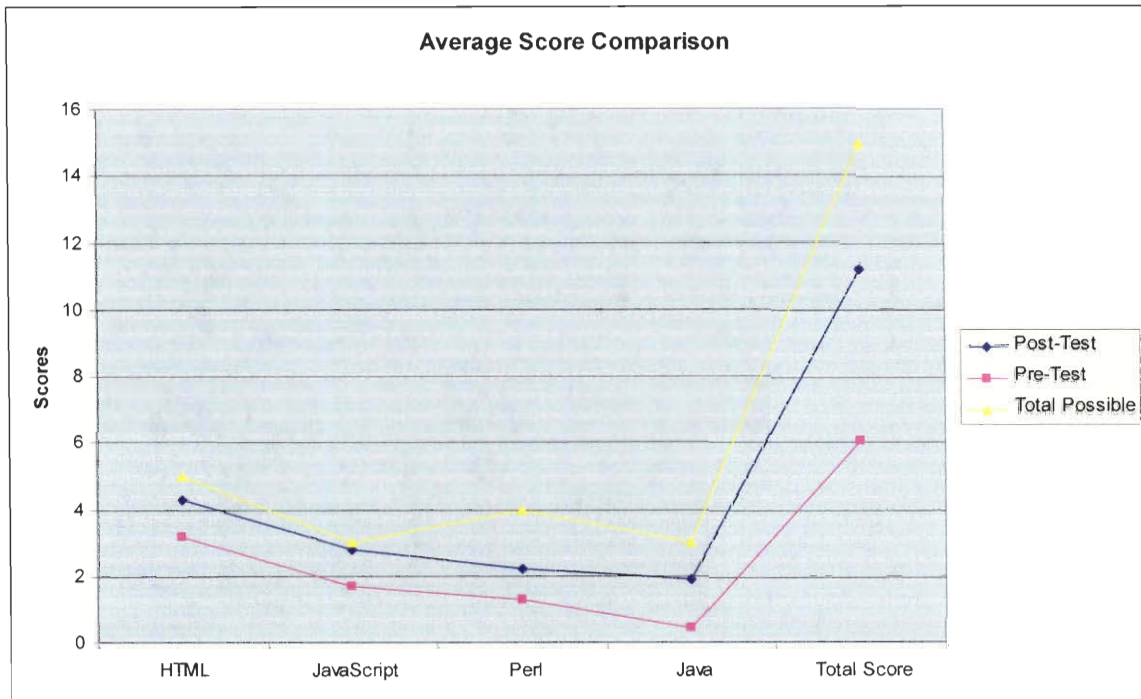


Figure 4.2.P

As we are able to see from the chart, there is a substantial gain in the total score for test. The average score for the test increased fifty-four percent. Out of fifteen points, the lowest grade in the post-test was a nine and the highest grade was a fourteen.

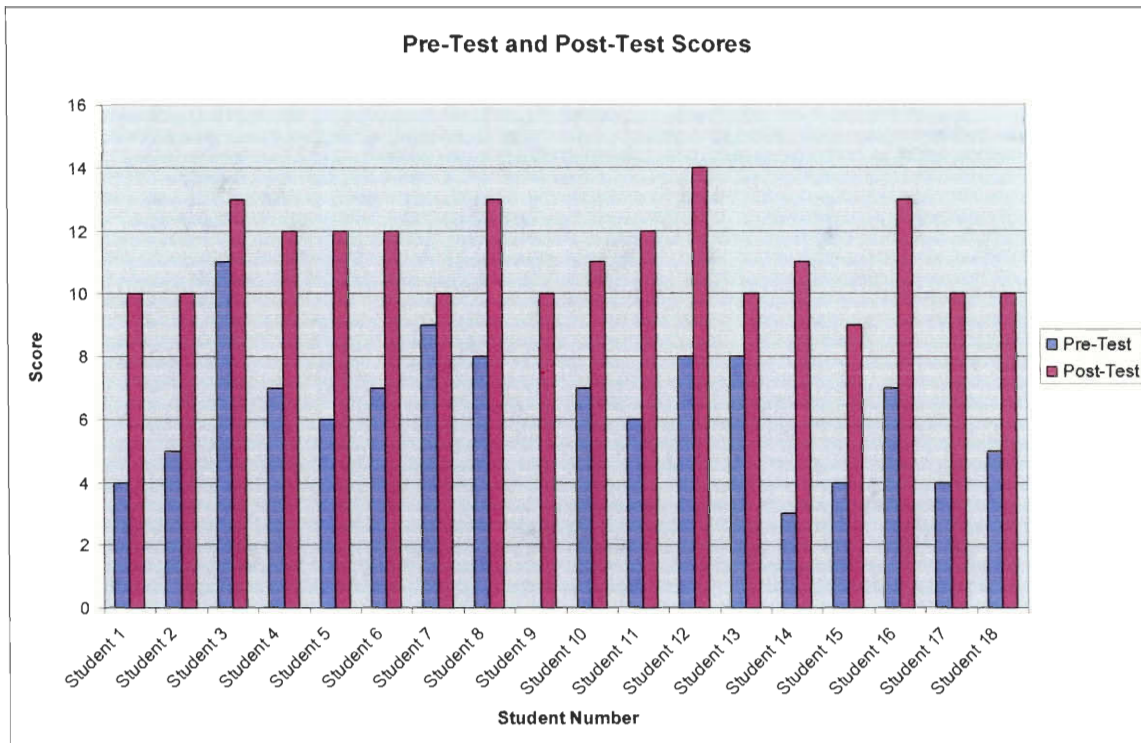


Figure 4.3.P

#### 4.6 Individual Student Progress

The most important assignment throughout the two week educational process was the Frontiers Qualifying Project (FQP), which would be completed during the second half of the program. There was an assortment of ten different projects for the students to decide upon. We anticipated each student spending a majority of their time during the first week working in the Perl or java applet labs. For this reason alone all of the possible FQPs were concerned with either Perl or java programming. After completing the assignments based on the labs in the Frontiers program; the students were then individually polled regarding which of the FQPs they would be interested in working on. They were also asked to specify which of their piers they would prefer to work with and which they would prefer not to work with. Taking this information into consideration, we then divided the students into project groups each consisting of two to four members.

Six of the ten possible projects were selected. The following individual student information is organized according to which project each student was directly involved with.

#### **4.6.1 Stock Ticker Project**

In this assignment students would write a CGI script to look up stock data. This project provides students with exposure to real world HTML on the web and experience with writing Perl scripts in the context of CGI and to work with HTML forms. Students would create a web page where a user can enter stock symbols in a text box and select various stock information options to be displayed such as company name, price, change, volume, average volume, day's range, fifty-two week range, and market capitalization.

##### **4.6.1.1 Student 1.1**

This particular student demonstrated some knowledge of HTML on the pre-test, however lacked knowledge in other sections. During the first week, he completed all of the HTML activities, most of the JavaScript, and the introduction to Perl. This student worked on a javaBOTL program rather than the java applets.

##### **4.6.1.2 Student 1.2**

Student 1.2 knew some HTML and some JavaScript; this was proven on the pre-test exam. He was a dedicated student who spent a bulk of time completing every activity including the advanced sections; however he was unable to complete the advanced java applets section due to time constraints. This student carried much of the weight of his group project. He worked with little help from his group members on completing the Perl programming assignment.

#### **4.6.1.3 Student 1.3**

This student failed to show a prospective score on the pre-test. He only completed two activities in the HTML section and two activities in the JavaScript section. He scored well on the first two tests, and poorly on the Perl section. He did not take the fourth quiz. When working in the stock ticker group, this student spent all his time on the HTML pages and did not help with any of the Perl programming.

#### **4.6.2 Web Browser Project**

The web browser project involves Perl/CGI, exposure to real world HTML, working with HTML forms, and maintaining state on the web using cookies. For this assignment, students would create their own web browser. First students collaborated several ideas for a catchy and marketable name for the browser. After completing that step in the process they would create an HTML page containing two frames. The first frame would contain a form where a user could enter a web URL. On form submission, the target of the provided URL would be displayed in the second frame. Cookies would be used to implement various navigational buttons such as ‘back,’ ‘forward,’ and ‘home.’

##### **4.6.2.1 Student 2.1**

This student came into the program with some knowledge of HTML. He completed all activities in the HTML lab, most of the JavaScript, and a good number of the Perl activities. He was very at ease with asking questions when he was mystified with a problem or when he was just curious about subject matter beyond what was being covered. He really enjoyed programming in Perl and was unable to get to the java applet lab. He only had one project partner and they worked well together.

#### **4.6.2.2 Student 2.2**

Student 2.2 completed all of the HTML and Perl activities, most of the JavaScript activities, and none of the java applets activities. He preferred not to ask for help when he had problems but was generally able to work through them alone. He worked well with his project partner and scored well on the post-test.

#### **4.6.3 E-Commerce Site**

The goal of this project was to create an e-commerce site which would accept orders from buyers over the web. The information provided by the user concerning purchases would be displayed as a dynamically created shopping cart by a Perl/CGI script. The application would originate with a home page that has buttons leading to the department pages as well as the shopping cart page. The department page would provide a name and price for each product and a text box where the user can specify the desired quantity of each distinct product. The shopping cart page would provide a list of selected products and buttons to finalize the purchase, return to the home page, and reset the current shopping cart. On order finalization, the application would provide a summary of purchases and a total price for the order.

##### **4.6.3.1 Student 3.1**

This student completed all of the HTML and Perl activities, most of the JavaScript activities, and none of the java applet activities. He received a perfect score on the HTML, JavaScript and Perl quizzes. He did not take the java applet quiz. Due to the vast knowledge that the student encompassed he was able to contribute a great deal of assistance to his group members for the completion of their project.

#### **4.6.3.2 Student 3.2**

Student 3.2 scored well on the pre-test and had some background information in HTML and java programming. This student would ask questions when he was unsure of the material at hand. Therefore he moved through all four of the labs during the first week. He also scored well on all of the quizzes. Overall his contribution, similar to his partners was impressive, his previous background knowledge allowed him to ascertain the best possible techniques to efficiently accomplish the project.

#### **4.6.3.3 Student 3.3**

This student had some experience with HTML prior to the Frontiers program. He completed most of the HTML and JavaScript activities, all of the Perl activities, and none of the java applet lab. He scored well on the first three quizzes but did not take the java applet quiz. This student's knowledge was useful considering the work and quizzes; however he moved far slower than his other group members and may have held the group behind.

#### **4.6.4 Cryptography Project – Cipher text Decryption**

The goal of the cryptography project was to write a java program that decrypts a cipher text file encrypted by a public key. A cipher text file would contain an encrypted message. A public key consists of a very large number:  $z$ , and another smaller number called the encryption key:  $n$ . to convert the cipher text, the students would need a private key consisting of three numbers:  $p$ ,  $q$ , and  $s$ .  $p$  and  $q$  are both prime numbers with the property:  $p * q = z$ .  $s$  is called the decryption key.

##### **4.6.4.1 Student 4.1**

This student scored very well on the pretest and had some obvious knowledge of HTML, JavaScript, and Perl. He completed all the activities for all the labs except for the advanced JavaScript section because he was encouraged to move on. This student's knowledge did not move the group towards any extensive conquests; however he did work diligently to complete the tasks at hand.

#### **4.6.4.2 Student 4.2**

Student 4.2 began the course with no programming experience. He was considering electrical engineering as well as computer science to as his major. He moved through the material on the course web page very slowly and even took notes, unlike many of the other students. He completed most of the HTML and JavaScript activities and received a perfect score on the corresponding quizzes. Rather than moving onto the Perl and java applet labs, this student was instructed to begin object-oriented programming with the RoBOTL programming language. This prepared him for an FQP in the java programming language which helped his score in the java applet section of the post-test despite not completing any of the original java applet lab material. Since the students took a great deal of time to understand the material he was able to contribute to the project extensively. His industrious attitude enabled him to do so well.

#### **4.6.4.3 Student 4.3**

This student completed all of the activities for the labs, including advanced sections, and he scored well on the four quizzes. He was an active member within the group and may have been the anchor between the students. Despite this progress, he did not score well on the post-test.



#### **4.6.4.4 Student 4.4**

Student 4.4 began the program with possibly some knowledge of HTML. He completed every single activity and scored very well on all of the quizzes. He was very comfortable asking questions and did some extra work with graphics in the Movie Lab. He and student 4.3 they were able to provide most of the informative background on the project.

#### **4.6.5 Meet the Sticks**

The assignment for the 'Meet the Sticks' project was to develop an applet which would draw simple stick people. The first method utilized would be called 'drawPerson' and it would require the ability to draw a stick figure. The next method, called drawFamily, would draw two adults and two children by calling the drawPerson method four times and providing appropriate heights and locations. Additionally, students would add a width parameter to the drawFamily method and provide scrollbars.

#### **4.6.5.1 Student 5.1**

After completing most of the HTML and JavaScript and all of the Perl material, this student was advised to begin object-oriented programming in the RoBOTL programming language. This worked well because he then chose an FQP which involved java applets, and did very well on the project itself.

#### **4.6.5.2 Student 5.2**

Student 5.2 began the course with some background in HTML. She completed all of the activities in the HTML, JavaScript, and Perl labs. Rather than moving directly to the fourth lab (java applets), she was directed to start with the RoBOTL programming

language. Although she did not complete any of the java applet material, she chose a java applet related FQP because of her experience in RoBOTL and was successful.

#### **4.6.5.3 Student 5.3**

This student did most of the activities for the HTML, JavaScript, and Perl labs. He was more interested in getting to the java programming. He did all of the java applet activities. He also did most of the work on his java applet project.

#### **4.6.6 Create a Fractal Interface**

In the fractal interface project, students would learn about the creation of fractals, and how to create a series of web-accessible tools to inspect and explore fractals using java applets.

##### **4.6.6.1 Student 6.1**

On the pretest, this student scored well in the HTML and JavaScript sections. He went through the material for the labs and completed all of the activities very quickly. He skipped over the last Perl activity to begin java programming sooner. After completing the course material early, this student began learning about XML and then scheme. His knowledge and ability to move quickly through the material was beneficial to the group

##### **4.6.6.2 Student 6.2**

Student 6.2 began the class with some knowledge of HTML and possibly java. He was very anxious to begin programming with java. He did the minimum number of activities for the HTML, JavaScript, and Perl labs. While he completed the java applets

lab, he did not spend much time in the lab doing actual work, but was in turn successful in contributing to the project as a whole.

#### **4.6.6.3 Student 6.3**

Student 6.3 had some background with HTML. He completed most of the activities in all four labs, omitting each of the advanced sections. This student was able to move on without completing the advanced sections because that material was not included in the quizzes and tests. This student scored well comparatively and showed substantial improvement between the pre-test and the post-test. His gradual understanding of the material was beneficial to the group because it not only allowed him to contribute but to continue to learn in the process.

## **5 Conclusion**

### **5.1 Summary**

Technological advancements are rapidly shaping today's society. Changes are being made in all aspects of everyday life, including the area of education. With the expansion of the Internet, existing teaching methods have been enhanced and new techniques have been conceived. Education and the Internet are equivalent in that the primary purpose of both is to share information.

By creating an online learning environment, the team was able to structure the course material to fit the Frontiers program. Students were presented with activities to motivate them and background material to help them succeed. They were also encouraged to work together in solving problems. If a student became interested in a particular subject, the class structure would allow him/her to research on their own—outside of the chosen course material. The self-paced aspect of the online program compensated for the individual students' varied computer programming backgrounds. Traditional lecturing would have made this compensation impossible.

During the first week of the course, all students sufficiently progressed through the HTML and JavaScript material. Each spent a majority of his/her time learning to program with Perl/CGI or java, depending on their previous experience and personal preference. These experiences provided the programming background necessary for each of the Frontiers Qualifying Project groups to succeed during the second week.

One of the major flaws in this kind of a program is that all students are not as motivated as others. These students either did not do any of the work, or just sat in front of the computer, searching on the web. We saw this happen to a few students during

class. These students did not finish the activities that they were supposed to or even reach to the point where they should have after the first week of class was over. This was one of the biggest flaws in the program.

Even though the program was not flawless, we saw a considerable number of students learn a lot during the two weeks of the program. We are able to see this because of the pre-test and the post-test scores. There is substantial amount of change in these recorded scores. We also saw that mostly all students were actively involved in their Frontiers Qualifying Project. Everyone was involved and wanted to finish their project to the best of their abilities. This is where our other mistake came in. Some of the projects that were offered to the students were very difficult to finish in the time provided to them. A couple of these groups did not finish their projects because they were not given sufficient amount of time to do so.

## ***5.2 Future Work***

While having plenty of class time for questions and collaboration was effective, it would have been more helpful to use the discussion board that we had available for the class. The team should have encouraged the students in the class to use this tool. This would have made some of the common problems, which individual students were experiencing, more easily addressable for the teaching staff.

The quizzes for each lab were in multiple choice format. Because the program was self paced and each student took the quizzes when they decided they were ready, it became a substantial task to administer the quizzes, correct them, record grades, and return them to the students. All of this had to be done quickly because, students would await their quiz results before continuing to the next lab. This process could have been

conducted much more efficiently if the quizzes had been automated using some web based quizzing system. We believe that the quizzing system that we had available should have been used. We encourage this for the next program. This would have greatly improved class organization.

Many of the more advanced students had previously been exposed to HTML and JavaScript and therefore breezed over the first two labs. In a few cases, it was important especially when working on the FQPs for students to have a better understanding of these topics. A possible solution would be to organize the course material so that each of the topics would be covered a little bit at a time. An example would be to cover basic UNIX commands, then an introduction to HTML, then an introduction to JavaScript, and then an introduction to Perl/CGI. In this case, topics in the HTML section would only provide enough information for students to succeed in the introductory JavaScript or Perl/CGI labs. This format would reduce the amount of time for all students to begin learning the more exciting Perl/CGI or java programming and encourage them to learn only what they primarily need in the beginning HTML sections without skipping over it.

One of our primary objectives was to keep the students learning for the duration of the course regardless of their previous web programming experiences. There were a few cases where students progressed through all of the course material. Future extensions on this material may include topics regarding the current version of HTML, cascading style sheets (CSS), Dynamic HTML (DHTML), or PHP (recursive acronym for "PHP: Hypertext Preprocessor").

## References

Ausserhofer, Andreas. *Web-Based Teaching and Learning: A Panacea?*. IEEE Communications Magazine. March 1999.

Bakken, S. S., A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lerdorf, A. Zmievski, J. Ahto. *PHP Manual*, PHP Documentation Group, (April 2002), <http://www.php.net/manual/en/>.

A manual for using PHP: Hypertext Preprocessor, covers getting started with PHP, language basics, major features, and a function reference

Bradenbaugh, Jerry. *JavaScript Application Cookbook*, O'Reilly & Associates, (September 1999).

A source for well documented JavaScript applications

Brusilovsky, P., et al. *A Tool for Developing Adaptive Electronic Textbooks on WWW*, (October 1996) <http://www.contrib.andrew.cmu.edu/~plb/WebNet96.html>

A description of an approach for developing adaptive electronic textbooks. The goal is to design world wide web based applications which help the user learn at their own pace without the assistance of a human teacher.

Clark, Scott, Dan Ragle, Andy King. *Browser Compatibility*, 1997, <http://www.webreference.com/js/column6/index.html>.

Eckel, Bruce. *Thinking in Java*, 2<sup>nd</sup> Edition, Prentice Hall, Inc., Upper Saddle River, NJ, 2000.

A complete introduction to object oriented programming in Java, includes java applets.

Flanagan, David. *JavaScript: The Definitive Guide*, ? Edition, O'Reilly & Associates, (June 1998)

A guide to learning to program with JavaScript

Frontiers. (January 2002). <http://www.wpi.edu/Admin/AO/Frontiers/>.

Information on the frontiers program at WPI.

Gibson, Elizabeth J., et al. *A Comparative Analysis of Web-Based Testing and Evaluation Systems*. (October 1999), <http://renoir.csc.ncsu.edu/MRA/Reports/WebBasedTesting.html>.

This is an assessment of four web based testing and evaluation systems; Mkleesson, Eval, Tutorial Gateway, and OLAA. The six main bases of criteria are testing

functionality, tracking capabilities, grading capabilities, automatic tutorial building capabilities, implementation issues, and security issues.

Gundavaram, Shishir. *CGI Programming on the World Wide Web*.

A guide for CGI programming, includes input, output, forms, server side includes, and cookies.

Harper, Barry. *Creating Motivating Interactive Learning Environments: a Constructive View*. 1997.

<http://www.curtin.edu.au/conference/ascilite97/papers/Harper/Harper.html>.

Lander, Denis. *Online Teaching: Educational Considerations*,

<http://homepages.eu.rmit.edu.au/resdl/teaching3.html>. (1997)

Lash, David A. *The Web Wizard's Guide to Perl and CGI*, Addison Wesley, Boston, (2002).

This is a book designed to teach people with little or no programming experience how to use Perl and CGI. The book covers Perl basics, generating HTML, variables, loops, hash lists, subroutines, regular expressions, files, cookies, and basic UNIX commands.

Lehtinen, Emo. *Computer Supported Collaborative Learning: A Review*.

<http://www.kas.utu.fi/papers/clnet/clnetreport.html>. (1999)

Lemone, Karen A. *Adaptive Web Technologies*,

<http://penguin.wpi.edu:4546/course/cs525/objectives.html>.

This is a distance learning web page which covers many types of web programming such as HTML, JavaScript, Perl, and XML.

Lemone, Karen A. *Frontiers: Computer Science Page*, (2001).

<http://penguin.wpi.edu:4546/course/Frontiers>.

This is the frontiers page that was used for Computer Science in 2000.

Levine, Alan, *Writing HTML – A Tutorial for Creating Web Pages*, Maricopa Center for Learning and Instruction, (June 2000), version 4.5.2,

<http://www.mcli.dist.maricopa.edu/tut/>.

This is a tutorial for helping teachers create learning resources that access information on the Internet. Includes standards on HTML and lessons in HTML, an introduction to JavaScript, introduction to CGI, and an introduction to multimedia on the web.

Newton, Robert. *Pre and Post Testing*. 1999.

[http://www.icbl.hw.ac.uk/lti/cookbook/info\\_pre\\_and\\_post/](http://www.icbl.hw.ac.uk/lti/cookbook/info_pre_and_post/).



Richmond, Alan. *Introduction to HTML*, 2002,  
<http://www.wdvl.com/Authoring/HTML/Intro/>.

Savetz, Kevin M. *Getting the Jump on JavaScript*,  
<http://www.mactech.com/articles/mactech/Vol.12/12.07/JavascriptIntro/>.

Wall, L., T. Christiansen, and R. Schwartz. *Programming Perl*, 2<sup>nd</sup> Edition, O'Reilly & Associates, (September 1996).

A guide to learning Perl

Weiss, Arron. *The Perl You Need to Know*, 2000,  
<http://www.wdvl.com/Authoring/Languages/Perl/PerlfortheWeb/index.html>.

World Wide Web Consortium (W3C), *HTML 4.01 Specification – W3C Recommendation 24 December 1999*, (December 1999), <http://www.w3.org/TR/html401/>.

This is a specification for Hyper Text Markup Language (HTML) version 4.01. This is a recommendation made by the World Wide Web Consortium.

## Appendix A - Background Material

### Lab 1 - Creating Web Pages

- Client-Server Basics
- Basics of the UNIX Operating System
- Introduction to HTML
- More HTML
- Advanced HTML

### Lab 2 - JavaScript

- Introduction to JavaScript
- More Javascript
- Advanced JavaScript

### Lab 3 - Perl/CGI

- Introduction to Perl
- Introduciton to CGI
- Advanced Perl

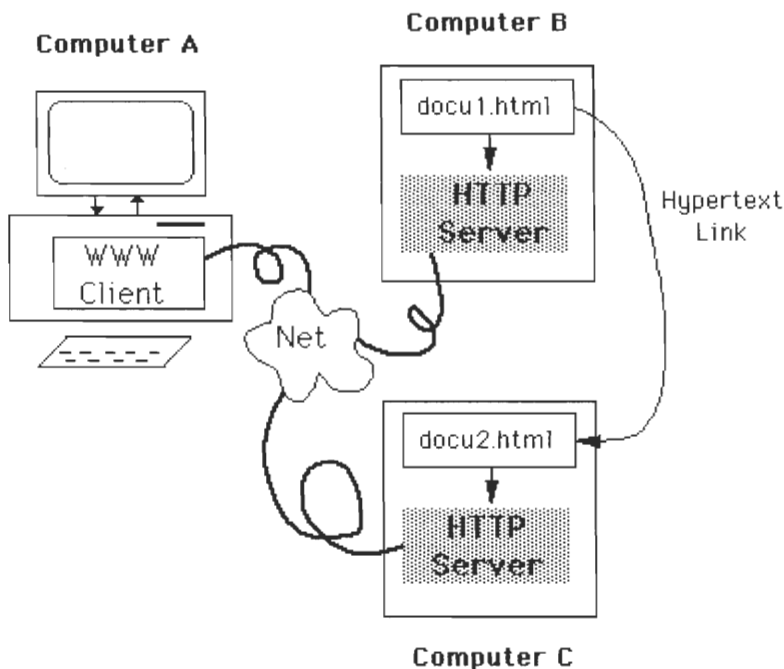
### Lab 4 - Java Applets

- Introduction to Java Applets
- More Java Applets
- Advanced Java Applets

# Client/Server Basics

Web Programming, depends heavily on the *Client/Server Model* of computing.

This document that you are reading was "delivered" to you by a client and a server working together. Your browser, most likely Netscape or Internet Explorer, is the client. This client (e.g., IE) requested this document from a WWW server - the one where this page is stored - if you look in the **Location:** field at the top of the page, you will see the *server* - it is the address after *http://*. When you click on any of the links within a page, the server where these linked pages are stored will "serve" them to your client, IE, which will then display them.



In this picture, by Michael Grobe, at the University of Kansas, the computer you are using is *Computer A*. The computer at the server site where this page is stored, is *Computer B*, and *Computer C* would be the computer that contains any document that you transfer to by clicking on a link.

## HTTP

Hypertext Transfer Protocol is the name of the language that clients and servers use to communicate with each other. HTTP is the underlying protocol of the World Wide Web. It enables us to interconnect and access many different types of documents such as text and graphics and sound, etc. that can reside on any computer on the Internet.

HTTP is a Client/Server protocol: functioning of the WWW is divided between two groups of applications: Web Servers, and Web Clients (often Browsers.)

Web Browsers are the *user* interface to the Web. They allow users to request any document on the Web, and they handle displaying of these documents. The most common type of document is the HTML text file. Pictures, sound, video, applications, and other file formats are also found on the Web.

The program that accepts browser requests, and delivers the data is called a Web Server.

Each communication between a Web browser and a Web server (an HTTP transaction) consists of a Request and a Response.

A Request is always initiated by the browser. The Server just waits for the browser to connect. In the request message, the browser asks for a specific document. The Name of this document can be typed in by the user:

```
http://cs.wpi.edu/~kal
```

or it can be embedded in HTML document tags:

```
<a href="http://cs.wpi.edu/~kal"> KAL's Home Page </a>  
<img src=images/www.gif">  
<form action="http://cs.wpi.edu/cgi-bin/kal/hw.pl"> </form>
```

The Browser will strip the server network name, and put the file name in the request message. Besides the file name, the Request can contain data from the HTML form, or other parameters.

When it receives the Request, the server will check if the requested file exists. If it does, the server checks if it is a CGI program. If the file is a CGI script, the server will run it, and pass its output to the browser. If it is not a CGI script, the complete file will be sent to browser in the response message.

On the client side, after it receives the response, the browser displays the document according to document type.

# Basics of the UNIX Operating System

## Introduction

We will be using the UNIX operating system, a powerful network operating system, or OS. The user interface of a UNIX system is called a *shell*. Shells are what actually respond to what you type and tell the OS what you want it to do.

## Logging In

This module will introduce you to some fundamental ideas and techniques for using the UNIX environment.

We will be using PC's and logging in over a network. One way to do this is to:

1. Go to the start menu
2. Select *Run ...*
3. Type *telnet* when the window for the program appears
4. Under the *Connect* menu, select one of the wpi machines (ask us which they are!) or select *Remote System* and type *wpi.wpi.edu* and select *connect*.
5. Type your User Name and Password when prompted to do so.

## Understanding Directories

The directory structure of the UNIX system is very similar to that of a personal computer. Folders on a personal computer are called directories in UNIX. Directories can contain files and other directories called subdirectories that, in turn, can contain files and subdirectories and so on. The UNIX directory structure is analogous to the trunk and roots of a tree.

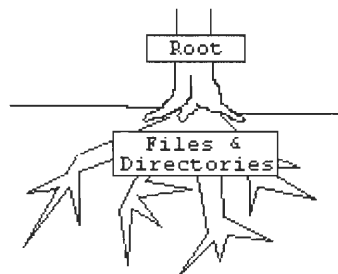


Figure 1: Directory Structure Example

Every UNIX system contains a 'root' directory which is the trunk of the tree. The files and directories branch out from the trunk. Users have their own space set aside on the disk for their files and directories. This space is called the user's 'home directory.' The home directory of a user is represented by their username.

/ This refers to the root directory.

- ~ This refers to the users home directory.
- .
- .. This refers to the super directory of the current working directory, or the directory up one level.
- This refers to the previous directory.

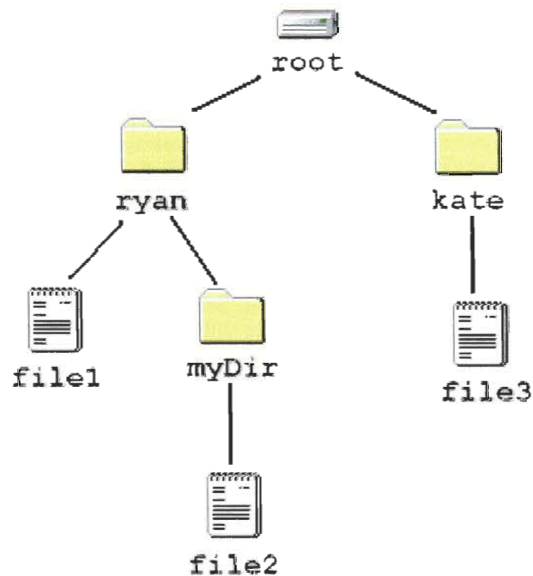


Figure 2: Sample Directory Tree

Figure 2 represents a sample directory tree. **ryan's** home directory contains two items: a subdirectory, **myDir**, and a file, **file1**. Every time **ryan** logs into this computer, he will begin at his home directory. The home directory of a user can be represented by a '~,' (called a tilde). Similarly, the home directory of any other user can be represented by '~username'.

For example, for **ryan**, '~' would represent his own home directory, while **~kate** would represent the home directory of user **kate**.

UNIX is case-sensitive; that means **file1**, **File1**, and **FILE1** all refer to different files. A '/' is used to separate subdirectories. A **pathname** is a sequence of symbols used to identify a file or directory. Every file has a **filename**. The simplest type of pathname is just a filename. If you specify a filename as the pathname, the operating system looks for that file in the current working directory. If the file resides in a different directory, you must tell the operating system how to find that directory by specifying a pathname.

There are 2 types of pathnames. If the pathname starts from your current working directory then it is a **relative pathname**. If it starts from the root directory then it is an **absolute pathname**.

For Figure 2, the pathname of **File1** would be:

If user **ryan** is in his home directory:

```
file1
```

If user **ryan** is outside his home directory:

```
~/file1
```

If user **kate** wants to refer to **ryan's** file:

```
~ryan/file1
```

Absolute pathname for **ryan's** file:

```
/ryan/file1
```

Similarly, the pathname of **file2** would be:

If user **ryan** is in his home directory:

```
myDir/file2
```

If user **ryan** is outside his home directory:

```
~/myDir/file2
```

If user **kate** wants to refer to **ryan's** file:

```
~ryan/myDir/file2
```

Absolute pathname for **ryan's** file:

```
/ryan/myDir/file2
```

## ***Managing Files***

Now that you've seen the UNIX directory structure, we can look at some commands to use. The following is an overview of some useful commands.

### **cd (Change Directory)**

The command **cd** switches the user from the current working directory to a different directory.

For example, in order to change **myDir** in **ryan**'s home directory:

If user **ryan** is in his home directory:

```
cd myDir
```

If user **ryan** is outside his home directory:

```
cd ~/myDir
```

If user **kate** wants to access **ryan**'s files in **myDir**:

```
cd ~ryan/myDir
```

**cd** can also perform the following specific tasks:

```
cd ~
```

takes users to their home directory.

```
cd ..
```

takes a user up one directory. (If **ryan** is in **myDir**, this command will take him to the directory above, which is his home directory.)

```
cd -
```

takes users to the directory they were previously in. (If **kate** were previously in her home directory, but had changed to **ryan**'s (via **cd ~ryan**,) then **cd -** would take her back to her home directory.)

```
cd /
```

takes users to the root directory.

## ls (List)

**ls** displays either the contents of the current directory, if no path is specified, or the contents of the specified directory. For example

```
ls ~ryan
```

would display:

```
myDir/      file1
```

Thus, the contents of the home directory of **ryan** is the subdirectory, **myDir**, and the file **file1**.

**ls** also has some options. One of the most common is **ls -l** which displays the contents of the specified or current directory with more information. For example:

```
ls -l ~ryan
```

might produce the following output:

```
drwx-w---x 2 ryan 100 4096 Jun 14 2000 myDir
-rwx-w---x 1 ryan 100 1450 Jun 14 13:01 file1
```



The first character in each line represents whether the item is a file or a directory (**d** for directory, **-** for file). The next nine characters represent the permissions set to that item (permissions will be discussed below). The next number represents the number of links of the file. (Not discussed here.) Next is the owner of the file, **ryan**. Following this is the userid of the user; **ryan** is user 100. Following this is the size. **file1** is 1450 bytes. A byte is the amount of space needed to store 1 character in a computer. **myDir** is 4096 bytes (all directories are 4096 bytes). Next is the month, followed by the day, followed by either the year or time the item was created. If the file was created during the current year, the time is displayed, if not the year is displayed. And finally, the last column is the full filename.

### **pwd (Present or Print Working Directory)**

This command displays the pathname of the current directory. For example, if **ryan**'s current directory is **myDir**,

```
pwd
would display:
/ryan/MyDir/
```

### **rm (Remove)**

The remove command deletes a file. For example, if **ryan** wishes to delete **file1** he can type:

```
rm file1
```

It is important to remember that once a file is deleted, it is gone forever. Use this command carefully!

### **cp (Copy)**

The copy command, is used to copy a file or group of files. If **kate** wishes to copy **file1** to **myDir** she types:

```
cp file1 ~/myDir/
```

We can use cp to make a copy of a file. If **kate** wants another file just like **file1** in her home directory called **file1save**, she can type:

```
cp file1 file1save
```

### **mv (Move)**

Move performs the exact same function as cp, except that after the file is copied, it is deleted from the original location. For example, if **kate** wishes to move **file1** to **myDir**, (and keep the same name) she can type:

```
mv file1 ~/myDir/
```

If she is in her home directory and wants to move it to **myDir** AND give it a different name, she can type:

```
mv file1 ~/myDir/newFile1
```

An ls of the home directory would no longer show **file1**. **mv** is often used to rename a file.

## mkdir (Make Directory)

The **mkdir** command creates a new directory. For example, if **ryan** wishes to create the directory **newDir** within his home directory he types:

```
mkdir newDir
```

## rmdir (Remove Directory)

The **rmdir** command removes an empty directory (so you have to remove the files in the directory first with **rm**). For example, if **ryan** wishes to delete the empty directory **newDir** he types:

```
rmdir newDir
```

## more

The command **more** displays the contents of files one screen at a time. For example, to display the contents of **file1** one screen at a time use the following command:

```
more file1
```

(Then hit the space bar to get another screenful)

## Getting Information

### who or whoami

**who** and the related command **whoami** give information about users logged into the computer. **whoami** informs you of your username while **who** lists all the people currently logged into the same computer that you are logged into.

For example, for user **ryan**, the command **whoami** displays:

```
ryan
```

Possible output for the command **who**:

```
kate tty0 Jul 14 12:00  
ryan tty1 Jul 14 11:45  
mike tty8 Jul 14 05:45
```

The first field is the username, the second field is the terminal port they are connected to, and the final two fields are the date and time of connection.

### finger

An easy way to find out more about someone is to 'finger' them from your terminal. 'finger'ing someone will tell you if they're currently logged on, or when they last logged off, their home directory, real name, shell, and their mail status. The following is sample output for the command:

```
finger ryan  
Login name: ryan In real life: Ryan Doe  
Phone: 555-1234  
Directory: /usr1/ryan Shell: /sh/tcsh
```

```
No unread mail.  
User      Real Name Idle TTY Host  
ryan      Ryan Doe  0:03 p1  moose.wpi.edu
```

Typing **finger** with no arguments returns a list of all users on the system.

## quota

**quota** displays your disk quota. A disk quota is the amount of disk space that you are allocated. Every once in awhile, you should check to see if you are running out of disk space: The following is sample output from the command **quota**.

```
Disk quotas for user ryan (uid 100):  
Filesystem blocks quota limit  
/ryan      3498   4000  5000
```

## Getting Help

### man command\_name

The man command accesses the online unix manual pages. Because they are hard to read, we often try every other option before resorting to the man pages (ask your friend, try the web, look it up in a book, etc.) In the above, `command_name` is the name of the command you want to look up. If you don't know the name of the command, you can do a keyword search by adding the "-k" option before the `command_name`. If the command you are looking for is anywhere in the man pages, this will find it.

## Understanding File Permissions

Unlike personal computers where usually only one person has access to the information contained within the computer, UNIX allows many people to access the computer and its information. In order to ensure the privacy and security of data and programs on a UNIX system, the file system is designed to allow users access only to that information that they have permission to access.

Every file is stored with a file permission that designates who has the ability to (1) read from, (2) write to, and/or (3) execute that particular file.

A file permission consists of nine items: three groups of the three letters, **r**, **w** and **x** where **r** means **read**, **w** means **write** and **x** means **execute**. For example:

```
-rwx-w---x file1.txt
```

The three groups are Owner, Group and Public. For the above:

Owner: rwx Group: r-- Public: --x

Here, the Owner (usually the creator) of the file has permission to read, write (change) and execute **file1.txt**. We won't be using groups, but here the Group has permission to Read (look at) the file, but not to Write (change) or Execute the file. And the final three

characters represent the level of access that the Public (sometimes called the World) has to the file.

Therefore, in our example the Owner of the file has permission to **read**, **write**, and **execute** the file. The Group has the ability only to **read** from the file, and the Public has the ability only to **execute** the file.

## Changing File Permission

When we create our web pages, we will give the Public permission to access our pages, but not change them. **chmod** is a unix command to change the permissions of a file. there are two methods to use chmod, one is to use the binary/octal method, the other is to use **arguments** to specify how to change the permissions.

Binary/Octal Method:

```
chmod permission filename
```

where permission is a three digit number with each digit an octal number (0 - 7). The first digit represents the Owner's permissions, the second represents the Group's permissions, and the third represents the Public's permissions. Each digit is a combination of the three possible permissions: **read**, **write**, and **execute**. We have to look at each octal digit in binary (base 2) to understand the meaning. Let's do this with the example

```
chmod 755 file1.txt
```

In binary, 755 is 111 101 101 (spaces for clarity). Each group of three binary digits represents the triple **rxw**. The first group, 111, means that the owner can **read**, **write** and **execute** **file1.txt**. The next group of three, 101 indicates that the Group can **read** and **execute**, but not **write** (change) **file1.txt**. The third group, also 101 indicates that the Public can **read** and **execute**, but not **write** to **file1.txt**. Therefore, the value for a permission of Owner to **read**, **write**, and **execute**, Group **read**, and Public **read**, (**rxw r-- r--**) would be 744; the value for a permission of Owner to **read**, **write**, and **execute**, Group to **read**, **write**, and **execute**, and Public none (**rxw rxw ---**) would be 770; and the value for permission for all 3 groups to perform all 3 operations (**rxw rxw rxw**) would be 777.

The other syntax for this command is:

```
chmod who[+/-]permissions file
```

**who** has to be a single letter signifying which set of permissions to affect: user, group, or other. The codes are as follows:

```
u = user
g = group
o = other
a = all
```

The [+/-] designates whether to allow or disallow the following permissions. The permissions, like octal mode, are for reading, writing, and executing. The codes for these are:

```
r = read
w = write
```

```
x = execute
```

There can even be more than one argument to specify how to set all the permissions. These arguments must be separated by a comma:

```
chmod who[+/-]permissions,who[+/-]permissions file
```

If I have a file named `foo`, and I want to make sure everyone has permission to read it, I would execute the following command:

```
chmod a+r foo
```

This means, for the file `foo`, I want to add the read permission to all the sets of permissions. Suppose I want to give everyone read and write, and execute permission (this means anyone can erase it too!). The command would look like:

```
chmod a+rwX foo
```

Now, suppose I want to give everyone read permission, but only give myself write and execute permissions, I would type:

```
chmod a+r,u+wx foo
```

The advantages of using this method over the octal method is that it is easier to understand, and you can set one permission regardless of the other permissions. The disadvantage is that a complicated command takes longer to type and think out (once you know the octal codes, the octal method is almost always faster, but if you forget them, the letter method is usually easier to understand). The best way to learn how to use `chmod` is to use it. Try the examples in your unix account. You can create a dummy file named `foo` by typing:

```
touch foo
```

you can check the permissions on `foo` by typing:

```
ls -l foo
```

# Introduction to HTML

## What is HTML?

Hypertext markup language, or HTML, is the language of the World Wide Web. Every page found on the World Wide Web, from [USAToday](#) to [WPI](#), is written in HTML. HTML is a way of describing to the browser, such as Netscape Navigator, Microsoft Internet Explorer, or Mosaic, how to display and format text and images on a page. It consists of a series of commands, called *tags*, which begin with a < and then end with a >. Tags tell the browser what to do with the text or images associated with it.

## How HTML Tags Work

An HTML tag is made up of a < followed by the command and ending with a >. Inside the <> is the name of the command, the **tagname**. Some tags also take **attributes**, which gives the tag some specific information, or **value**. Therefore, the basic structure of a tag is:

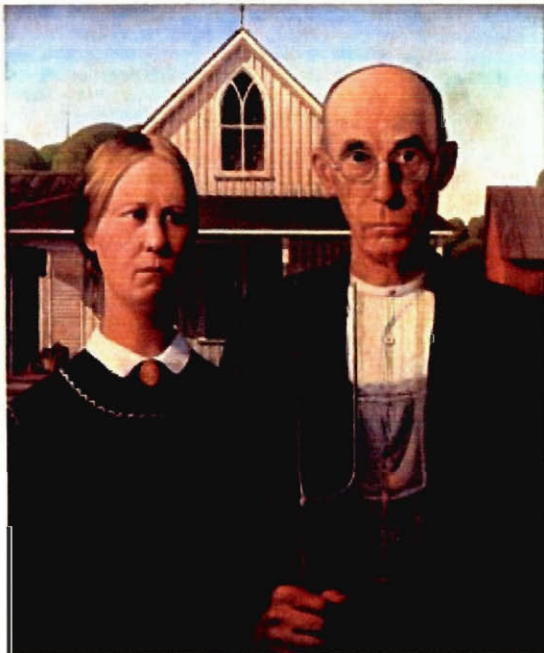
```
<tagname attribute="value" ... >
```

For example, in order to insert an image into a page you would use the <img> tag like this:

```

```

Assuming that there was a file named "americanGothic.png" in the current directory, this tag would result in:



There are two main types of tags. The first type are *standalone tags*, like <img>, in which one tag is all that is needed to perform an action. The second type are *container tags*,

which consist of a starting tag and a ending tag that alters everything between them. The structure of a container tag is:

```
<tagname attribute="value" ... ></tagname>
```

An example of a container tag would be `<u></u>`, which underlines text. The HTML text

```
<u>This text is underlined.</u> This text is not.
```

would produce:

This text is underlined. This text is not.

Underlining isn't always a good idea because links are underlined also, and readers may get confused and not know if the underline is for emphasis or if the underline represents a link.

HTML is a case-insensitive language. For example, `<Center>` is the same as `<center>` which is the same as `<CENTER>` which is the same as `<CeNtEr>`. It is your preference how you choose to write your tags.

## **Basic Tags**

The four most important tags that each page must contain are `<html>`, `<head>`, `<title>`, and `<body>`. Each of these are container tags that appear at the top of each HTML file in the following order:

### **<html>**

This tag tells the browser that this file is an HTML document. This is the first tag that will appear in all HTML documents; since it is a container tag, it will also be the last tag in all HTML documents. Its syntax is simply:

```
<html>
.
.
</html>
```

### **<head> and <title>**

The next necessary part to all web pages is the `<head>` tag. This tag contains the title of the page (and sometimes other information). The title of the page tells the reader what the page is about and will appear at the top of the browser's window. The syntax for these tags is:

```
<head>
<title>Title of Web Page</title>
</head>
```

### **<body>**

The final tag that most web pages need is the `<body>` tag. This tag tells the browser that the following information should be displayed. It can also contain the formatting information for the page (this will be discussed later). The syntax for this tag is:

```
<body>
.
.
</body>
```

Therefore all web pages must have the following structure:

```
<html>
<head>
<title>Title of Web Page</title>
</head>
<body>
.
Body of page
.
</body>
</html>
```

## **End of line tags**

Probably the most basic tags that will be used many times in web pages are `<br>`, break, and `<p>`, paragraph. These tags will end the current line, much the same as a carriage return in a word processor. It is important to note that these tags must be used in place of a carriage return since the browser will not recognize a carriage return in an HTML file. It is also important to note that all whitespace characters such as tabs, spaces, and carriage returns will be combined by the browser into one space.

### **<br>**

The break tag, `<br>`, will end the current line and any text placed after it will continue on the next line. For example:

```
I want to end this line here--> <br> This is the next line.
```

becomes

```
I want to end this line here-->
This is the next line.
```

### **<p>**

The `<p>` tag acts exactly like `<br>`, except that it is to be used at the end of paragraphs. It will end the current line and create a blank line between the paragraphs (this can also be achieved by using two `<br>` tags). For example:

```
...this is the end of this paragraph--> <p> This is the
beginning of the next...
```

becomes



...this is the end of this paragraph-->

This is the beginning of the next...

## **Basic text formatting tags**

It is often necessary to change the format of some text in order to emphasize a point, to set it apart from other text, or to show importance. The following container tags enable the block formatting of text to show emphasis:

### **<b>, <u>, and <i>**

The <b> tag changes the text contained within the tag to **boldface**; the <u> tag underlines the text contained within the tag; and the <i> tag changes the text contained within the tag to *italics*.

```
<b>This text is bold.</b> <br>  
<i>This text is italicized. </i> <br>  
<u>This text is underlined.</u> <br>  
This text is not formatted.
```

becomes

**This text is bold.**

*This text is italicized.*

This text is underlined.

This text is not formatted.

### **<h#>**

The heading tag, <h#>, makes text larger than the surrounding text, as in a newspaper headline. The number in the heading tag tells the browser how big you want the text to be. A number from 1 to 6 may be used, with 1 being the largest and 6 the smallest. The heading tag automatically creates a break after itself, therefore any text placed after the heading will continue on the next line.

```
<h1>This is heading 1</h1>  
<h2>This is heading 2</h2>  
<h3>This is heading 3</h3>  
<h4>This is heading 4</h4>  
<h5>This is heading 5</h5>  
<h6>This is heading 6</h6>
```

becomes

**This is heading 1**

***This is heading 2***

**This is heading 3**

**This is heading 4**

***This is heading 5***

**This is heading 6**

**<center>**

The <center> tag, as its name implies, centers text contained within the tag. This tag can be used to center anything from one character to the entire contents of the page.

```
<center>This text is centered.</center>
<br>
This text is not centered.
```

becomes

This text is centered.

This text is not centered.

## ***Horizontal line tag***

**<hr>**

It is often necessary to separate portions of a page from one another through the use of horizontal line separators (as seen in this page). In order to create these lines, use the <hr> tag. The <hr> tag will automatically create a break before and after itself, therefore any text following it will continue on the next line.

```
... end of one section
<hr>
beginning of next section ...
```

becomes

... end of one section

---

beginning of next section ...

It is possible to change the attributes of the line through the use of the *size*, *width*, and *noshade* attributes.

The *size* attribute changes the thickness of the line. The value of the size is represented in pixels and is called point size.

```
Size 1 line:  
<hr size=1>  
Size 6 line:  
<hr size=6>  
Size 20 line:  
<hr size=20>
```

becomes

Size 1 line:



Size 6 line:



Size 20 line:



The *width* attribute changes the width of the line. The value of the width can either be represented as a pixel value or a percentage of the total width of the page.

```
Width 200 line:  
<hr width=200>  
50% total width of page line:  
<hr width=50%>
```

becomes

Width 200 line:



50% total width of page line:



The *noshade* attribute changes the line to a solid grey line.

```
Size 1 line:  
<hr size=1 noshade>
```

```
Size 6 line:  
<hr size=6 noshade>  
Size 20 line:  
<hr size=20 noshade>
```

becomes

Size 1 line:



Size 6 line:



Size 20 line:



## Links

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

Navigating the World Wide Web is achieved through the use of **links**, or a connection from one web page to another. When a person clicks their mouse on a link, the browser takes them to the page the link represents. To create a link, use the *href* attribute of the `<a>` (anchor) container tag. This attribute is followed by the web address of the page you want to link to. **Note:** Both text and images may be used as a link.

```
This creates a link to <a href="http://www.wpi.edu">WPI's  
web page</a>
```

becomes

This creates a link to [WPI's web page](#)

## Images

```

```

Until now we have dealt only with text, but the real importance of the web is the ability to add images to a page in order to make it more pleasing to a visitor. To add images to a page, use the `<img>` standalone tag. To specify the image to display use the *src* (source) attribute followed by the path and filename of the image. Valid image formats include GIF (.gif), JPEG (.jpg or .jpeg), and PNG (.png).

```
This is the WPI logo: 
```

assuming 'wpi.png' is contained within the same directory as the HTML file, this becomes:



This is the WPI logo:

## More HTML

### **<body>**

As you have learned, the <body> tag contains all of the text and images in a page. However, this is not the only function this tag performs. <body> also controls the color scheme and the background of a page through its attributes.

The *background* attribute can be used to designate an image file as a background to a page. The image would be tiled across the page if it is smaller than the browser window.

The syntax for this tag is:

```
<body background="filename.gif/jpg/png">
```

The following are two methods for specifying color in a web page.

### **#RRGGBB (Red, Green, Blue)**

This method requires you to provide the amounts of red, green and blue contained within the color in hexadecimal form. For example, red would be represented by #FF0000 (red:255, green:0, blue:0) and likewise blue would be represented by #0000FF. This method can be used on all browsers; however, it requires a knowledge of hexadecimals and of color content.

### **Netscape Color Names**

The method involves simply using the name of the color in the attribute. For example, "red" would produce red and "blue" would produce blue. This method is advantageous if you do not know the exact makeup of the color. However, this method only works in Netscape Navigator and Internet Explorer - other browsers may not display the page correctly.

Since the #RRGGBB method works for all browsers, we will be using it exclusively.

The **bgcolor** attribute sets the background color of the page. Its syntax is:

```
<body bgcolor="#RRGGBB">
```

The **text** attribute sets the color of the text on the page. This attribute will affect all the text not specifically colored by the [<font> tag](#). Its syntax is:

```
<body text="#RRGGBB">
```

The **alink**, **vlink**, and **link** attributes set the color of the links contained in the page. **link** refers to the color of the unvisited (unclicked) link. **vlink** refers to the color of the link after it has been visited (clicked). **alink** refers to the color of the link while it is active (while it is being clicked). The syntax of this attribute is:

```
<body link="#RRGGBB" vlink="#RRGGBB" alink="#RRGGBB">
```

## **<basefont>**

The <basefont> tag can be used to set the size and font of all the text contained within a page through the use of the **size** and **face** attributes. This tag usually appears directly after the body tag.

The **size** attribute specifies the size of the font. Valid values range from 1 - 7 (1 being the smallest). For example:

This is font 1

This is font 2

This is font 3

This is font 4

This is font 5

This is font 6

This is font 7

The syntax for this attribute is:

```
<basefont size="[1-7]">
```

The **face** attribute sets the type of font displayed on the page. This attribute will only work correctly on computers that have the specified font installed on them. (i.e., if you want your page to use "Times New Roman" font, in order for a visitor to your page to see the font, they must have "Times New Roman" installed on their system.) This attribute is also currently supported only by Microsoft Internet Explorer and Netscape Navigator. The syntax for this attribute is:

```
<basefont face="font type (i.e., Arial)">
```

## **<font>**

The <font> container tag sets the color, font face, and size of selected text through the **color**, **size**, and **face** attributes.

The **color** attribute sets the color of the text inside the container tag. The syntax of this attribute is:

```
<font color="#RRGGBB">Text to be changed</font>
```

This HTML text:

```
<font color="#FF0000">This is red</font> <br>  
This is normal <br>  
<font color="#0000FF">This is blue</font>
```

would result in:

This is red  
This is normal  
This is blue

The **size** attribute works the same as the size attribute of <basefont> in that it sets the size of the text inside the container tag. However, if the <basefont> tag has been used to set the font size for the page, you may change the size of the text relative to the base font (i.e. +2 would be the base font + 2). The syntax for this attribute is:

```
<font size="(1-7) or (+/- relative value)">
```

These tags:

```
<font size="2">This is set to font 2</font> This is not
```

or

```
<basefont size="3">  
<font size="+2">This is set to font 5</font>
```

would result in:

This is set to font 2 This is not  
This is set to font 5

The **face** attribute sets the type of font of the text inside the container tag. This attribute will only work correctly on computers that have this type of font installed on them. (i.e. if you want your page to use "Times New Roman" font, in order for a visitor to your page to see that font, they must have "Times New Roman" installed on their system. This attribute is also only supported by Netscape Navigator and Internet Explorer. The syntax for this attribute is:

```
<font face="font type">
```

## **More on the <IMG> Tag**

The <img> tag is used to display images on web pages. Moreover, it is possible to control both the appearance of the image and how it reacts to the surrounding text through the use of the **align**, **alt**, **border**, **width**, and **height** attributes.

The **align** attribute controls the way text reacts to the image. The syntax for this tag is:

```

```

The results of each of the align values are:

```
align="left"
```



This text is aligned to the left



This text is aligned to the right

```
align="right"
```



```
align="top"
```



This text is aligned to the top

```
align="middle"
```



This text is aligned to the middle

```
align="bottom"
```



This text is aligned to the bottom

The **alt** attribute is used to display text in place of the image when the image is not, or cannot be, loaded (usually when the user is viewing a page with a text-only browser.) The syntax for this attribute is:

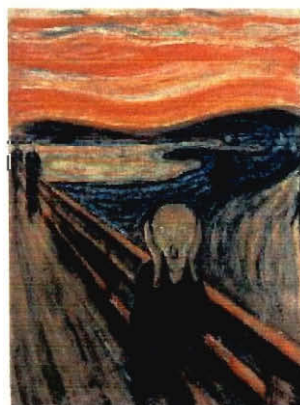
```

```

For example, the following tag

```
Edvard Munch 
```

would result, on a graphical browser, in:



Edvard Munch

and a text-only browser in:

Edvard Munch <The Scream>

The **border** attribute controls the size of the black border surrounding the image. The syntax for this attribute is:

```

```

For example:

```

```



This is a border of size 5.

```

```



This is a border of size 20.

The **width** and **height** attributes set the size of the image in order to enlarge or reduce the size of the image. You can set the exact size of the image by specifying these attributes. For example, with our 140x109 image of Van Gough's "Starry Night",

```
  

```

Becomes:



It is not necessary to specify both **height** and **width** attributes for an image. If you only specify one, the browser will automatically scale the other.

## Lists

HTML allows a page designer to present data in the form of a list. There are three main types of lists: ordered, unordered, and definition lists.

An **ordered list**, as its name implies, is used to display a list of items that are sorted by importance or sequence. An ordered list is created by using the `<ol>...</ol>` container tag. Each item is signified with the `<li>` (List Item) tag. The syntax for this tag is:

```
<ol>
<li>first item
<li>second item
...
</ol>
```

For example:

```
<ol>
<li>Do this first
<li>Do this next
<li>Do this last
</ol>
```

produces:

1. Do this first
2. Do this next
3. Do this last

It is possible to change the type of count marks (e.g., 1 2 3 to a b c, etc...) through the **type** attribute and the starting position through the **start** attribute.

There are five different types of count marks for use with the **type** attribute:

```
type="A"  capital letters (e.g. A, B, C, ...)  
type="a"  lowercase letters (e.g. a, b, c, ...)  
type="I"  capital roman numerals (e.g. I, II, III, ...)  
type="i"  lowercase roman numerals (e.g. i, ii, iii, ...)  
type="1"  default numbers (e.g. 1, 2, 3, ...)
```

The **start** attribute allows the list to be started anywhere within the list. For example, `start="5"` would start the list with number 5 (or whichever count mark is being used).

For example:

```
<ol type=I start=3>
```

```
<li>Open the file menu
<li>Click on 'save as'
<li>Type 'document.txt'
<li>Click 'OK'
</ol>
```

This would result in

- III. Open the file menu
- IV. Click on 'save as'
- V. Type 'document.txt'
- VI. Click 'OK'

An **unordered list** is used to present a list of items marked by bullet points instead of count marks. An unordered list is created by using the `<ul>...</ul>` container tag. As with an ordered list, each list element is denoted by the `<li>` tag. The syntax for an unordered list is the same as that of an ordered list.

An example of an unordered list is:

```
<ul>
<li>Item one
<li>Item two
<li>Item three
</ul>
```

- Item one
- Item two
- Item three

It is possible to change the shape of the bullet from the default disc to a square or a circle through the use of the **type** attribute. The type attribute may be used in the `<ul>` tag, to change all the bullets, or in the `<li>` tag, to change all bullets following that tag. For example:

```
<ul type="square">
<li>Item one
<li type="circle">Item two
<li>Item three
<li type="disc">Item four
</ul>
```

- Item one
- Item two
- Item three
- Item four

The last major type of list that is supported by HTML is the **definition list**. A definition list consists of a list of terms followed by their corresponding definitions indented on the line below. A definition list is created by implementing the `<dl>..</dl>` container tag.

Each term is denoted by the <dt> (Definition list Term) tag and, likewise, each definition is denoted by the <dd> (Definition list Definition) tag. The syntax for this tag is:

```
<dl>
  <dt>Dog
  <dd>(n.) a canine
  <dt>Cat
  <dd>(n.) a feline
</dl>
```

Which becomes:

```
Dog
(n.) a canine
Cat
(n.) a feline
```

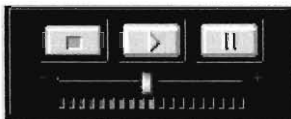
## ***Adding sound to a web page***

It is possible to add not only images, but also add sound to web pages; thus making them truly dynamic. The <embed> tag allows for the addition of sound to a web page. Valid sound formats are WAV (.wav), AU (.au), and MIDI (.mid). The syntax of this tag is:

```
<embed src="path/sound.xxx" loop="value">
```

The **loop** attribute defines the number of times the sound will repeat. The value would either be a number, signifying the number of repeats, or 'infinite,' signifying that the sound will repeat until a different page is loaded.

The <embed> tag behaves very much like the <img> tag; however, instead of displaying an image on the page, the <embed> tag plays the specified sound and displays a console on the page.



This console allows the visitor to your page to stop, pause, and control the volume of the background sound. As with the <img> tag, the <embed> tag also allows the page designer to control the height and width of the console. In order to display the console as above, use the following values:

```
<embed src="sound.xxx" height="60" width="145">
```

In order to not display the console at all, simply set the height and width attributes to 0.

# Advanced HTML

## Tables

Tables make it possible to format text, data, images, or some combination in such a way as to align them in a sensible manner. An HTML table consists of several container tags: `<table>`, `<tr>`, `<td>`, and `<th>`.

### `<table>`

This tag is the wrapper that surrounds the other tags that make up the table. The `<table>` tag is necessary in every table; without it or the ending `</table>` all the interior tags will be ignored.

### `<tr>`

The Table Row tag, `<tr>..</tr>`, denotes the beginning of a row of the table. The `</tr>` tag must be used to end each row. The total number of `<tr>` tags specifies the number of rows contained within a table.

### `<td>`

The Table Data tag, `<td>..</td>`, contains the actual data of the table. The `<td>` tag may only appear within a table row, `<tr>`, tag. Each data tag is called a cell and it is not necessary to have the same number of cells in each row -- the short rows will be filled with blank cells.

### `<th>`

The Table Header tag, `<th>..</th>`, acts the same as the `<td>` tag except that, by default, all data within the cell is aligned to the center and is boldface.

An example of a simple table:

```
<table>
  <tr>
    <th>Name</th>
    <th>BA</th>
    <th>HR</th>
  </tr>
  <tr>
    <td>Larkin</td>
    <td>.300</td>
    <td>3</td>
  </tr>
  <tr>
    <td>Morris</td>
    <td>.290</td>
    <td>6</td>
  </tr>
</table>
```

becomes

### **Name BA HR**

Larkin .300 3

Morris .290 6

As with most other tags, it is possible to change the appearance of a table through the use of attributes. Each table tag has its own attributes with some attributes having the ability to be used in more than one tag. The following is a description of each attribute and the table tags it can be used in.

The **border** attribute can only be used with the table tag. It controls the size of the border surrounding the table. The size is represented by a pixel value with the default being 0, no border.

```
<table border="2">
  <tr>
    <th>Name</th>
    <th>BA</th>
    <th>HR</th>
  </tr>
  <tr>
    <td>Larkin</td>
    <td>.300</td>
    <td>3</td>
  </tr>
  <tr>
    <td>Morris</td>
    <td>.290</td>
    <td>6</td>
  </tr>
</table>
```

becomes

<b>Name</b>	<b>BA</b>	<b>HR</b>
Larkin	.300	3
Morris	.290	6

The **cellpadding** and **cellspacing** attributes can only be used with the table tag. They control the space between table elements. The cellpadding attribute controls the amount of space between cell data and the cell wall. The cellspacing attribute controls the space between the cells of a table. The size is represented by a pixel value.

```
<table border="2" cellpadding="20" cellspacing="5">
  <tr>
    <th>Name</th>
    <th>BA</th>
    <th>HR</th>
  </tr>
  <tr>
    <td>Larkin</td>
    <td>.300</td>
    <td>3</td>
  </tr>
```

```

<tr>
  <td>Morris</td>
  <td>.290</td>
  <td>6</td>
</tr>
</table>

```

becomes

Name	BA	HR
Larkin	.300	3
Morris	.290	6

The **align** attribute can be used on all of the table tags. It controls the alignment of text to the left, center, or right inside a cell, or to align the table on the page. Valid values for this attribute are **align="right"**, **align="center"**, and **align="left"**. When used in a `<table>` tag, the entire table is affected. If used in the `<tr>` tag, all cells in that specific row will be aligned accordingly. If used within either the `<th>` or `<td>` tag, only that cell will be aligned.

```

<table border="2">
  <tr>
    <th>Player Name</th>
    <th>Batting Average</th>
    <th>Home Runs</th>
  </tr>
  <tr align="center">
    <td>Larkin</td>
    <td>.300</td>
    <td align="right">3</td>
  </tr>
  <tr>
    <td align="center">Morris</td>
    <td>.290</td>
    <td>6</td>
  </tr>
</table>

```

Player Name	Batting Avg.	Home Runs
Larkin	.300	3
Morris	.290	6

The **valign** attribute can be used on all of the tags. It controls the vertical alignment of the text within a cell. Valid values for this attribute are **valign="top"**, **valign="middle"**, and **valign="bottom"**. As with **align**, what cells the attribute affects is based on what tag it is used in. If the attribute is used in the `<table>` tag, all cells in the table are aligned



accordingly. If used in the <tr> tag, all cells in that specific row will be aligned accordingly. If used within either the <th> or <td> tag, only that cell will be aligned.

```
<table border="2">
  <tr>
    <th>Player Name</th>
    <th>Batting Average</th>
    <th>Home Runs</th>
  </tr>
  <tr>
    <td>Barry<br>Larkin</td>
    <td valign="top">.300</td>
    <td>3</td>
  </tr>
  <tr>
    <td>Hal<br>Morris</td>
    <td valign="bottom">.290</td>
    <td valign="middle">6</td>
  </tr>
</table>
```

Player Name	Batting Avg.	Home Runs
Barry Larkin	.300	3
Hal Morris	.290	6

The **bgcolor** attribute can be applied to all of the table tags. This attribute controls the background color of the cells. The color can be represented by either the actual color name (i.e. "red") or its hexadecimal value (i.e. #FF0000 = red). If the attribute is used in the <table> tag, all cells in the table are colored accordingly. If used in the <tr> tag, all cells in that specific row will be colored accordingly. If used within either the <th> or <td> tag, only that cell will be colored.

```
<table bgcolor="white" border="2">
  <tr bgcolor="#FF0000">
    <th>Player Name</th>
    <th>Batting Average</th>
    <th>Home Runs</th>
  </tr>
  <tr>
    <td>Barry<br>Larkin</td>
    <td>.300</td>
    <td bgcolor="#00FF00">3</td>
  </tr>
  <tr>
    <td>Hal<br>Morris</td>
    <td>.290</td>
    <td bgcolor="0000FF">6</td>
  </tr>
</table>
```

Player Name	Batting Avg.	Home Runs
Barry	.300	3

Larkin		
Hal Morris	.290	6

The **width** and **height** attributes can be applied to the <table>, <td>, and <th> tags. They control the size of the table elements. The sizes are represented as either pixel sizes or as a relative percentage. If the attribute is used in the <table> tag, the width and height control the size of the entire table. A percentage, in this case, is relative to the width and height of the entire web page.

```
<table border="2" width="35%">
  <tr>
    <th>Name</th>
    <th>BA</th>
    <th>HR</th>
  </tr>
  <tr>
    <td>Larkin</td>
    <td width="50">.300</td>
    <td>3</td>
  </tr>
  <tr>
    <td>Morris</td>
    <td>.290</td>
    <td height="50">6</td>
  </tr>
</table>
```

Name	BA	HR
Larkin	.300	3
Morris	.290	6

The **colspan** and **rowspan** attributes apply only to the <td> and <th> tags. They control the number of columns and rows (respectively) that a single cell spans. The value represents the number of columns or rows the cell should span. The results are the same whether this attribute is used in either tag.

```
<table border="2" cellpadding="3">
  <tr>
    <td colspan="4" bgcolor="red">Cincinnati Reds  
Stats</td>
  </tr>
  <tr>
    <th>Name</th>
    <th>BA</th>
    <th>HR</th>
    <th>RBI</th>
  </tr>
  <tr>
    <td>Larkin</td>
    <td>.300</td>
    <td>3</td>
```

```

    <td rowspan="2">40</td>
  </tr>
  <tr>
    <td>Morris</td>
    <td>.290</td>
    <td>6</td>
  </tr>
</table>

```

becomes

Cincinnati Reds Stats			
Name	BA	HR	RBI
Larkin	.300	3	40
Morris	.290	6	

## Frames

An HTML frame is an advanced formatting technique that allows a web page to be split up into sub-areas that each hold a separate HTML document. Each sub-area, or frame, retains the full functionality of a non-frame HTML document with the added ability to interact with the other frames. [Here is an example of a framed page.](#)

### Structure of a frame

Every framed HTML page begins with a **frame document**, that tells the browser exactly how to load each frame and where to place it on the screen. A frame document is exactly the same as a normal web page except that the <body> tag is replaced by the <frameset> tag. Let's examine, line-by-line, the frame document of the example framed page:

```

<html>
<head>
  <title>Frames Example</title>
</head>
<frameset rows="50%,*" border=1>
  <frameset cols="50%,*">
    <frame src="tile1.html">
    <frame src="tile2.html">
  </frameset>
  <frameset cols="60%,*">
    <frame scrolling="yes" src="tile3.html">
    <frame src="tile4.html" name="four">
  </frameset>
</frameset>
</html>

```

```
<frameset rows="50%,*" border="1">
```

This tag begins by setting the size of the two rows, in this case. By specifying '50%' the first row will be half the size of the screen. The '\*' means 'the rest of'; since in this case the first row takes up half the screen, the remainder of the screen is used by the second

row. It is possible to specify a pixel value for the size, but since screen size may vary it is often better to give relative values.

```
<frameset cols="50%,*">
```

This tag, since it is nested within the other frameset, will split the top row into, in this case, two pieces. This tag is setting the column size to half the screen, therefore both frames will be the same size.

```
<frame src="tile1.html">
```

```
<frame src="tile2.html">
```

These two frame tags are loading the html pages that will appear in the top two columns. The first instance of the frame tag loads the first HTML file, while the remaining instances load pages into the frames in the order that they are listed.

```
</frameset>
```

This ending tag closes the first row. Therefore, at this point, there are two pages loaded in the first row and the second row is blank.

```
<frameset cols="60%,*">
```

This frameset tag will now control the second (bottom) row. This time it sets the first column of the second row to 60%; thus the second column will be 40% the size of the screen.

```
<frame scrolling="yes" src="tile3.html">
```

This frame tag loads the HTML document 'tile3' into the first column of the second row. The **scrolling** attribute controls the frame's ability to be scrollable (as well as whether it has a scrollbar). Since the attribute is set to 'yes' this frame is scrollable.

```
<frame src="tile4.html" name="four">
```

This tag loads the final HTML document, placing it into the second row, second column. The **name** attribute gives the frame a name. This is used to load documents into a frame using a link contained within a separate frame. This is accomplished through the use of the **target** attribute of the <a href ... > tag. For example:

```
<a href="doc1.html" target="frame2">
```

This tag will load the HTML document 'doc1' into the frame named 'frame2'. This is how the example was able to change frame #4 to frame #5. Also, in order to load a document full screen instead of inside a frame, use the target value "\_top".

```
<a href="fullscreen.html" target="_top">
```

## Good Web Publishing Techniques

In order to create effective web pages, it is not enough to know how to program in HTML. It is also important to understand what is included in well designed web pages. The design and lay out of a web page is as important as the information it contains. The following is a list of things to consider when designing an effective web page:

- **Design first** - Sit down before you start coding the page and layout what you want the page to look like. The worst thing you can do is 'improvise' your web site.
- **Have a purpose** - Make sure you have a reason for creating each page. It can be as simple as "I want people to know about me." (No, "just because" is not a good reason) Make sure the information on the page follows and complements the purpose.
- **Choose good backgrounds** - Make sure when, or if, you decide to use a background color or image that it does not interfere with either the text or images

contained within the page. Also, choose backgrounds that are easy on the eyes: neon yellow text on an orange background is sure to make any visitor to your page run screaming to the optometrist.

- **No Spelling or Grammatical Errors** - Check and double check your page for both. No one will take any of your information seriously if it is riddled with lazy errors.
- **Put the most important information first** - Only about 10% of all web visitors scroll through an entire page. Not many people will wade through an ocean of background or unrelated information to get to the good stuff hidden three pages down.
- **Don't put too much (or too little) information on a page** - If one page is fifteen screens long, it is a good bet that no one will take the time to get to the bottom. Likewise, it is a waste of bandwidth to have one page contain only a couple of sentences. Break up pages by topic, trying to keep them between one and four screens long.
- **Don't clutter** - Make sure to keep everything spaced logically. It makes viewing and understanding the page much easier when the visitor does not have to fight the layout of the page.
- **Make sure all links work** - Make sure when including links to other pages that you have linked to a stable page. Check the links contained within your page often to make sure they still work. There is nothing worse than clicking on a link and being rewarded with a "404 - Page not found" error.
- **Include a link to your main page** - Always include a link to your main (front) page on all your pages. This is especially helpful if your main page contains a site map, the web equivalent of a table of contents.
- **Avoid long download times** - Always keep in mind that most people do not have fast connections to the web; therefore, make sure that all of your images and other data are as small as possible. Most people will lose patience and stop loading a page after more than a minute.
- **No graphics larceny** - Please do not steal original pictures or images from other peoples' web pages for use on your own. Take the time to design your own graphic - it will make your page more individual.
- **Always use the *alt* attribute** - Not everyone who will visit your site uses a graphical browser. By including the *alt* attribute within the image tag everyone who visits your site can enjoy and understand it.
- **Always use the "target=\_top" attribute when using frames** - If you use frames on any of your pages, always use the 'target=\_top' attribute in <a href...> tags to link to outside pages. There is nothing worse than clicking on a link within a framed page that is supposed to go to an outside site and having the page load within that frame (in fact, some people have gotten sued for it).
- **Last but not least, TEST YOUR PAGE!!!** - Before you finalize your page, make sure you test it on many different browsers, at several different screen resolutions, and (if possible) on several different modem speeds.

# Introduction to JavaScript

Javascript allows us to add programming code to our web pages. It can be used to create small applications such as calculators, games etc. Javascript has many applications which can enhance the performance of your website. You can

1. Detect browsers. Yes, we can detect the browsers from which a client over the world wide web is making a request and depending on whether it is Netscape or Microsoft Internet Explorer, we can show the page better.
2. It can be used for "cookies". When a user comes to a specific website, we can store information about the user's machine, so that when he or she returns to the same site, we can use the information which was stored for that specific user.
3. We can validate forms using javascript. For instance, we can verify that that a person has entered the correct values as required by certain fields; if a user doesn't enter the correct information, we could **prompt** the user to do so.

## Implementing JavaScript in HTML

Javascript is different from HTML. As a result we need to indicate to the browser that there exists a javascript embedded in the web page. We do this using the `<script>` tag.

The browser looks for the `<script language="javascript">` ..... javascript statements....`</script>` tags to determine where a javascripts starts and where it ends. To print something like "This is an alert message" we write a small javascript as follows:

```
<html>
<head>
<title>Javascript Labs</title>
</head><br>
<body>

<script language="javascript">
    alert ("This is an alert message.");
</script>

</body>
</html>
```

### [Check it out](#)

"alert" used in the script above is a standard javascript command that when embedded in a script causes an alert box to pop up; it requires that the user click on the "OK" button to move on.

Entering the alert command outside the script tags causes the browser to interpret it as plain text, and it is just displayed on the screen.

Javascript can be embedded in the `<head>` as well as the `<body>` of the HTML page. However it is advisable to use it in the `<head>` of the web page.

## JavaScript Syntax

As well as embedding a javascript is in the `<script language="javascript">` and `</script>` tags, there are a few other things that you need to note about the syntax of javascript.

1. All lines in javascript end in a semicolon.
2. Text should always be included within `"`. If we don't include it in quotes then it is interpreted to be a javascript variable, and can lead to errors.
3. Javascript is case-sensitive for variables, but not for keywords; the variables `a`, and `A` are considered different. But typing `Alert("This is an error.")` is the same as `aLerT("This is an error")`.

We can also use javascript to write the same information on a web page, using the `document.write("string to be printed")` command.

Try running the following portion of javascript:

```
<html>
<head>
<title>Javascript Labs</title>
</head>
<body>
Sample program
<p>

<script>
    document.write("Running from within the script");
</script>

<p>
Out of the script
</body>
</html>
```

[Check it out](#)

# More JavaScript

## JavaScript Variables

Study (and perhaps run) the following example:

```
<html>
<head>
<title>Javascript Labs</title>
</head>
<body>

<script>
  course_name = "Frontiers";
  string_name = "This is the";
  str_to_display = string_name + " " + course_name + " course."
;
  document.write(str_to_display);
</script>

</body>
</html>
```

Comparing variable values:

```
if (course_name=="Frontiers") {
  document.write("Success");
}
else {
  document.write("Failed");
}
```

Some operators are:

- == Equal to
- != Not equal to
- < Less than
- > Greater than
- <= Less than or equals
- >= Greater than or equals

## Creating Popup Boxes

It is possible to create three different kinds of popup boxes through javascript.

1. Alert
2. Confirm
3. Prompt



When a confirm box is presented and the user clicks on OK the value returned is true.  
When the user clicks on CANCEL the value returned is false.

Consider the following:

```
if ( confirm("Do you agree with the terms and conditions of the
agreement") )
{
    alert("You agree and can continue setup");
}
else
{
    alert("You disagreed, exiting setup");
}
```

### **Using if...else Structures**

As we know, different browsers show the same web page differently. At such times we need to be able to bring up different pages to show accordingly depending upon the type of browser being used.

The syntax of the if else structure would be:

```
if (condition)
{
    action...
}
else
{
    action...
}
```

Now let's take a look at some sample javascript showing the if else structure, which you might want to test. Here we assume a 24 hour time format.

```
if (time >= 1200)
{
    alert ("It is after noon.");
}
else
{
    alert ("It is morning.");
}
```

In the example above, we would need to store a value in the variable 'time' which would check the condition and then take the necessary action.

We can also have nested if else structures, so that we can check for a variety of conditions in a nested fashion. Here's how:

```
if (condition1)
{
    action1...
}
else if (condition2)
{
    action2...
}
else if (condition3)
{
    action3...
}
...
else
{
    actionN...
}
```

Here's an example of nesting if else structures:

```
if (gpa > 3.5)
{
    alert ("You are a very good student.");
}
else if (gpa > 2.9)
{
    alert ("You are a good student.");
}
else if (gpa > 2.5)
{
    alert ("You have an OK GPA.");
}
else
{
    alert ("You really need to put in more effort.");
}
```

In the example above we would have a value stored in the variable 'gpa' which we would use in evaluating the various conditions of the if else structure.

You can also use **and**, **or**, and **not** to check for multiple conditions in the if statements.

# Advanced JavaScript

## Events and Writing Functions

You have seen some various pop-up boxes, the alert box and the confirm box. These boxes appear on screen even before the entire page loads in the browser window. Ideally, you would want to see these messages on the occurrence of an **event**. The way to do this is to use functions. When we write functions, we can perform specific tasks only when we need these tasks be performed and not otherwise. Here's a sample code for all 3 types of pop-up boxes:

### Pop up Boxes

```
<html>

<head>
<script>

    function alertSample()
    {
        alert("This is an alert box.");
    }

    function confirmSample()
    {
        if (confirm("If you are 18 or older press OK. "))
        {
            alert("confirm() returned TRUE. You can view this
page.");
        }
        else
        {
            alert("confirm() returned FALSE. You cannot view this
page.");
        }
    }

    function promptSample()
    {
        your_name = prompt("This is a prompt box. Now tell me
your name.");
        if (your_name != "undefined")
        {
            alert("You entered " + your_name + ".");
        }
        else
        {
            alert("You didn't enter anything.");
        }
    }

</script>
</head>
```

```

<body>

<form name="javascriptform">
<input type="button" value="Alert" onClick="alertSample()">
<input type="button" value="Confirm" onClick="confirmSample()">
<input type="button" value="Prompt" onClick="promptSample()">
</form>

</body>

</html>

```

Try clicking on the button below and you will trigger an event.

Similarly, another function would cause another type of popup window to open. The following code is from above:

```

<form name="javascriptform">
...
<input type="button" value="Confirm"
onClick="confirmSample()">
...
</form>

```

This creates the confirm button:

The `value` tag above with `value="Prompt"` and `onClick="promptSample()"` produces the following:

The syntax of a function in javascript is as follows:

```

function function_name(variable1, variable2,... variableN)
{
    function statements...
}

```

The open and closed curly brackets (`{,}`) mark the start and the end of the function.

## Looping

There are two different kinds of loops: the 'for loop' and the 'while loop'.

### For loop

The syntax of the 'for loop' is:

```

for( variable = initial_value;
    variable conditional final_value;
    variable operator increment/decrement_value )
{
    statements to be looped...
}

```

A factorial is the sum of numbers beginning with 1 up to some other amount (n). The resulting sum is referred to as "n factorial" or "n!". A function to calculate n!:

```

function factorial(n) {

```

```

var sum = 0;
if ( (n >= 1) && (n <= 100) ) {
    for ( var i=1; i<=n; i+1 )
    {
        sum = sum + i;
    }
    alert(n + "! = " + sum);
}
else
{
    alert("Invalid Input");
}
}

```

The following code is placed within the `body` tags.

```

Enter a number between 1 and 100:
<form name="sampleForm">
<input type="text"
    size="5"
    name="upTo">
<input type="button"
    value="Calculate"
    onClick="factorial(document.sampleForm.upTo.value) ">

```

Enter a number between 1 and 100:

## While loop

The syntax of the while loop is:

```

while (loop_variable relational_operator
terminating_value)
{
    statements to be looped...
}

```

The previous for loop could be replaced by the following while loop:

```

var i = 1;
while ( i <= n )
{
    sum = sum + i;
    ++i;
}

```

`++i` means "increment i". It is the same as `i = i + 1`

## Arrays

Here's how we declare an array:

```
array_name = new Array;
```

array\_name can be any variable name,

You can initialize the elements of an array to the value 0 as follows:

```
for (i = 0 ;i < 20; i++)
{
    array_name[i] = 0;
}
```

The HTML code displays a button labeled "Enter" and a text area. When the button is clicked (`onClick`) the JavaScript function `displayArray()` is called. A sample array is initialized and the contents of that array are displayed in the text area.

```
<html>
<head>
    <title>Arrays</title>
    <script language="javascript">
        function displayArray()
        {
            myArray = new Array;
            myArray[0] = "Huntington";
            myArray[1] = "Hills";
            myArray[2] = "High";
            for ( var i=0; i<myArray.length; ++i )
            {
                document.myForm.textFour.value =
document.myForm.textFour.value + " " + myArray[i];
            }
        }
    </script>
</head>
<body>
    <form name="myForm">
        <p>
            <input type="button"
                value="Enter"
                name="enter"
                onClick="displayArray();">
        <p>
            Result:
            <input type="text"
                size="40"
                name="textFour">
        </form>
</body>
</html>
```

[Check it out](#)

All arrays in JavaScript are objects. One property of any Array object is the `length`. Since there are 3 entries in `myArray` in the example above, `myArray.length = 3`.

## Cookies

Cookies are used by web pages to help get and store data on the client side. They are taken care of by the browsers in name/value pairs.

When the following html is loaded into a web browser, a cookie set for the corresponding site with name="Cookie" and value="Monster!!!":

```
<html>

<head>
<script>
    function setCookie ()
    {
        document.cookie="Cookie=Monster!!!";
    }
</script>
</head>

<body onLoad="setCookie()">
Cookie Monster
</body>

</html>
```

### Set the cookie

The following HTML can then read the cookie:

```
<html>

<head>
<script>
    function readCookie () {
        if ( document.cookie.length < 1 ) {
            document.write('No cookie set for this site.');
```

```
        }else {
            document.write('Cookie for this site: ' +
document.cookie);
        }
    }
</script>
</head>
```

```
<body onLoad="setCookie()">
Cookie Monster
</body>
```

```
</html>
```

### Read the cookie

For this site, depending on the browser you are using, the cookie will be stored with the following information:

Name:        Cookie  
Value:        Monster!!!

Host: penguin.wpi.edu:4546  
Path: /course/web/lab02  
Server Secure: no  
Expires: at end of session



# Introductory Perl

## Introduction

Perl means *Practical Extraction and Report Language*. Those of you who know *awk*, *sed*, and *shell scripts* will think Perl is very much like these.

Actually, Perl does much more than each of these. Perl syntax also looks like these as well as like C. This lab presumes only that you know C.

There is a good Perl book for beginners by [Eileen Quigley](#), published by Prentice-Hall called *Perl by Example*.

[Try some examples](#)

## Scalar Variables

The following is excerpted from Jacqueline D. Hamilton's awesome online class at <http://www.cgi101.com>. (She has now published this in book form and I highly recommend it.) In particular, this material comes from <http://www.cgi101.com/class/ch2/text.html>.

Perl has three types of variables: scalars, arrays, and hashes.

A *scalar variable* stores a single (scalar) value. Perl scalar names are prefixed with a dollar sign (\$), so for example, *\$x*, *\$y*, *\$z*, *\$username*, and *\$url* are all examples of scalar variable names. Here's how these variables are used:

```
$foo = 1;  
$name = "Fred";  
$pi = 3.141592;
```

You do not need to declare a variable before using it. A scalar can hold data of any type, be it a string, a number, or whatnot. You can also use scalars in double-quoted strings:

```
$fnord = 23;  
$blee = "The magic number is $fnord.";
```

Now if you print **\$blee**, you will get "The magic number is 23."

Let's create a file called *first.pl* and add some scalars to it:

```
#!/usr/bin/perl
```

```
$classname = "Electronic Documents";

print "Hello there.  What is your name?\n";

$you = <STDIN>;

chomp($you);

print "Hello, $you.  Welcome to $classname.\n";
```

Save, and run the script by typing: Perl *first.pl*

This time, the program will prompt you for your name, and read your name using the following line:

```
$you = <STDIN>;
```

STDIN is *standard input*. This is the default input channel for your script; if you're running your script in the shell, STDIN is whatever you type as the script runs.

The program will print "Hello there. What is your name?", then pause and wait for you to type something in. (Be sure to hit return when you're through typing your name.) Whatever you typed is stored in the scalar variable *\$you*. Since *\$you* also contains the carriage return itself, we use

```
chomp($you);
```

to remove the carriage return from the end of the string you typed in. The following print statement:

```
print "Hello, $you. Welcome to $classname.\n";
```

prints this new value of *\$you*. As in C, The "\n" at the end of the line is the perl syntax for a carriage return.

## Arrays

Perl array names are prefixed with an at-sign (@). Here is an example:

```
@colors = ("red", "green", "blue");
```

In Perl, array indices start with 0, so to refer to the first element of the array @colors, you use \$colors[0]. Note that when you're referring to a single element of an array, you prefix

the name with a \$ instead of the @. The \$-sign again indicates that it's a single (scalar) value; the @-sign means you're talking about the entire array.

If you wanted to loop through an array, printing out all of the values, you could print each element one at a time:

```
#!/usr/bin/perl
# this is a comment
# any line that starts with a "#" is a comment.
```

```
@colors = ("red","green","blue");
```

```
print "$colors[0]\n";
print "$colors[1]\n";
print "$colors[2]\n";
```

Or, a much easier way to do this is to use the *foreach* construct:

```
#!/usr/bin/perl
# this is a comment
# any line that starts with a "#" is a comment.
```

```
@colors = ("red","green","blue");
```

```
foreach $i (@colors) {
    print "$i\n";
}
```

For each iteration of the foreach loop, Perl sets \$i to an element of the @colors array - the first iteration, \$i is "red". As in C, the braces {} define where the loop begins and end, so for any code appearing between the braces, \$i is set to the current loop iterator.

## **Array Functions**

Since an array is an ordered list of elements, there are a number of functions you can use to get data out of (or put data into) the list:

```
@colors = ("red","green","blue","cyan","magenta","black","yellow");
```

```
$elt = pop(@colors); # returns "yellow", the last value of the array.
```

**\$elt = shift(@colors); # returns "red", the first value of the array.**

In these examples we've set \$elt to the value returned, but you don't have to do that - if you just wanted to get rid of the first value in an array, for example, you'd just type `shift(@arrayname)`. Both `shift` and `pop` affect the array itself, by removing an element; in the above example, after you `pop` "yellow" off the end of `@colors`, the array is then equal to ("red", "green", "blue", "cyan", "magenta", "black"). And after you `shift` "red" off the front, the array becomes ("green", "blue", "cyan", "magenta", "black").

You can also add data to an array:

```
@colors = ("green", "blue", "cyan", "magenta", "black");  
push(@colors, "orange"); # adds "orange" to the end of the @colors array
```

`@colors` now becomes ("green", "blue", "cyan", "magenta", "black", "orange").

```
@morecolors = ("purple", "teal", "azure");  
push(@colors, @morecolors); # appends the values in @morecolors to the  
end of @colors
```

`@colors` now becomes ("green", "blue", "cyan", "magenta", "black", "orange", "purple", "teal", "azure").

Here are a few other useful functions for array manipulation:

```
@colors = ("green", "blue", "cyan", "magenta", "black");  
sort(@colors); # sorts the values of @colors alphabetically
```

`@colors` now becomes ("black", "blue", "cyan", "green", "magenta"). Note that `sort` does not change the actual values of the array itself, so if you want to save the sorted array, you have to do something like this:

```
@sortedlist = sort(@colors);
```

The same thing is true for the `reverse` function:

```
@colors = ("green", "blue", "cyan", "magenta", "black");  
reverse(@colors); # inverts the @colors array
```

`@colors` now becomes ("black", "magenta", "cyan", "blue", "green"). Again, if you want to save the inverted list, you must assign it to another array.

```
$#colors # length-1 of the @colors array, or the last index of the array
```

In this example, the value of `$#colors` is 4. The actual length of the array is 5, but since Perl lists count from 0, the index of the last element is `length - 1`. If you want the actual length of the array (the number of elements), you'd use the `scalar` function:

```
scalar(@colors); # the actual length of the array
```

In this case, the value of `scalar(@colors)` is equal to 5.

```
join(" ", @colors); # joins @colors into a single string separated by the  
expression " ", "
```

@colors becomes a single string: "black, magenta, cyan, blue, green".

## Hashes

A hash is a special kind of array - an associative array, or paired group of elements. Perl hash names are prefixed with a percent sign (%), and consist of pairs of elements - a key and a data value. Here's how to define a hash:

```
Hash Name      key          value

%pages = ( "fred",    "http://www.wpi.edu/~fred/",
           "beth",    "http://www.wpi.edu/~beth/",
           "john",    "http://www.wpi.edu/~john/" );
```

Another way to define a hash would be as follows:

```
%pages = ( fred => "http://www.wpi.edu/~fred/", beth =>
"http://www.wpi.edu/~beth/", john => "http://www.wpi.edu/~john/" );
```

Note that quotes aren't needed here.

This hash consists of a person's name for the key, and their URL as the data element. You refer to the individual elements of the hash with a \$ sign (just like you did with arrays):

```
$pages{'fred'}
```

In this case, "fred" is the key, and \$pages{'fred'} is the value associated with that key - in this case, it would be "http://www.cgi101.com/~fred".

If you want to print out all the values in a hash, you'll need a foreach loop:

```
foreach $key (keys %pages) { print "$key's page: $pages{$key}\n"; }
```

This example uses the *keys* function, which returns an array consisting only of the keys of the named hash. One drawback is that *keys %hashname* will return the keys in random order - in this example, *keys %pages* could return ("fred", "beth", "john") or ("beth", "fred", "john") or any combination of the three. If you want to print out the hash in exact order, you have to specify the keys in the foreach loop:

```
foreach $key ("fred","beth","john") { print "$key's page: $pages{$key}\n"; }
```

Hashes will be especially useful when you use CGIs that parse form data, because you'll be able to do things like \$FORM{'lastname'} to refer to the "lastname" input field of your form.

## Hash Functions

Here is a quick overview of the Perl functions you can use when working with hashes.

```
delete $hash{$key}    # deletes the specified key/value pair,  
                      # and returns the deleted value
```

```
exists $hash{$key}    # returns true if the specified key  
exists                # in the hash.
```

```
keys %hash            # returns a list of keys for that hash
```

```
values %hash          # returns a list of values for that hash
```

```
scalar %hash          # returns true if the hash has elements  
                      # defined (e.g. it's not an empty hash)
```

# Introductory CGI

## ***What is CGI?***

CGI is not a language. It's a simple protocol that can be used to communicate between Web forms and your program. A CGI script can be written in any language that can read STDIN, write to STDOUT, and read environment variables, i.e. virtually any programming language, including C, Perl, or even shell scripting.

## ***Structure of a CGI Script***

Here's the typical sequence of steps for a CGI script:

1. Read the user's form input.
2. Do what you want with the data.
3. Write the HTML response to STDOUT.

The first and last steps are described below.

---

## ***Reading the User's Form Input***

When the user submits the form, your script receives the form data as a set of name-value pairs. The names are what you defined in the INPUT tags (or SELECT or TEXTAREA tags), and the values are whatever the user typed in or selected. (Users can also submit files with forms, but this primer doesn't cover that.)

This set of name-value pairs is given to you as one long string, which you need to parse. It's not very complicated, and there are plenty of existing routines to do it for you. Here's [one in Perl](#), [a simpler one in Perl](#), or [one in C](#). For a more elaborate CGI framework, see Perl's [CGI.pm](#) module. The [CGI directory at Yahoo](#) includes many CGI routines (and pre-written scripts), in various languages.

If that's good enough for you, skip to the next section. If you'd rather do it yourself, or you're just curious, the long string is in one of these two formats:

```
"name1=value1&name2=value2&name3=value3"  
"name1=value1;name2=value2;name3=value3"
```

So just split on the ampersands or semicolons, then on the equal signs. Then, do two more things to each name and value:

1. Convert all "+" characters to spaces, and

2. Convert all "%xx" sequences to the single character whose ascii value is "xx", in hex. For example, convert "%3d" to "=".

This is needed because the original long string is **URL-encoded**, to allow for equal signs, ampersands, and so forth in the user's input.

So where do you get the long string? That depends on the HTTP method the form was submitted with:

- For GET submissions, it's in the environment variable **QUERY\_STRING**.
- For POST submissions, read it from STDIN. The exact number of bytes to read is in the environment variable **CONTENT\_LENGTH**.

*(If you're wondering about the difference between GET and POST, see the [footnote discussing it](#). Short answer: POST is more general-purpose, but GET is fine for small forms.)*

---

## ***Sending the Response Back to the User***

First, write the line

```
Content-type: text/html
```

plus another blank line, to STDOUT. After that, write your HTML response page to STDOUT, and it will be sent to the user when your script is done. That's all there is to it.

Yes, you're generating HTML code on the fly. It's not hard; it's actually pretty straightforward. HTML was designed to be simple enough to generate this way.

If you want to send back an image or other non-HTML response, [here's how to do it](#).

---

## **That's it. Good Luck!**

See how easy it is? If you still don't believe me, go ahead and write a script. Make sure to put the file in the right place on your server, and make it executable.



# Introductory CGI

## **What is CGI?**

CGI is not a language. It's a simple protocol that can be used to communicate between Web forms and your program. A CGI script can be written in any language that can read STDIN, write to STDOUT, and read environment variables, i.e. virtually any programming language, including C, Perl, or even shell scripting.

## **Structure of a CGI Script**

Here's the typical sequence of steps for a CGI script:

1. Read the user's form input.
2. Do what you want with the data.
3. Write the HTML response to STDOUT.

The first and last steps are described below.

---

## **Reading the User's Form Input**

When the user submits the form, your script receives the form data as a set of name-value pairs. The names are what you defined in the INPUT tags (or SELECT or TEXTAREA tags), and the values are whatever the user typed in or selected. (Users can also submit files with forms, but this primer doesn't cover that.)

This set of name-value pairs is given to you as one long string, which you need to parse. It's not very complicated, and there are plenty of existing routines to do it for you. Here's [one in Perl](#), [a simpler one in Perl](#), or [one in C](#). For a more elaborate CGI framework, see Perl's [CGI.pm](#) module. The [CGI directory at Yahoo](#) includes many CGI routines (and pre-written scripts), in various languages.

If that's good enough for you, skip to the next section. If you'd rather do it yourself, or you're just curious, the long string is in one of these two formats:

**"name1=value1&name2=value2&name3=value3"**  
**"name1=value1;name2=value2;name3=value3"**

So just split on the ampersands or semicolons, then on the equal signs. Then, do two more things to each name and value:

1. Convert all "+" characters to spaces, and
2. Convert all "%xx" sequences to the single character whose ascii value is "xx", in hex. For example, convert "%3d" to "=".

This is needed because the original long string is **URL-encoded**, to allow for equal signs, ampersands, and so forth in the user's input.

So where do you get the long string? That depends on the HTTP method the form was submitted with:

- For GET submissions, it's in the environment variable **QUERY\_STRING**.
- For POST submissions, read it from STDIN. The exact number of bytes to read is in the environment variable **CONTENT\_LENGTH**.

*(If you're wondering about the difference between GET and POST, see the [footnote discussing it](#). Short answer: POST is more general-purpose, but GET is fine for small forms.)*

---

## ***Sending the Response Back to the User***

First, write the line

```
Content-type: text/html
```

plus another blank line, to STDOUT. After that, write your HTML response page to STDOUT, and it will be sent to the user when your script is done. That's all there is to it.

Yes, you're generating HTML code on the fly. It's not hard; it's actually pretty straightforward. HTML was designed to be simple enough to generate this way.

If you want to send back an image or other non-HTML response, [here's how to do it](#).

---

## **That's it. Good Luck!**

See how easy it is? If you still don't believe me, go ahead and write a script. Make sure to put the file in the right place on your server, and make it executable.

# Advanced Perl

As in the preceding, the following is excerpted from Jacqueline D. Hamilton's awesome online class at <http://www.cgi101.com>. (She has now published this in book form and I highly recommend it.) In particular, this material comes from <http://www.cgi101.com/class/ch4/text.html>.

## Introduction

Now that you have learned some Perl we will use Perl to create useful forms.

When sending form data to your CGI, your web server encodes the data being sent. Alphanumeric characters are sent as themselves; spaces are converted to plus signs (+); other characters - like tabs, quotes, etc. - are converted to "%HH" - a percent sign and two hexadecimal digits representing the ASCII code of the character. This is called URL encoding. Here's a table of some commonly encoded characters:

### Normal Character URL Encoded String

<code>\t (tab)</code>	<code>%09</code>
<code>\n (return)</code>	<code>%0A</code>
<code>/</code>	<code>%2F</code>
<code>~</code>	<code>%7E</code>
<code>:</code>	<code>%3A</code>
<code>;</code>	<code>%3B</code>
<code>@</code>	<code>%40</code>
<code>&amp;</code>	<code>%26</code>

In order to do anything useful with the data, your CGI script must decode these. Fortunately, this is pretty easy to do in Perl, using the *substitute* and *translate* commands. Perl has powerful pattern matching and replacement capabilities; it can match the most complex patterns in a string, using regular expressions. But it's also quite capable of the most simple replacements. The basic syntax for substitutions is:

**`$mystring =~ s/pattern/replacement/;`**

This command substitutes "pattern" for "replacement" in the scalar variable "\$mystring". Notice the operator is a `=~` (an equal sign followed by a tilde) - this is a special operator for Perl, telling it that it's about to do a pattern match or replacement. Here's an example of how it works:

```
$greetings = "Hello. My name is xname.\n"; $greetings =~ s/xname/Bob/;
print $greetings;
```

The above code will print out "Hello. My name is Bob." Notice the substitution has replaced "xname" with "Bob" in the \$greetings string.

A similar but slightly different command is the translate command:

```
$mystring =~ tr/searchlist/replacementlist/;
```

This command translates every character in "searchlist" to its corresponding character in "replacementlist", for the entire value of \$mystring. One common use of this is to change the case of all characters in a string:

```
$lowerc =~ tr/[A-Z]/[a-z]/;
```

This results in \$lowerc being translated to all lowercase letters. The brackets around [A-Z] denote a class of characters to match.

### *Decoding Form Data*

With the POST method, form data is sent in an input stream from the server to your CGI script. To get this data, store it, and decode it, we'll use the following block of code (which is a more complex script that can be used with the [forms page](#) from Lab #1:

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+//;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
```

Let's look at each part of this. First, we read the input stream using this line:

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
```

The input stream is coming in over STDIN (standard input), and we're using Perl's read function to store the data into the scalar variable \$buffer. You'll also notice the third argument to the read function, which specifies the length of data to be read; we want to read to the end of the CONTENT\_LENGTH, which is set as an environment variable by the server.

Next we split the buffer into an array of pairs:

```
@pairs = split(/&/, $buffer);
```

Form data pairs are separated by & signs when they are transmitted - for example, `fname=joe&lname=smith`. Now we'll use a *foreach loop* to further splits each pair on the equal signs:

```
foreach $pair (@pairs) { ($name, $value) = split(/=/, $pair);
```

The next line translates every "+" sign back to a space:

```
$value =~ tr/+/ /;
```

Next is a rather complicated regular expression that substitutes every %HH hex pair back to its equivalent ASCII character, using the `pack()` function. For now we'll just use it to parse the form data:

```
$value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
```

Finally, we store the values into a hash called %FORM:

```
$FORM{$name} = $value; }
```

The keys of %FORM are the form input names themselves. So, for example, if you have three text fields in the form - called *name*, *email-address*, and *age* - you could refer to them in your script by using `$FORM{'name'}`, `$FORM{'email-address'}`, and `$FORM{'age'}`.

Let's try it. Create a new CGI script with the following, calling it `post.cgi` (or `post.pl`), save it, and `chmod` it:

```
#!/usr/bin/perl

print "Content-type:text/html\n\n";

read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

@pairs = split(/&/, $buffer);

foreach $pair (@pairs) {

    ($name, $value) = split(/=/, $pair);

    $value =~ tr/+/ /;

    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;

    $FORM{$name} = $value;

}

print "<html><head><title>Form Output</title></head><body>";
print "<h2>Results from FORM post</h2>\n";
```

```
foreach $key (keys(%FORM)) {  
    print "$key = $FORM{$key}<br>";  
}  
  
print "</body></html>";
```

Source code: <http://www.cgi101.com/class/ch4/post.txt>

This code can be used to handle almost any form, from a simple guestbook form to a more complex order form. Whatever variables you have in your form, this CGI will print them out, along with the data that was entered.

Let's test the script. Create an HTML form with the fields listed below:

```
<form action="post.cgi" method="POST">  
    Your Name: <input type="text" name="name">  
    Email Address: <input type="text" name="email">  
    Age: <input type="text" name="age">  
    Favorite Color: <input type="text" name="favorite_color">  
<input type="submit" value="Send">  
<input type="reset" value="Clear Form">  
</form>
```

Source code: <http://www.cgi101.com/class/ch4/post.html>

Enter some data into the fields, and press "send" when finished. The output will be the variable names of these text boxes, plus the actual data you typed into each field.

**Tip:** If you've had trouble getting the boxes to align on your form, try putting `<pre>` tags around the input fields. Then you can line them up with your text editor, and the result is a much neater looking form. The reason for this is that most web browsers use a fixed-width font (like Monaco or Courier) for preformatted text, so aligning forms and other data is much easier in a preformatted text block than in regular HTML. This will only work if your text editor is also using a fixed-width font! Another way to align input boxes is to put them all into a table, with the input name in the left column, and the input box in the right column.

### *A Form-to-Email CGI*

Most people using forms want the data emailed back to them, so, let's write a form-to-mail CGI. First you'll need to figure out where the sendmail program lives on the Unix system you're on. (For cgi101.com, it's in /usr/sbin/sendmail. If you're not sure where yours is, try doing "which sendmail" or "whereis sendmail"; usually one of these two commands will yield the location of the sendmail program.)

Copy your post.cgi to a new file named mail.cgi. Now the only change will be to the foreach loop. Instead of printing to standard output (the HTML page the person sees after clicking submit), you want to print the values of the variables to a mail message. So, first, we must open a *pipe* to the sendmail program:

**`$mailprog = '/usr/sbin/sendmail'; open (MAIL, "|$mailprog -t")`**

The pipe causes all of the output we print to that filehandle (MAIL) to be fed directly to the sendmail program as if it were standard input to that program.

You also need to specify the recipient of the email, with either:

**`$recipient = 'nullbox@cgi101.com'; $recipient = "nullbox\@cgi101.com";`**

Perl will complain if you use an "@" sign inside a double-quoted string or a print <<EndHTML block. You can safely put an @-sign inside a single-quoted string, like 'nullbox@cgi101.com', or you can escape the @-sign in other strings by using a backslash. For example, "nullbox\@cgi101.com".

You don't need to include the comments in the following code; they are just there to show you what's happening.

```
#!/usr/bin/perl

print "Content-type:text/html\n\n";

# parse the form data.
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+//;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
}
```

```
# where is the mail program?
$mailprog = '/usr/sbin/sendmail';

# change this to your own email address

$recipient = 'nullbox@cgil01.com';

# this opens an output stream and pipes it directly to the
# sendmail program.  If sendmail can't be found, abort nicely
# by calling the dienice subroutine (see below)

open (MAIL, "|$mailprog -t") or dienice("Can't access
$mailprog!\n");

# here we're printing out the header info for the mail
# message.  You must specify who it's to, or it won't be
# delivered:

print MAIL "To: $recipient\n";

# Reply-to can be set to the email address of the sender,
# assuming you have actually defined a field in your form
# called 'email'.

print MAIL "Reply-to: $FORM{'email'} ($FORM{'name'})\n";

# print out a subject line so you know it's from your form cgi.
```



```

# The two \n\n's end the header section of the message.
# anything you print after this point will be part of the
# body of the mail.

print MAIL "Subject: Form Data\n\n";

# here you're just printing out all the variables and values,
# just like before in the previous script, only the output
# is to the mail message rather than the followup HTML page.

foreach $key (keys(%FORM)) {
    print MAIL "$key = $FORM{$key}\n";
}

# when you finish writing to the mail message, be sure to
# close the input stream so it actually gets mailed.

close(MAIL);

# now print something to the HTML page, usually thanking
# the person for filling out the form, and giving them a
# link back to your homepage

print <<EndHTML;
<h2>Thank You</h2>
Thank you for writing. Your mail has been delivered.<p>
Return to our <a href="index.html">home page</a>.
</body></html>
EndHTML

```

```

# The dienice subroutine, for handling errors.
sub dienice {
    my($errmsg) = @_ ;
    print "<h2>Error</h2>\n";
    print "$errmsg<p>\n";
    print "</body></html>\n";
    exit;
}

```

Now let's test the new script. Here's the form again, only the action this time points to mail.cgi:

```

<form action="mail.cgi" method="POST">
    Your Name: <input type="text" name="name">
    Email Address: <input type="text" name="email">
    Age: <input type="text" name="age">
    Favorite Color: <input type="text" name="favorite_color">
    <input type="submit" value="Send">
    <input type="reset" value="Clear Form">
</form>

```

Save it, enter some data into the form, and press "send". If the script runs successfully, you'll get email in a few moments with the results of your post. (Remember to change the \$recipient in the form to your email address!)

### **Sending Mail to More Than One Recipient**

What if you want to send the output of the form to more than one email address? Simple: just add the desired addresses to the \$recipients line:

```
$recipient = 'kira@cgi101.com, kira@io.com, webmaster@cgi101.com';
```

### ***Subroutines***

In the above script we used a new structure: a subroutine called "dienice." As in many languages, a subroutine is a block of code, separate from the main program, that only gets run if it's directly called. In the above example, *dienice* only runs if the main program can't open sendmail. Rather than aborting and giving you a server error (or worse, NO error), you want your script to give you some useful data about what went wrong; *dienice* does that, by printing the error message and closing html tags, and exiting from Perl. There are several ways to call a subroutine:

***&subname; &subname(args); subname; subname(args);***

The &-sign before the subroutine name is optional. ***args are values to pass into the subroutine.***

***Subroutines are useful for isolating blocks of code that are reused frequently in your script. The structure of a subroutine is as follows:***

***sub subname { ...code to execute... }***

***A subroutine can be placed anywhere in your CGI, though for readability it's usually best to put them at the end, after your main code. You can also include and use subroutines from different files and modules.***

***You can pass data into your subroutines. For example:***

***mysub(\$a,\$b,\$c);***

***This passes the scalar variables \$a, \$b, and \$c to the mysub subroutine. The data being passed (the arguments) are sent as a list. The subroutine accesses the list of arguments via the special array "@\_". You can then assign the elements of that array to special temporary variables:***

***sub mysub { my(\$tmpa, \$tmpb, \$tmpc) = @\_; ...code to execute... }***

***Notice the my in front of the variable list? my is a Perl function that limits the scope of a variable or list of variables to the enclosing subroutine. This keeps your temporary variables visible only to the subroutine itself (where they're actually needed and used), rather than to the entire script (where they're not needed).***

***We'll be using the dienice subroutine throughout the rest of the book, as a generic catch-all error-handler.***

# Introduction to Java Applets

## Basic Applet Structure

Most programs run directly on your computer. A Java applet is a special kind of program that runs inside a web browser. The **Applet class** does what's needed to make that work.

Let's look at a bare-bones applet. This applet will just put a gray box on the screen.

```
GrayBox.java

import java.awt.*;
import java.applet.*;

public class GrayBox extends Applet
{
}
```

[Check this code](#)

In JavaBOTL, even the bare-bones program did more than the applet above. We learned that a lot of behind the scenes stuff was going on. Now, we have to do some of that behind the scenes stuff ourselves. This gives us a lot more freedom.

Let's look at this program line by line. The first two lines are:

```
import java.awt.*;
import java.applet.*;
```

We use the `import` keyword to get some things needed for our applet from other files. Importing `java.awt.*` gets all of the things needed for drawing on the screen in Java. Doing the same with `java.applet.*` gives us all the things we need to build applets. Now let's look at the next line.

```
public class GrayBox extends Applet
```

The `class` keyword is used whenever we define a new type of object. The syntax for defining a new **class** is:

```
public class NewClassName extends ExistingClassName
```

**GrayBox** is the name of our new **class**. It **extends**, or inherits from, **Applet**. The **Applet class** sets up a skeleton which we fill in to create Java applets. By inheriting from **class Applet**, we get some functions that we can use.

```
import java.awt.*;
import java.applet.*;

public class GrayBox extends Applet
{
    public void init() {
```

```

        // Do one time setup stuff
    }
    public void paint( Graphics g ) {
        // Draw the screen
    }
}

```

We can change the functions we inherited from the **Applet class** to have our **class** do more than it does by default. This is called overriding the functions. Two important functions we will override in our applet are **init** and **paint**.

## ***Paint Function***

Once we have our applet skeleton in place, we'd like to make our program do something. So let's start simple. Let's take a look at the classic "Hello, world!" program done as a Java applet. This program puts the words "Hello, world!" on the screen.

```

import java.awt.*;
import java.applet.*;

public class Hello extends Applet
{
    public void init()
    {
    }

    public void paint( Graphics g )
    {
        g.drawString("Hello, World!", 50, 25 );
    }
}

```

[Check this code](#)

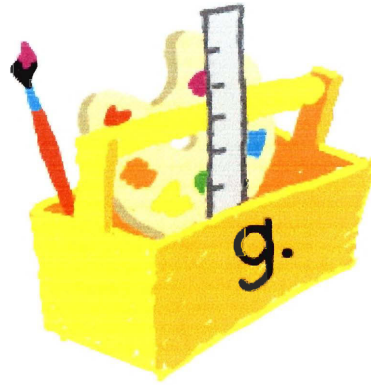
The **paint** function is called automatically every time our applet needs to draw itself. In JavaBOTL, even a blank program still drew stuff on the screen. If we want our Java applet to draw anything, we have to code it ourselves. The place we do this is inside the **paint** function.

```

public void paint( Graphics g )

```

The **paint** function of our applet has a parameter named **g** that is of type **Graphics**. We can use this parameter within the **paint** function. The **Graphics class** contains the tools we'll need to draw inside an applet's window. We use the **Graphics g** parameter as our "graphics toolbox".



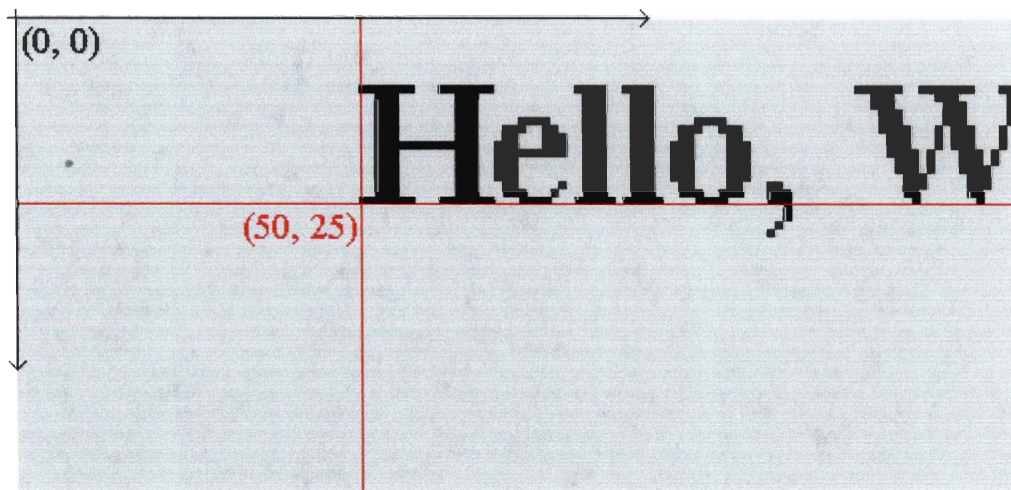
We can use the **Graphics class** to draw words, lines, shapes, or images. Our "graphics toolbox" has drawing functions that do these things. We will be using the **drawString** and **drawLine** functions.

To select which "tool" to use, we use the standard **ObjectName.FuctionName** syntax. For example, the line of code now added to our **paint** function uses the **drawString** tool from our "graphics toolbox".

```
public void paint( Graphics g )
{
    g.drawString( "Hello, World!", 50, 25 );
}
```

Again, **g** is our "graphics toolbox" that we got as a parameter to our **paint** function. The dot (.) tells Java we are going to be using something in that toolbox. In this case, that something is the **drawString** function.

The **drawString** function is one of the functions defined in **class Graphics**. It draws a string on the screen. A string is a word or phrase surrounded by double quotes. The string in our example is "Hello, World!". The **drawString** function takes three parameters. The first is the string we want to draw on the screen. The second and third are the x and y coordinates for where to start drawing.



Putting "Hello, world!" on the screen is a good start, but we'd like to do something a bit more interesting. This program shows how to draw straight lines on the screen.

```
import java.awt.*;
import java.applet.*;

public class ShowLine extends Applet
{
    public void init()
    {
    }
    public void paint( Graphics g )
    {
        g.drawString( "Check out this line!", 50, 25 );
        g.drawLine( 50, 35, 150, 35 );
    }
}
```

[Check this code](#)

The only thing that's different here is the new line in the **paint** function.

```
g.drawLine( 50, 35, 150, 35 );
```

The **drawLine** function is another tool from our "graphics toolbox". To use it, we do almost the same thing we did with **drawString**. The difference is that with **drawLine**, the parameters are two pair of x and y coordinates, which represent the start and end points of the line.

## ***Init Function***

Now that we know how to use the **paint** function, let's take a look at another important part of our applet skeleton -- the **init** function. Let's take a look at a program that shows how to use the **init** function.

```
import java.awt.*;
import java.applet.*;

public class BigX extends Applet
{
    public void init()
    {
        resize( 250, 300 );
    }

    public void paint( Graphics g )
    {
        g.drawLine( 0, 0, 250, 300 );
        g.drawLine( 0, 300, 250, 0 );
    }
}
```

```
    }  
}
```

[Check this code](#)

The **init** function is called automatically. It is called when an applet starts, before anything else.

```
public void init()
```

If we look at the applet skeleton, we see that all one-time set-up stuff is done in this function. Above, we use the **init** function to set the size of our applet window. We do this by using the **resize** function.

```
resize( 250, 300 );
```

The **resize** function takes two parameters. The first is the width (in pixels) that we want for our applet window. The second is the height. Our example uses the **resize** function to make the applet window 250 pixels wide and 300 pixels high. In the **paint** function we use **drawLine** to draw an 'X' over the entire applet window by using these values.

## ***Embedding Java Applets in HTML***

The nice thing about applets is that they run on the client side. The browser downloads the applet to your local machine and runs it there. The speed with which an applet is run, is relatively high, as it will be running in your browser.

An applet exists as compiled bytecode (mostly) on some computer connected to a network anywhere in the world. As soon as your browser downloads the applet, the browser's Java interpreter interprets the bytecode and executes the bytecode instructions.

As a simple illustration:

Server side:

```
java code (*.java)--> Compiler--> Compiled Bytecodes  
(*.class file)
```

As an example, let's use the BigX.java file as described above. The `javac` command is used to create a bytecode file as follows:

```
ls  
    BigX.java  
  
javac BigX.java  
  
ls  
    BigX.class    BigX.java
```

After you have a compiled bytecode (\*.class) file, you can embed the java applet in an HTML file using the `applet` tag. The syntax for this tag is as follows:

```
<applet    basecode="applet url"  
          code="applet filename"
```



```
        width="pixel width"  
        height="pixel height">  
</applet>
```

The `codebase` attribute is optional. It specifies the base URL or pathname of the applet specified in the code attribute.

The `code` attribute is required. It specifies the file that contains the compiled Java code for the applet. If the codebase attribute is specified, the code must be located relative to that location.

The `width` attribute is required. It specifies the initial width in pixels that the applet needs in the browser's window.

The `height` attribute is required. It specifies the initial height in pixels that the applet needs in the browser's window.

The `alt` attribute is optional. It specifies text that should be displayed by browsers that understand the applet tag but do not support Java.

The `name` attribute is optional. It gives a name to the applet instance. Applets that are running at the same time can look each other up by name and communicate with each other.

The `align` attribute is optional. It specifies the applet's alignment on the page. It behaves just like the align attribute of the img tag, and should support at least the same alignment values that the img tag does. These values include: top, middle, and bottom.

The `vspace` attribute is optional. It specifies the margin in pixels that the browser should put above and below the applet. It behaves just like the vspace attribute of the img tag.

The `hspace` attribute is optional. It specifies the margin in pixels that the browser should put on either side of the applet. It behaves just like the hspace attribute of the img tag.

The `<param>` tag, with its name and value attributes, specifies a named parameter and a string value that are passed to the applet. These parameters function like environment variables or command line arguments do for a regular application. An applet can look up the value of a parameter specified in a `param` tag with the `Applet.getParameter()` method. Any number of parameters can be included for an applet.

If a web browser does not support Java and does not understand the applet tag, it will ignore the applet and its parameters and will simply display any text that appears in the `alternate-html` field.

## More Java Applets

### Variables

To write our Tic Tac Toe applet, we are going to have to use variables so it can remember important details. Let's take a look at a Java applet that uses variables to show a quote from *The Princess Bride*.

```
import java.awt.*;
import java.applet.*;

public class PrincessBride extends Applet
{
    int Xposition;

    public void init()
    {
        Xposition = 100;
    }

    public void paint( Graphics g )
    {
        g.drawString( "Hello!", Xposition, 25 );
        g.drawString( "My name is Inigo Montoya.", Xposition, 50
);
        g.drawString( "You killed my father.", Xposition, 75 );
        g.drawString( "Prepare to die.", Xposition, 100 );
    }
}
```

[Check this code](#)

The first thing in the definition of our **PrincessBride class** is a variable declaration.

```
int Xposition;
```

We declare a variable called **Xposition** of type **int**, so it can only hold integer values. The variable **Xposition** is declared here so that both the **init** and the **paint** functions can use it.

After we declare the variable, we need to initialize it. If we look at the applet skeleton from the basic applet structure we see that all one-time set-up stuff, such as variable initialization, is done in the **init** function. In our example, we initialize the **Xposition** variable to have value of 100.

```
Xposition = 100;
```

Now that our variable has been initialized, we can use it. In our example, we use **Xposition** in the **paint** function.

```
g.drawString( "Hello!", Xposition, 25 );
g.drawString( "My name is Inigo Montoya.", Xposition, 50 );
g.drawString( "You killed my father.", Xposition, 75 );
g.drawString( "Prepare to die.", Xposition, 100 );
```

In these four calls to the **drawString** function, we use **Xposition** as the X coordinate parameter. So, all four strings will be drawn starting at X coordinate 100.

That's how we use variable in Java. First we declare them. Then we initialize them to a starting value. Then we use them.

## Arrays

We've already seen variables in action. They can store a single value. Sometimes, though, we need to work with a list of values. We do this using arrays. Using the **for** loop and arrays, Java makes it easy to manage lists of items.

This program shows arrays in action"

```
import java.awt.*;
import java.applet.*;

public class BoxesExample extends Applet
{
    int Boxes[];        // Declare an array of integers

    public void init ()
    {
        resize (320, 50);
        Boxes = new int[5];
        for ( int i=0; i<5; i++ )
        {
            Boxes[i] = 0;
        }
        Boxes[2] = 1;
        Boxes[4] = 1;
    }

    public void paint( Graphics g )
    {
        for ( int i = 0; i<5; i++ )
        {
            if (Boxes[i] == 1 )
            {
                g.drawString ( "X", i*60 , 25 );
            }
            else
            {
                g.drawString ( "O", i*60 , 25 );
            }
        }
    }
}
```

```
}
```

### [Check this code](#)

This program uses an array to hold a list of five values. An array is just a fancy name for a list of things. Like other variables, arrays need to be declared and initialized before we can use them. The first thing we do in class **BoxesExample** is declare an array of integers:

```
int Boxes[];
```

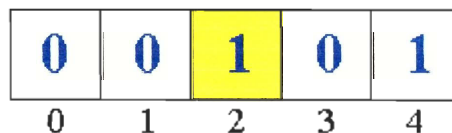
This declares an array of integers named **Boxes**. The pair of brackets [] tells Java that we're declaring an array and not just a variable. The next thing we do is to tell Java how many items we want in the array.

```
Boxes = new int[5];
```

This tells Java that there will be five integers in our array. The next thing we need to do is initialize the individual items in the array. We can initialize them individually, like this:

```
Boxes[0] = 0;  
Boxes[1] = 0;  
Boxes[2] = 0;  
Boxes[3] = 0;  
Boxes[4] = 0;
```

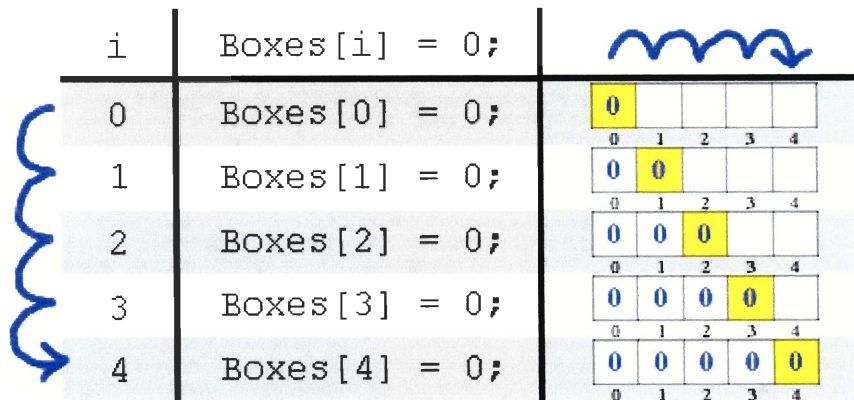
Each of these lines accesses an item from our Boxes array and sets it to a value of 0. We use the brackets [] here to specify which item in the list. The number in the brackets [] is our array index. Index's start from 0 and go to 1 less than the size of the array.



For example, array index 2 refers to the 3rd item in our list. But initializing our array one item at a time is a bit tedious. So let's look at our old friend the **for** loop.

```
for ( int i=0; i<5; i++ )  
{  
    Boxes[i] = 0;  
}
```

As we know, the example above will cause the code in the **for** loop to repeat 5 times. The variable **i** can be used within the loop as a counter. This counter starts at 0 and then 1 is added to it each time the loop repeats.



In the **paint** function, we use a **for** loop to step through our array again:

```
for ( int i = 0; i<5; i++ )
{
    if (Boxes[i] == 1 )
    {
        g.drawString ( "X", i*60 , 25 );
    }
    else
    {
        g.drawString ( "O", i*60 , 25 );
    }
}
```

This time we use the **for** loop to check the value of each item in our **Boxes** array. When **Boxes[i]** is equal to 1, (**Boxes[i] == 1**), it draws an X, otherwise it draws an O.

Our **i** variable is also used to position the X's and O's. We use **i\*60** as the X coordinate parameter for the **drawString** function. We can multiply (\*), divide (/), add (+), or subtract (-).

## Two Dimensional Arrays

Although arrays are good for lists, the squares in a Tic Tac Toe game don't fit well into a simple list. What we need is a table that has rows and columns. We'd like to look at each square using its row and column number. A table, or two- dimensional array, allows us to do that.

This program uses a 2D array to determine where to put X's in a 4 X 4 grid.

```
import java.awt.*
import java.applet.*

public class FourByFour extends Applet
{
    int Jeep[][];
```

```

public void init()
{
    resize(200, 200);
    Jeep = new int[4][4];
    for (int row=0; row<4; row++)
    {
        for (int column=0; column<4; column++)
        {
            Jeep[row][column] = 0;
        }
    }
    Jeep[0][0] = 1;
    Jeep[3][3] = 1;
    Jeep[1][2] = 1;
    Jeep[2][1] = 1;
}

public void paint( Graphics g )
{
    g.drawLine( 0, 50, 200, 50);
    g.drawLine( 0, 100, 200, 100);
    g.drawLine( 0, 150, 200, 150);

    g.drawLine( 50, 0, 50, 200);
    g.drawLine( 100, 0, 100, 200);
    g.drawLine( 150, 0, 150, 200);

    for (int row=0; row<4; row++)
    {
        for (int column=0; column<4; column++)
        {
            if (Jeep[row][column] == 1)
            {
                g.drawString("X", column*50 + 20, row*50 +
20);
            }
        }
    }
}
}

```

[Check this code](#)

The first thing we do is declare a 2D array like this:

```
int Jeep[][];
```

The two pair of brackets [][] tell Java that **Jeep** is a two-dimensional array. We create **Jeep** almost the same way we created one dimensional arrays.

```
Jeep = new int[4][4];
```

The only difference is the second set of brackets. The first number in brackets is how many rows are in the table. The second number is how many columns. Here, we make **Jeep** a 4 X 4 array.

Since tables have two dimensions, items inside them are accessed using two indices instead of one index.

```
Jeep[1][2] = 1;
```



This sets the value in row 1, column 2 of **Jeep** to 1. Initializing a 2D array line by line like this would be very tedious. We learned how to initialize arrays using a **for** loop. To initialize 2D arrays, we have to use two **for** loops.

```
for (int row=0; row<4; row++)
{
    for (int column=0; column<4; column++)
    {
        Jeep[row][column] = 0;
    }
}
```

Because **Jeep** has two dimensions, we need two loops, one inside the other, to step through each item in the table. We couldn't use **i** for both loops here, because Java wouldn't know which **i** to use. Since we're using the loop counters for row and column numbers, we call them **row** and **column**.

In the **paint** function, we step through the array using two **for** loops.

```
for (int row=0; row<4; row++)
{
    for (int column=0; column<4; column++)
    {
        if (Jeep[row][column] == 1)
        {
            g.drawString("X", column*50 + 20, row*50 + 20);
        }
    }
}
```

Here we use the two **for** loops to check every item in our table to see if it equals 1. If this is true, we draw an X. Like with one dimensional arrays, we use math to position our X.

When we draw the X's, **row** becomes the Y coordinate and **column** becomes the X coordinate. The math just helps us make things look good. Java follows the same order of operations as algebra. It multiplies and divides, and then adds and subtracts.



# Advanced Java Applets

## Event Handling

Every time a mouse button or keys on the keyboard are pressed or released, a special signal, called an event, goes off. The Applet class skeleton contains functions that are run automatically when these events occur. If we want our applet to do something special when one of these events occurs, we just need to override the right function.

To show events in action, this program shows an 'X' in the center of the screen applet window, and then alternates between 'X' and 'O' with each click of the mouse button.

```
import java.awt.*;
import java.applet.*;

public class XsAndOs extends Applet
{
    // declare variables
    int CurPlayer;

    public void init ()
    {
        resize(100,100);
        CurPlayer = 1;
    }

    public void paint( Graphics g )
    {
        if (CurPlayer == 1)
        {
            g.drawString ("X", 45, 50);
        }
        else
        {
            g.drawString ("O", 45, 50);
        }
    }

    public boolean mouseUp (Event evt, int x, int y )
    {
        repaint();
        if ( CurPlayer == 1 )
        {
            CurPlayer = 2;
        }
        else
        {
            CurPlayer = 1;
        }
        return true;
    }
}
```

[Check this code](#)

We use the **CurPlayer** variable to keep track of the player whose turn it is.

In the **init** function, we set up a 100 by 100 applet window, and initialize the **CurPlayer** variable to 1. The **paint** function uses a couple of **if** statements to check the value of the **CurPlayer** variable. It draws an 'X' if **CurPlayer** is 1, or an 'O' otherwise.

What's really new here is the **mouseUp** function.

```
public boolean mouseUp (Event evt, int x, int y )
```

The **mouseUp** function is a part of the applet skeleton we haven't seen before. It's called automatically whenever someone lets go of the mouse button.

The parameters to **mouseUp** are an **Event**, called **evt**, and two integers, which are the x and y coordinates of the mouse pointer when the button was released. We don't use them for anything in our example program, but in our **TicTacToe** program we will use **x** and **y** to check which square players click the mouse over.

First, our **mouseUp** function calls the **repaint** function.

```
repaint();
```

The **repaint** function is used to force Java to redraw our applet window. In redrawing the applet window, Java calls **paint** automatically.

The next thing our **mouseUp** function does is update the **CurPlayer** variable, switching it to 2 if it was 1 and changing it back to 1 otherwise.

```
if ( CurPlayer == 1 )
{
    CurPlayer = 2;
}
else
{
    CurPlayer = 1;
}
```

Finally, Java expects **mouseUp** to return **true**.

```
return true;
```

## Logical Operators

We already know that we use **if** statements to check if a condition is true or false. We can use logical operators to construct complex conditions. These complex conditions are a much more powerful tool. They let us create very specific **if** statements. For our Tic Tac Toe applet to determine a winner, we need it to utilize complex conditions.

This program divides the applet window into three boxes. When the mouse is clicked over a box, an X is drawn in it. And when all three boxes show an X, it displays a message.

```
import java.awt.*;
import java.applet.*;

    // This applet will use logical operators to check for a
    // winner. The user must click once on each box in the
    // grid.

class LogicExample extends Applet
{
    int Boxes[];          // Declare an array of integers

    public void init ()
    {
        resize(240,120);
        Boxes = new int[3];

        for ( int i=0; i<3; i++ )
        {
            Boxes[i] = 0;
        }
    }

    public void paint( Graphics g )
    {
        for ( int i = 0; i<3; i++ )
        {
            if (Boxes[i] == 1 )
            {
                g.drawString ( "X", i*80 , 60 );
            }
        }

        if ((Boxes[0] == 1 ) &&
            (Boxes[1] == 1 ) &&
            (Boxes[2] == 1 ))
        {
            g.drawString ("We have a Winner!", 10, 30);
        }
    }

    public boolean mouseUp ( Event evt, int x, int y )
    {
```

```

int index = x/80;
for ( int i = 0; i<3; i++ )
{
    if (i == index )
    {
        Boxes[i] = 1;
    }
}
repaint();
return true;
}
}

```

[Check this code](#)

Everything should look familiar except for the **&&** in the **paint** function. **&&** is a logical operator. Logical operators are used to combine the results of conditions. Let's look at some logical operators:

Syntax	Meaning
<i>Condition1 &amp;&amp; Condition2</i>	AND - Both conditions must be satisfied for the statement to be true.
<i>Condition1    Condition2</i>	OR - Either one or both conditions must be satisfied for the statement to be true.

In the program above, we use logical operators to check if there are Xs in all three positions.

```

if ((Boxes[0] == 1 ) &&
    (Boxes[1] == 1 ) &&
    (Boxes[2] == 1 ))

```

The complex conditional statement here checks if Boxes[0] AND Boxes[1] AND Boxes[2] are all equal to one. The if statement is true only if all three conditions are true (there's an X in all three positions). If it's true, the message "We have a Winner!" is displayed.

## Graphics

We've already seen the **drawString** and **drawLine** functions from our **Graphics** toolbox. But if we want to make our Tic Tac Toe game look better, we'll need a few more things.

This applet demonstrates a few of them. It changes the background color, changes the drawing color, changes the pen width, and then draws a lowercase 'a' using a line and circle.

```
import java.applet.*;
import java.awt.*;

public class Ademo extends Applet
{
    public void init()
    {
        resize(80, 120);
    }
    public void paint(Graphics g)
    {
        g.setColor (Color.blue);
        g.fillRect ( 5, 5, 70, 110);

        g.setColor (Color.red);
        g.drawOval ( 10, 50, 60, 60);
        g.drawLine ( 70, 110, 70, 40);

        g.drawArc ( 10, 20, 60, 60, 0, 180);
    }
}
```

[Check this code](#)

Let's quickly go through the new functions here:

---

Function	Explanation
<code>setColor (Color c)</code>	Changes the drawing color for functions using the current graphics toolbox to whatever <code>c</code> is. Class <b>color</b> predefines a number of colors.

---

```
fillRect (int x, int y, int
width, int height)
```

Draws a solid rectangle with an upper left-hand corner of x, y. Its width is equal to width; height is equal to height.

```
drawOval (int x, int y, int
width, int height)
```

Draws an oval with an upper left-hand corner of x, y. Its width is equal to width; height is equal to height.

```
drawArc (int x, int y, int
width, int height, int
startAngle, int EndAngle)
```

Draws a portion of an oval much like **drawOval**. **startAngle** and **endAngle** are in degrees; 0 is the 3 o'clock position.

---

## Adding Sound

One of the cooler things about Java is its built in multimedia capabilities. Graphics, sound, and internet capabilities are built into the language. Java also has things like buttons and menus built in.

This applet plays a sound file whenever one clicks on the applet.

```
import java.applet.*;
import java.awt.*;

public class Radio extends Applet
{
    String Status;

    public void init()
    {
        resize(100, 100);
        Status = "Click to play";
        setBackground ( Color.cyan);
    }
    public void paint ( Graphics g)
    {
        g.drawString ( Status, 5, 40 );
    }
    public boolean mouseUp(Event evt, int x, int y)
    {
        play(getCodeBase(), "audio/joy.au");

        Status = "Sound clip played";
        repaint();

        return true;
    }
}
```

```
}  
}
```

[Check this code](#)

Let's go through the new things here:

---

Function	Explanation
<code>String Status;</code>	Create a <b>String</b> variable called Status. A string is just a collection of letters, numbers, and spaces.
<code>setBackground (Color c)</code>	Changes the background color of the applet.
<code>play (URL location, String file)</code>	Play an audio (.au) clip named <i>file</i> using <i>location</i> as the base URL.
<code>getCodeBase()</code>	Returns the URL where the applet is stored.

---

## Appendix B - Activities

### Lab 1 - Creating Web Pages

Activity 1.1

Activity 1.2

Activity 1.3

Activity 1.4

### Lab 2 - JavaScript

Activity 2.1

Activity 2.2

Activity 2.3

### Lab 3 - Perl/CGI

Activity 3.1

Activity 3.2

Activity 3.3

### Lab 4 - Java Applets

Activity 4.1

Activity 4.2

Activity 4.3



# Activity 1.1

Create a directory in your home directory named `public_html`.  
Set the permissions on that directory to `rxw r-- r--`.

Create a dummy file in that directory named `index.html` using the `touch` command.  
Set the permissions on that file to `rxw r-- r--`.

Check the permissions using `ls -l filename or directory`

## Activity 1.2

Using HTML, make a personal homepage for yourself by editing  
`~/public_html/index.html`.

This page should include:

- a title
- a heading
- horizontal line(s)
- a link to "<http://penguin.wpi.edu:4546/course/web>"
- any additional links you would like to add

Your page can be viewed in any web browser by going to the location:  
"<http://users.wpi.edu/~username>".

## Activity 1.3

Change the background of your homepage with the `background` or the `bgcolor` attributes for the `body` tag.

Write a short paragraph about yourself. For example, you could discuss what your hobbies are, what high school you go to, what movies you like or have seen recently. Use the `font` tag on this paragraph and change the `size`, `color`, and `face` attributes however you like.

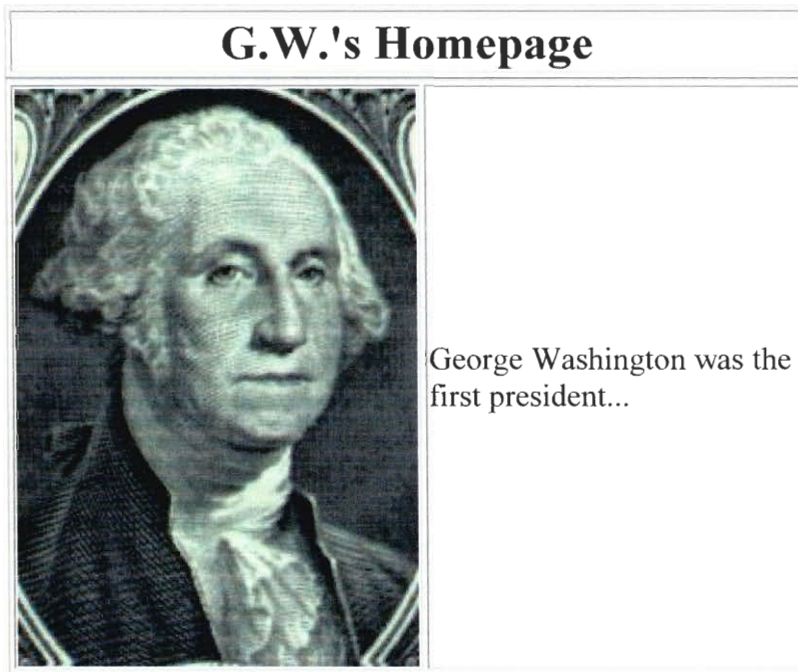
Add a picture of yourself to your homepage. Be sure to set the `alt` tag. Set the `align` and `border` however you like.

Include a list containing WPI building names, your group member names, or anything else.

## Activity 1.4

Use a table to format all the data in your homepage however you like.

Example:



## Activity 2.1

Create a Lab 2 page at `~/public_html/lab2.html`. In this file, imbed a JavaScript which displays some welcome message in an alert box when the page is loaded.

Examples:

```
"Caution, you are entering your name's Lab 2 page."  
"Welcome to your name's Lab 2 page."
```

Include a link to `lab2.html` in your homepage (`index.html`).

## Activity 2.2

In your Lab 2 page, after the alert box from Activity 2.1 is displayed, display a confirm box with some message. Depending on the users choice, (OK or CANCEL), display a different message of the page using the `document.write()` function.

Example:

"Do you agree with the terms and conditions of the agreement?"

If OK, display "You agree and can continue setup."

If CANCEL, display "You disagree and must exit setup."

## Activity 2.3

In your Lab 2 page, make an HTML form with three text areas, a button, and a fourth text area labeled "Result:". The user should be directed to input data in the first three boxes and then click on the button. When the button is clicked, a JavaScript function is called and the values of those 3 text areas are passed to the function as parameters. These values should be stored in an array and displayed, in reverse order, in the "Result:" text box.

Use examples given in [Advanced JavaScript](#) as guidelines.

## Activity 3.1

Write a perl script that prompts the user for the following:

1. user name
2. course name
3. first quiz score
4. second quiz score
5. third quiz score
6. fourth quiz score

The script should then calculate the user's average and display the following message:

"`user name`'s average for `course name` is `average quiz score`."

## Activity 3.2

Make an HTML form which asks the user for various personal information. Example:

- First Name
- Middle Initial

- Last Name
- Street Address
- City
- State
- Zip
- Email Address

On submission of this form, the information is sent to a CGI script using the POST method. The information should be sent back to the user by the CGI script in a formatted table.

There should be a minimum of 5 fields to be filled in.

### **Activity 3.3**

Use the form you made in [Activity 2.3](#). On submission of this form, instead of being formatted into a table, the information should be emailed to the owner of the site.

(Chapter 8, Hands-On Exercises #5) Create an HTML form that allows end users to specify a preferred background color when visiting your Web site. Notify end users that only red, yellow, green, and white can be used (and enforce that limitation). When a user selects a color, generate another screen that thanks the Web site visitor. Set the background to that color (if a valid one was selected) and set a cookie with this preference. Use this color when the end user visits your site again.

## Learning Java - Module 1

### Project 1: Drawing the Board

Now we are ready to start writing our Tic Tac Toe game. Let's start by drawing the board..

Here's the skeleton of our Tic Tac Toe game.

```
import java.awt.*;
import java.applet.*;

public class TicTacToe extends Applet
{
    public void init ()
    {
        // set up the game
    }
    public void paint( Graphics g )
    {
        // draw the board
    }
}
```

1. Fill in the **init** function so that it resizes the applet window to be 150 pixels wide and 200 pixels high.
2. Fill in the **paint** function so that it does the following:
  - draw the Tic Tac Toe grid on the screen using the **drawLine** function. The board should create squares that are 50 x 50 pixels.
  - use the **drawString** function to draw an "X" in the middle square
  - use the **drawString** function to draw "Game Ready" below the board.

The program should look like [this](#) when it's done.

[Back](#) - Step 3: The init Function and Variables

[Next](#) - Module 2: Xs and Os

---

## **Web References**

[Java Reference Guide](#)  
[Compiling Java Programs](#)

---

## **Learning Java Credits**

## Learning Java - Module 2 Project 2: The X and O Hotel

Before we go on to the final module, let's use what we've learned to add to our Tic Tac Toe applet. It needs a way of looking at each square individually. So let's make a few changes to *TicTacToe*. Here's a revised skeleton.

```
import java.awt.*;
import java.applet.*;

public class TicTacToe extends Applet
{
    // declare variables

    public void init ()
    {
        // set up the game
    }
    public void paint( Graphics g )
    {
        // draw the board
        // draw X's and O's in the right place
    }
}
```

1. Declare a 2D array of integers called *Board*.
2. In the **init** function
  - a. Make *Board* a 3 X 3 array.
  - b. Initialize each element in *Board* to 0 by using two **for** loops.
  - c. Set value of the center square to 1.
  - d. Set value of the upper left-hand square to 2.
3. In the **paint** function
  - a. Remove the code that drew one 'X' in the center of the screen.
  - b. Use two for loops to step through *Board*, drawing an 'X' if the current square is 1, an 'O' if the current square is 2, and nothing if the current square is 0.

**Hint:** Take a close look at the program from Module 2, Step 3. It does almost everything that needs to be added to *TicTacToe*.

[Take a look](#) at what your applet should look like.

---

[Back](#) - Step 3: 2D Arrays

[Next](#) - Module 3: Making it all work

---

### **Web References**

[Java Reference Guide](#)  
[Compiling Java Programs](#)

---

**Learning Java Credits**



## Learning Java - Module 3 Project 3: Playing the Game

Now that we know events, we can add some interactivity to our Tic Tac Toe program.

Here's an updated skeleton our *TicTacToe* program.

```
import java.awt.*;
import java.applet.*;

public class TicTacToe extends Applet
{
    // declare variables

    public void init ()
    {
        // set up the game
    }
    public void paint( Graphics g )
    {
        // draw the board
        // draw X's and O's in the right place
    }

    public boolean mouseUp( Event evt, int x, int y)
    {
        // update board and switch players (if appropriate)
    }
}
```

Make the following changes to your program from [Project 2](#).

1. Add an integer variable named *CurPlayer* to keep track of whose turn it is.
2. In the **init** function, initialize *CurPlayer* to 1.
3. Add a **mouseUp** function to your TicTacToe program. It should do the following:
  - a. Divide the *x* and *y* parameters by 50 to get column and row numbers.
  - b. Check if the square clicked is a legal place for the current player to move (there isn't already an X or an O there).
  - c. If it is a legal move, update *Board*, then change the current player, and repaint the applet.

**Hints:**

1. You're going to need to convert the x and y pixel coordinates in **mouseUp** to row and column numbers. You can use these numbers to access the item in your array that represents that square. You might want to use variables to store the results.
2. You're going to need a way to keep track of the difference between empty squares, X's, and O's. Use 0 for empty, 1 for "X", and 2 for "O".

When you're done with this, you'll have a working game.

Here's the [game in action](#).

---

[Back](#) - Step 1: The Main Event

[Next](#) - Step 2: Vulcan Logic

---

### Web References

[Java Reference Guide](#)  
[Compiling Java Programs](#)

---

[Learning Java Credits](#)

## Appendix C - Student Evaluations

Pre/Post-Test

Quiz 1

Quiz 2

Quiz 3

Quiz 4

## Pre/Post Test

### Client/Server Basics

1. Web programming depends heavily on the Client/Server Model. In this model, what is the web browser?
  - a) Shell
  - b) Server
  - c) Client
  - d) Directory

### UNIX Basics

2. Which command is used to change file permissions in UNIX?
  - a) finger
  - b) man
  - c) cp
  - d) chmod

### HTML

3. Which tag is used to define a link?
  - a) img
  - b) a href
  - c) link
  - d) html
4. Which is not an attribute for the <body> tag?
  - a) basefont
  - b) background
  - c) text
  - d) link
5. Which tag is used to define a cell in a table?
  - a) tr
  - b) table
  - c) td
  - d) th

### JavaScript

6. Which command is used to display text?
  - a) print

- b) cout
- c) System.out.println
- d) document.write

7. When comparing values, which operator is used to test for equality?

- a) =
- b) ==
- c) :=
- d) !=

8. If you had an array named array\_name, how would you set the first element in that array equal to 0?

- a) array\_name.1 = 0
- b) array\_name.0 = 0
- c) array\_name[1] = 0
- d) array\_name[0] = 0

## Perl/CGI

9. Why is Perl a good language to use with CGI? More than one answer.

- a) replacement capabilities
- b) powerful pattern matching capabilities
- c) object oriented language
- d) it is the only language that can be used with CGI

10. What hash function is used in Perl to determine if a hash is empty?

- a) scalar
- b) exists
- c) empty
- d) hash

11. Which perl command is used to replace a pattern?

- a) m/pattern/replacement
- b) s/pattern/replacement
- c) switch/pattern/replacement
- d) replace/pattern/replacement

## Java Applets

12. Which command would you use to compile a java program?

- a) java
- b) compile
- c) javac
- d) javacompile

13. Which of the following attributes is required in the applet tag?

- a) name
- b) codebase
- c) width
- d) align

14. Which of the following is not a mouse event?

- a) mouseDown
- b) mouseOver
- c) mouseExit
- d) mouseDrag

# Quiz 1

## Multiple Choice

1. Web programming depends heavily on the Client/Server Model. In this model, what is the web browser?
  - a. shell
  - b. server
  - c. client
  - d. directory
2. Which UNIX command is used to delete a file?
  - a. delete
  - b. del
  - c. remove
  - d. rm
3. The img tag is used in HTML to display images. Which is a required attribute for the img tag?
  - a. border
  - b. image
  - c. src
  - d. picture
4. Which tag is used to define a cell in a table?
  - a. tr
  - b. table
  - c. td
  - d. cell

## Short Answer

5. Write an HTML file which would display the following:

This text is normal.

**This text is bold.**

*This text is italicized.*

**This font size is 5.**

`This font face is courier.`

**This font color is red.**

## Quiz 2

### Multiple Choice

1. What HTML tag do you use to tell the browser there is JavaScript in the file?
  - a. `<script language="javascript"> </script>`
  - b. `<language="javascript"> </language>`
  - c. `<javascript> </javascript>`
  - d. `<js language="javascript"> </js>`
2. What do all lines in JavaScript end in?
  - a. ,
  - b. :
  - c. ;
  - d. .
3. In JavaScript, are the variables Ax and aX the same or different?
  - a. same
  - b. different
4. What is the JavaScript variable that you would use to set a cookie?
  - a. cookie
  - b. javascript.cookie
  - c. document.cookie
  - d. chocolate.chip

### Short Answer

5. Write an HTML file with embeded JavaScript code which does the following:

Displays a confirm box with the message:  
"If you will be a senior this fall, press OK. Else,  
press CANCEL."

If the user presses OK, then display:  
"Congratulations Class of 2003!"

If the user presses CANCEL, then display:  
"You are not graduating next fall."



## Quiz 3

### Multiple Choice

1. Why is Perl a good language to use with CGI? More than one answer.
  - a. replacement capabilities
  - b. powerful pattern matching capabilities
  - c. object oriented language
  - d. none of the above
2. What programming languages can CGI be used for?
  - a. C++
  - b. C
  - c. Java
  - d. all of the above
3. In Perl, are the variables Ax and aX the same or different?
  - a. same
  - b. different
4. When using the GET method for CGI, data is sent to a perl program in the form of one long string of name-value pairs. What is the name of the environment variable which contains this string?
  - a. GET\_STRING
  - b. CONTENT\_LENGTH
  - c. QUERY\_STRING
  - d. ENV\_VAR

### Short Answer

5. What will this print:
  - 6.
  7. `$day = "Hello, Students. Today is a xday, we are going to learn scalar variables.\n";`
  8. `$day =~ s/xday/Wednesday/;`
  9. `print $day;`

## Quiz 4

### Multiple Choice

1. Which keyword is used in the beginning of a java file to get things needed for an applet from other files?
  - a. public
  - b. class
  - c. get
  - d. import
2. Which keyword is used to define a new object?
  - a. class
  - b. extends
  - c. javac
  - d. object
3. Which function is called automatically everytime an applet needs to draw itself?
  - a. paint
  - b. draw
  - c. init
  - d. create
4. Which function is part of the applet skeleton and gets called automatically everytime someone lets go of a mouse button?
  - a. mouseRelease
  - b. mouseUp
  - c. mouseClicked
  - d. letGo

### Short Answer

5. How would you initialize the following 2 dimensional array named `mulTable` using 2 for loops?

<b>mulTable</b>	<b>col 1</b>	<b>col 2</b>	<b>col 3</b>	<b>col 4</b>	<b>col 5</b>
<b>row 1</b>	1	2	3	4	5
<b>row 2</b>	2	4	6	8	10
<b>row 3</b>	3	6	9	12	15
<b>row 4</b>	4	8	12	16	20
<b>row 5</b>	5	10	15	20	25

## Appendix D - Frontiers Qualifying Projects

Stock Ticker Project

Web Browser Project

E-Commerce Site

Cryptography Project

Meet the Sticks

Create a Fractal Interface

# Stock Ticker Program

This project involves:

- **Perl/CGI**
  - Exposure to real world HTML on the Web.
  - Techniques used in web page layout and design.
  - HTML parsing with [regular expressions](#) in Perl.
  - Working with HTML forms.
  - Maintaining state on the Web.
- 

## Level 1

In this assignment you will develop a simplified Web client. Your cgi script will make requests to a remote Web server for an HTML page which contains stock information. To simplify this process, you may use the functions defined in [LWP::Simple](#). Here is a sample URL of a yahoo page with a single stock:

```
http://finance.yahoo.com/q?s=csc&d=t
```

You will then need to write a web page parser which uses regular expressions to go thru the HTML page. You should locate and extract the following specific information:

- company name
- price
- change
- image chart URL

You then need to interpret that stock information and format it into an HTML page which is then returned to the user. (The image chart itself should be displayed, not the URL.)

---

## Level 2

Your cgi script should have the functionality of the previous level.

Your script should start by displaying an HTML form. The form should have a text area, checkboxes, a submit button, and a clear all button. The text area will be used to enter stock symbols. The checkboxes will be used to select what data should be displayed. The submit button should submit the form to the cgi script. The clear all button should reset your input fields to the original values. The form should look **similar** to this:

Enter Symbols:

price

change

image chart

---

The script should be able to make requests and provide results for multiple stock symbols. Here is a sample URL of a yahoo page with multiple stocks:

<http://finance.yahoo.com/q?s=cscot+a&d=t>

You should then add functionality to allow the user to select the following information to be displayed as well:

- volume
  - average volume
  - day's range
  - 52 week range
  - market capitalization
- 

### Level 3

Your program must have Level 2 functionality. Also, you must implement a welcome page, where the user can enter a user name and a password to login onto the Web site. If the user is new to the site, they should be able to create a new account and enter at least the following information:

- first name
- last name
- street address
- city
- state
- zip code
- e-mail
- user name
- password
- confirm password

After a successful login or account creation, the user should be allowed to lookup stock symbols. For this, you must write an extended cgi script that can create the new account,

ensure unique user names, and store the account information in either a flat ASCII file or by using functions in the [Storable](#) module. Also, your HTML forms should use the POST method so that account information (especially passwords) can not be viewed in the URL.

---

## Level 4

You must implement the functionality of a Level 3 assignment. Your cgi script needs to maintain state between Web pages so that for every request that it receives, it knows who the user is. To do this you will use cookies. You must ensure that all cookies do not contain real data, such as login information. Here is an example:

- A user visits your site for the first time or from a different computer than previously.
- Your cgi script realizes that the user is new and replies with a welcome page containing a login prompt and create new account prompt.
  1. **On login:** The cgi script looks you up in the database and replies with either an error page or a stock lookup page.
  2. **On create account:** The cgi script validates the information, an entry is made in the database, and replies with either an error page or a stock lookup page.
- Any request for stock data that does not contain a valid cookie should be answered with an error page that also allows the user to enter a different login or create a new account.

# Web Browser

This project involves:

- **Perl/CGI**
  - Exposure to real world HTML on the Web.
  - HTML parsing with [regular expressions](#) in Perl.
  - Working with HTML forms.
  - Maintaining state on the web.
- 

## Level 1

In this assignment you will create your own Web browser. The first thing you need to do is think of a catchy and marketable name for your Web browser. (Examples: Internet Explorer, Netscape, Mozilla, Opera) Then you must build an HTML page which contains two frames. In one frame you should put an HTML form which contains a text box. This text box will be used to input the name of a Web document. There should also be a submit button. When the submit button is clicked, your cgi script should retrieve the document and display it in the second frame. You can use the functions in [LWP::Simple](#) module.

---

## Level 2

You must fully implement a Level 1 assignment. In addition, you should use cookies to implement the following navigation buttons:

- back
  - forward
  - home
  - set current page to home
- 

## Level 3

You must fully implement the Level 2 assignment. In addition, when your cgi script loads an HTML page, it should parse the document to locate references to other files such as images and links. These references may be absolute URLs, relative to the current directory, or relative to the path specified in a `base` tag. Your program should convert all of these to absolute URLs before it gets displayed. This will display the images and allow links to work properly.

## Level 4

You must fully implement the Level 3 assignment. Your first frame should contain another button which allows you to store current web locations in a bookmarks list. The bookmarks list should be displayed in the first frame as a drop down menu. When a bookmark is selected, your cgi script should load it in the target frame.



---

## Create Your Own E-Commerce Site

---

The goal of the project is to create an e-commerce-"like" application using a CLIENT-SERVER architecture, similar to many seen on the web. The CLIENT is a web page running on a browser that accepts orders from buyers and sends them to the SERVER. The SERVER returns information to the browser, which displays the information to the user as a current shopping cart. The shopping cart will be created dynamically by a CGI script. The CGI script will be written in Perl for the server side and HTML (with forms) for the client side. The project will guide you in learning some Perl code and extra HTML code. The project, when done, will "act" like an e-commerce site. It is up to you how "professional" you would like the site to look. Choose your own e-commerce business along with products and prices.

### PROCEDURE:

1. [Specification](#)
2. [Example Implementation](#)
3. [Creating Forms](#)
4. [Creating CGI with Perl](#)
5. [Using Perl](#)
6. [Finalizing Project](#)

*You should write:*

- [A home page for the store](#)
- [Department pages for the various departments of the store \(at least 3 departments\)](#)
- [A CGI script that dynamically generates a shopping cart](#)
- [A summary finalized page](#)

The **home page** has buttons leading to the department pages and the shopping cart page.

The page for a **department** gives, for each product, its name and price, and allows the user to specify the desired quantity of product in a text field. It has a button for adding the selected products to the cart and a button for returning to the home page.

The **shopping cart** page gives the list of products selected so far by the user and their quantity. It has buttons to finalize the purchase, return to the home page, or reset the current shopping cart.

A **summary finalized page** finalizes the order (empties the cart) and prints out on the screen the total purchases of the user in an easy-to-read format consisting of product name, quantity ordered, price per quantity, and total price for that product. Somewhere there has to be a total price for the entire order.

You have to maintain separate shopping carts for each user. For this assignment we can assume one user per host name (this is not the case in the real world but it makes it much easier)

*Extra Work to make it cool:*

Give the user the ability to change their order in the **shopping cart page**. Make all the **department pages one CGI script page** (a lot harder than it sounds).

Create a **login page**, which can be accessed from the home page. You will have to keep track of the user using hidden tags throughout the pages. There will have to be a data file of users as well as a separate **CGI home page** to keep track of the user (this could be fun). Remember to allow a new user to login. You will probably want at least name, password, address and e-mail.

# Cryptography Project - Ciphertext Decryption

Project Advisor: ?

## **Introduction**

The FBI's [Carnivore](#) "email snooping system" is only one of many concerns regarding the integrity and privacy of personal electronic communications today . The popular use of electronic communications (email, instant messaging services, web-based financial transactions, etc) makes it easier than ever for an outsider to intercept supposedly private communications between individuals, organizations, businesses and governments.

One method of securing electronic communications is encryption. Many styles of encryption exist, dating back to the beginning of human history: the Julian shift cipher (used by Julius Caesar himself) and the Enigma system (used extensively by the German military, and cracked by the Allies, in World War II) are just two examples of encryption systems. However, one encryption system that is most readily applicable to electronic communications is asymmetric-key encryption.

Probably the most well-known asymmetric-key encryption system is Pretty Good Privacy (PGP), written by Phil Zimmerman, and published online (for free, yeah!) in the early 1990s. The underlying mathematical equations and mechanisms of PGP are so strong that the system was classified as a national security concern. Because PGP was available all over the world via the Internet, Zimmerman was charged with violation of national security by the American government. In some places "violation of national security" is called treason, and is punishable by death. (If you're still unconvinced of the seriousness of encryption export law, check out [www.cryptography.com](http://www.cryptography.com).)

Good encryption systems, when used properly, make communications more difficult to be read by a third party. However, as the German military learned in World War II, no encryption system is unbreakable. Your task in this project, should you choose to accept it, is to play the "unauthorized observer" role. Given an intercepted message, you must crack the code and steal your opponent's secrets. (Note: the US government's National Security Agency [NSA] handles this sort of role all the time. The NSA handles the majority of the American government's intelligence workload, is the world's largest employer of theoretical mathematicians and cryptologists, and measures its computing power in terms of "How many acres of NSA basement floor space are filled with supercomputers.")

## **Background**

[Author's note: It would be beneficial to refer to the glossary given below while reading through the background material.]

Suppose Alice wants to send a message to Bob over an insecure network (the Internet, for example), while Oscar is listening in. Alice strongly encrypts her message to prevent

Oscar from reading the contents, but Alice must make it relatively easy for Bob to decrypt the message. One way that this can be done easily over the Internet is by using an asymmetric-key encryption system.

*Here's how asymmetric-key encryption works:* Bob has two "keys," which are really just small collections of numbers. Bob has a public key (consisting of a very large number:  $z$ , and another smaller number called the encryption key:  $n$ ).

$$k[\text{pub}] = (z, n)$$

Bob also has a private key that consists of three numbers.  $p$ ,  $q$ , and  $s$ .  $p$  and  $q$  are both prime numbers, and have the property  $p \cdot q = z$ .  $s$  is another number, called the decryption key.

$$k[\text{prv}] = (p, q, s)$$

[Please note: the encryption and decryption keys,  $n$  and  $s$ , are not just "some random numbers." There is more to it than that. However, the mathematics is a bit complicated to discuss here. A more mathematically detailed description of asymmetric-key encryption is available at Professor Koeller's [algorithms page](#).]

When Alice wants to send an encrypted message to Bob, she gets his public key (containing  $z$  and  $n$ ) from Bob's website. Alice also has her plaintext message,  $m$ . Alice encrypts her plaintext message,  $m$ , into ciphertext,  $c$ , with the following equation:

$$c = m^n \bmod z$$

Alice then sends her encrypted message over the Internet to Bob. Bob receives the encrypted message,  $c$ , and turns it back into the original plaintext with the equation:

$$m = c^s \bmod z$$

Notice that this equation uses information available only in Bob's private key:  $s$ . For the unauthorized observer to obtain  $s$ , two things are required:  $p$  and  $q$ .  $p$  and  $q$  are plainly available in Bob's private key (which you don't have access to), but they can be obtained by factorizing  $z$ , which is available from Bob's public key (which you do have). [Note: Getting  $s$  from  $p$  and  $q$  is surprisingly simple. A process called the Extended Euclidean Algorithm does this quite nicely. Don't worry about the specifics of the Extended Euclidean Algorithm. A function called "exteuclid" will be provided for you. It will take two integer arguments,  $p$  and  $q$ , and will return  $s$ .]

The strength of asymmetric-key encryption systems rests mostly on properties surrounding prime numbers. Go ahead and solve the following two problems on your calculator:

- 1) What is  $13 \times 7$ ?

- 2) What are the prime factors of 133?

The numbers 7, 13, and 19 are relatively small, as prime numbers go. The next two problems use numbers that are a bit larger. If you'd like to attempt them, go ahead, but don't spend too much time on #4.

- 3) What's  $421 \times 509$ ?
- 4) What are the prime factors of 317461?

Notice that multiplying two prime numbers together is relatively simple, but finding the prime factors of a number (that is, "factorizing" the number) proves to be quite difficult. While multiplication is a simple mathematical and computational process, factorization is far more difficult.

Consider this: factorizing a standard PGP public key (which is a number about 200 digits long) takes about one year of constant processing power (almost 9000 hours of nonstop processor operations) on a computer a few hundred times as powerful as your home computer. And PGP is weak, compared to the cryptographic systems used by businesses and, especially, governments and their agencies.

As you'll be playing the unauthorized observer role in this project, you'll be given the ciphertext of Alice's message to Bob (which can be easily "snooped" from the Internet), and Bob's public key (which he freely posts on his website, so people can send him encrypted messages). With these two pieces of information, and plenty of programming skill, you'll crack the message.

In this case, cracking an encrypted message involves three steps.

- **Step1** Factorize Bob's public key. The public key will be some large number, which has two prime factors. You must find the two prime factors of Bob's public key by writing a factorization program. For example: Say Bob's public key is the number 713. Your program's output might look something like this:

$713 \% 2 \neq 0$

$713 \% 3 \neq 0 \dots$

$713 \% 23 == 0$ , so 23 is one prime factor of the public key. The other factor must be  $713 / 23 == 31$ .

Bob's public key has successfully been cracked!

- **Step2** Compute the decryption key,  $s$ , by using the Extended Euclidean Algorithm. The Extended Euclidean Algorithm solves the equation  $n*s \bmod \text{theta} = 1$  for the variable  $s$ . Here,  $\text{theta} = (p-1)(q-1)$ . Again, this might seem a bit complicated, but don't worry too much about the specifics of how the equations

work out. You can take my word for it, or you can read through the "lecture slides" part of Professor Koeller's [algorithms page](#) and convince yourself (learning some pretty cool number theory in the process.)

- **Step3** Decrypt the ciphertext message,  $c$ . The equation governing this process is  $m = c^s \pmod z$ , where:

$m$  = Alice's plaintext message

$c$  = Alice's ciphertext message, given in the file cipher.txt

$z$  = the larger number of Bob's public key. Note that this is the number you're already cracked by this point.

Don't worry too much about the interrelations of the above equations. Just realize that your goal is to compute 'm' from the first equation. To do that, you need 4 things:  $c$ ,  $s$ ,  $p$  and  $q$ . The ciphertext,  $c$ , is given to you. You find  $s$  from the Extended Euclidean Algorithm, and you've already found  $p$  and  $q$  by cracking  $z$ .

## Glossary

- Asymmetric-key encryption system - Any encryption system that uses two keys, like a public and private key. PGP is one such asymmetric-key encryption system.
- Brute-force attack - Cracking an encrypted message by using nothing more than raw computing power. Other ways of cracking encrypted communications involve the use of espionage and reverse engineering poorly designed encrypted systems. As you have no spies inside Alice and Bob's organization, and public-key encryption is a good method of securing transmissions, you're only option is to attempt a brute-force attack on the ciphertext.
- Ciphertext - Ciphertext is the encrypted (encoded) message that is sent to your associate. Alice sends her ciphertext to Bob over an insecure communications channel.
- Cracking - Informal for "factoring."
- Encrypt - To convert plaintext to ciphertext.
- Extended Euclidean Algorithm - A relatively simple algorithm, used to compute the decryption key "s" from the components of "z." A function called exteuclid will be provided for you. Exteuclid will take two integer parameters ( $p$  and  $q$ ) and will return  $s$ . (Recall that  $z = p * q$ )
- Factoring - Reducing some non-prime number to its prime factors. For example:

Factoring 12543 yields  $111 * 113$

Factoring 96 yields  $2 * 2 * 2 * 2 * 2 * 3$

Factoring 510510 yields  $2 * 3 * 5 * 7 * 11 * 13 * 17$

- Factorizing - Same as "factoring"

- **Linked List** - A data structure often used in various programs for its relative simplicity and extraordinarily large size. The linked list has a "head" node, which connects to another node, and so on, until the tail node is reached. Each node in a linked list can hold some amount of the same type of information. Thus, you can have a linked list of any single datatype (a linked list of integers, perhaps, for your BigInt).
- **Mod** - Short for modular arithmetic. Also denotes the mathematical operation "modulus." (Ex: "7 modulus 3" is the same as "7 mod 3" is the same as  $7 \% 3$ ) In integer division (where we don't concern ourselves with decimals)  $7 / 3 = 2$ , while  $7 \% 3 = 1$ .  $100 / 2 = 50$ , while  $100 \% 2 = 0$ .  $x \text{ mod } y$  equals the remainder of  $x$  divided by  $y$ .
- **Plaintext** - Plain text message. This is the message you write and want to send to your friend, associate, partner-in-crime, etc. Plaintext is not encrypted. This document, for example, can be considered a plaintext document.
- **Public Key** - Part of an asymmetric-key encryption system. The public key is what someone uses to encrypt a message, so public keys are usually posted on websites, printed on business cards, etc. Asymmetric-key encryption systems are designed so that if an unauthorized observer knows the public key the system is still very difficult to crack. Public keys consist of two numbers,  $z$  and  $n$ .
- **Prime Factors** - Any positive integer is either a prime number (5, 37, and 101, for example), or can be broken down into a string of prime factors ( $125 = 5 * 5 * 5$ ,  $2701 = 37 * 73$ , for example).
- **Prime Number** - (sometimes referred to just as a "Prime") An integer that is divisible only by itself and one.
- **Private Key** - Part of an asymmetric-key encryption system. Only one unique private key can decrypt the messages from its associated public key. The private key is just like your computer password: if it falls into the wrong hands, it spells disaster. Private keys consist of three numbers:  $p$ ,  $q$ , and  $s$ , where  $p * q = z$ , from the public key.

## Goal

Given a ciphertext file, encrypted by a public-key encryption system, build a brute-force attack program to decrypt the ciphertext.

## Requirements

1 - The program will take user-input of the ciphertext filename from the program command prompt, and will display the plaintext to the screen.

2 - As the program may take some time to execute (perhaps a few minutes at most) it would be useful to display some sort of progress report for each iteration of the 'cracking' loop. This doesn't have to be anything fancy; a simple line of text should suffice. One example of this progress report format is:

```
"Cracking 1431000479; 1431000479 % 2 != 0"
```

```
"Cracking 1431000479; 1431000479 % 3 != 0"
```

...and so on, until you determine the number's prime factors.

## ***Improvements/Extensions***

The scope of your program's ability is limited by the largest built-in integer value available in Java. In reality, public-key encryption systems have a target number to crack of more than 200 digits (and in some cases far larger) in length. Your new mission, should you choose to accept it, is to create a new type of integer class called `BigInt`. This new class will be a linked list, with each node in the list holding one digit of the entire number. As nodes in a linked list are dynamically allocated, the theoretical size of a `BigInt` is hindered only by the amount of memory of the computer system on which it is running. To be of use in attacking encrypted messages, you must overload the operators `/` (division), `%` (modulus), and `=` (equality).

Once your new integer type is up and running, you should use it in the cracking program you've already written. You can test your `BigInt` on numbers as small as four or five digits. Once you know the `BigInt` works for these relatively simple cases, go ahead and try "recracking" the given ciphertext. You'll probably note the speed difference with which the ciphertext is cracked by using default variable types, versus the `BigInt` variable type.



---

## Meet the Sticks

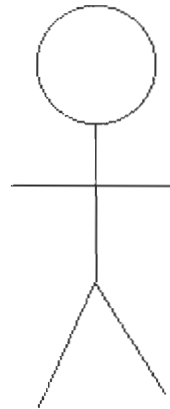
This project's goal is to develop an applet that draws simple stick people. You'll start out by creating a simple applet; then you will move on to creating more complicated ones. The language used will be java.

### PROCEDURE

1. [Create a method called drawPerson](#)
  2. [Write another method called drawFamily](#)
  3. [Add more functionality to drawFamily](#)
  4. Create a nice web page to display your work.
- 

## Meet the Sticks – Part 1

Write a method called **drawPerson**. As the name suggests, this method will draw a person. Not a fancy person - just a stick person, like this:



### *Stick Person*

It is very simple - a circle for a head and lines for the body, arms, and legs. In real life the length of people's arms and legs are usually proportional to their height. This means you should base the size of the body, arms, and legs on how tall the stick person is.

#### Getting Started:

The method header should be:

```
private void drawPerson (Graphics g, int height, int baseX, int baseY)
```

*baseX* and *baseY* should be coordinates located directly between the feet.

*height* is the height, given in pixels

Take a look at the [Pseudocode](#)

## Meet the Sticks – Part 2

Write another method called *drawFamily*. This method will draw a family of two “adults” and two “children.” The *drawFamily* method will call the *drawPerson* method four times, giving appropriate heights and locations of the feet each time.

Getting Started:

Decide on your own method header. Should the number of adults and children be passed as parameters? Should there be parameters that determine the locations of each family member or should it just be one center point and the family mirrored out from there?

Assume that the children are half the height of the parents... or you can have successive calls to *drawPerson* draw at different heights.

Somewhere in your method you will have:

```
public void paint (Graphics g) {  
    DrawFamily(whatever you decide);  
}
```

- Take a look at the [Pseudocode](#)

## Meet the Sticks – Part 3

One of the biggest concepts in “real-world” Computer Science is that all code should be able to be reused, reworked, or have some functionality added to it. This is what you will be doing in Part 3. You will be adding a *width* Parameter to the *drawFamily* method.

In addition to this width parameter you should also add some extra goodies to the applet. Maybe you should add a scrollbar to control the height and another scrollbar to control the width of each person. Each bar, of course, should be labeled appropriately.

Getting Started:

You will have a new method header for draw person:

**private void drawPerson (Graphics g, int height, int width, int baseX, int baseY)**

This will be different from what you created in Part 2.

You also will need to use the AWT to add the scrollbars.

Be sure to implement the *adjustmentListener* appropriate for the scrollbars.

Don't forget to have a function for adjusting the bars.

Look at the [Pseudocode](#)

---

## Create a Fractal Interface

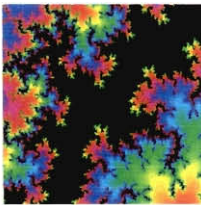
---

### [Complex](#) [Iteration](#) [Fractals](#) [Tools](#) [All Together](#)

In this project you will learn about the creation of fractals, and you will create a series of web-accessible tools to inspect and explore fractals. One important aspect of the project is understanding how the complex numbers work to make the right colors. Another major part of the project is designing a good set of tools for altering the fractals your program makes. When you are done, you should have an applet that everyone can enjoy!

#### PROCEDURE:

1. [Understanding complex numbers](#)
2. [Seeing how iteration works](#)
3. [Making your own fractals](#)
4. [Thinking about tools](#)
5. [Putting it all together](#)



## Understanding complex numbers

Complex Numbers are more interesting than regular numbers - each one has two parts to it.

- First, the *real* part of a complex number is just some regular number, which can have any value, like 0 or 1 or -22 or 3.1415926 or 1/3 or Pi.
- Second, the *imaginary* part is a real number times  $i$ , which is the symbol for an imaginary number. By imaginary numbers we don't mean twentytwo or eleven...  $i$  is the symbol for the square root of -1. There really isn't any way to count the square root of -1 things, so we call it imaginary. Any number times  $i$  is imaginary,

except for one case. Betcha can't guess what we multiply  $i$  by to get -1?

$$i = \text{the square root of } -1$$

so, when we put these two parts together, we get a **complex number** .  
Some examples of complex numbers are

$$c = -0.754 + 0.0491i$$

$$c = 5 + -1.22i$$

$$c = 0.0 + 1i$$

The complex number is the sum of its two parts. You might wonder why we can't just add the parts together, and make the whole thing a regular number, but there really isn't any way to make any imaginary number into a 'normal' number. Complex numbers can't be simplified any more than their two basic parts.

This is a good thing, because for fractals, it's just what you need! When graphing your pattern on the monitor, think of the *real* part of the number as the x coordinate, and the *imaginary* part of the complex number as the y coordinate. Another analogy is with the game of Battleship, where you shout out shots like 4A or 7D. Think of the *imaginary* number as the letter rows, and the *real* number as the number columns. The board on which you would be playing your game of Battleship would be like the *complex plane*. Different Complex numbers point to different places on the complex plane. The real part tells how far to go to the left or right, and the imaginary part tells how far to go up or down.

Complex numbers would be pretty boring if you couldn't change them. We can add, subtract, divide, and multiply complex numbers. The operations required to do this are not as simple as those for regular numbers, because complex numbers have two parts. The way to add a complex number is to add the pieces and put them together. For example, say we wanted to add two complex numbers. First we would add their real parts, and that would be the real part of our answer. Next we would add their imaginary parts to get the imaginary part of our answer. That's it for addition, and subtraction follows the same method (except that you're dealing with negative numbers).

If you understand this, you almost have everything needed to understand how to make a fractal of your own, but first, look a little at [Iteration](#)

## Iteration

Iteration is the process of applying a method of one or more steps to an object many times. For example, say you wanted a well-finished wooden table. You would take an Object, (the rough wooden table) and a Process, (varnish, let dry, sand) and Repeat the process on the object until you have a finished product. The number of times you went through the steps are called iterations.

There are other ways of repeating a process on an object, specifically *recursion*. The difference between iteration and recursion is that in iteration every step is separate, time-wise. In recursion, on the other hand, every step must start after and finish before the start and finish of the step before it. That means the first step lasts as long as all of them, and the second lasts as long as all but one of them, and so on. Iteration is what we want because it is much faster for the computation of the colors.

➤ In this applet, the sidebar is the only control for the iterated line. When you make your Fractal Interface, you need to think about what **Tools** it will need.

Here is a simple applet that replaces a straight line with a line forked into two segments. The sidebar controls the number of iterations. What other things would be fun to change within this applet?

## Making your own fractals

Enough preparation! Now you are ready to learn the equations that give us the Fractals.

The first thing you need are some constants and variables to work with. Let's agree that if you have an X by Y box for your display, that the difference between Xmin and Xmax is the number of pixels wide, and that Ymax - Ymin is the number of pixels high. Each pixel has a complex number attached to it. That number will depend upon your settings, but if you have the boundaries of the fractal, and the width and height of your box, you can know the numerical value of every pixel. This is important because you start with a set of numbers, and finish with a color for a pixel. The variable that holds the value of the color, and also the number of

iterations, (one variable) needs a name as well. You will need a large constant to test the value of your Complex numbers.

Like your tic-tac-toe board, you need to set up loops that find colors for all the pixels, column by column, row by row. At each pixel, you will get a complex number based on your position, this complex number will be multiplied by another complex number iteratively until the number gets far too large, or the color turns black. If the number gets too large, the iterative process stops, and a color is returned for that pixel.

In more detail: you have two complex numbers for every pixel. The first we can call  $X + Yi$  - this is the one that changes for every pixel. The other complex number we can call  $P + Qi$ . Iteratively, we will then multiply their parts, and see how large the first complex number gets. Here's how we find the new X and Y for every iteration.

**the new X = (X\*X) - (Y\*Y) - P**

**the new Y = 2\*X\*Y + Q**

We then see if the square of the first complex number ( $X + Yi$ ) is greater than a large constant, and our number of iterations is higher than the number of colors we can use. If either of those isn't true, the iteration stops, and the pixel is painted. In this fashion, all the pixels are colored, and you have your fractal! by tinkering with the numbers, you can create some beautiful results.

## Thinking about tools

The importance of getting the mathematics to show a fractal image is important, but equally important is making it easy to use. All of us, at one time or another have been frustrated with a poor set of controls, whether it's a game, or drafting software. You need to think hard about which controls you wish to include in your Interface.

Here are some things to get you going:

- Where on the screen will the main event -the fractal- be?
- What controls do you need? (numeric input)
- What controls would make it fun? (color changing function)
- Can you label everything so that anyone can use your interface, even in your absence?
- Can you include the mouse movement to help navigate the fractal?

These are just a few ideas. Some are harder to include than others. You need to decide which are the most important, or the ones you think make it the best program, and work on those. When you are done, everything should be a certain way for a certain reason, and it should be easy to use.

## Putting it all Together

Now that you've seen all the pieces necessary to make a fabulous fractal interface, you need to bring them all together. It is important to divide the work of the group up so that methods do not get confused, and so that things work together well even if they were not made by the same person. The finished product should look clean, easy to understand and use, and most of all, fun!

If you finish early, there are many things you can do to take this project further. You might want to make a website of favorite complex coordinates, an art gallery of numbers that can be plugged into the interface.