An Introduction to Computer Programming for Complete Beginners Using HTML, JavaScript, and C#

A Dissertation
Submitted to the Graduate School
in Partial Fulfillment of the Requirements
For the Degree
Doctor of Education
By Rembert N Parker

November 18, 2008

ABSTRACT OF THE DISSERTATION

An Introduction To Computer Programming

For Complete Beginners Using HTML, JavaScript, and C#

By Rembert N. Parker

Low student success rates in introductory computer programming classes result in low student retention rates in computer science programs. For some sections of the course a traditional approach began using C# in the .Net development environment immediately. An experimental course redesign for one section was prepared that began with a study of HTML and JavaScript and focused on having students build web pages for several weeks; after that the experimental course used C# and the .Net development environment, covering all the material that was covered in the traditional sections. Students were more successful in the experimental section, with a higher percentage of the students passing the course and a higher percentage of the students continuing on to take at least one additional computer science course.

Table of Contents

CHAPTER 1

Statement of the Problem

In a recent article, Bennedsen (2007) reports the results of a survey of Universities that reported their failure and withdrawal rates for Computer Science courses (I will refer to the sum of the failure and withdrawal rates as loss rates). While the survey indicated an average total loss rate of 33% for introductory courses, there were several responses that indicated more serious problems:

> One school reported an average failure rate, over a ten-year period, of 90%! Moreover, a university with 4000 students, where CS is the second largest major, reported a failure rate of 72%

The limitations of the survey were the limited number of participants, only 63 institutions; the authors noted that the low response rate may have been related to the reluctance of institutions to report embarrassingly high failure rates. Similar loss rates have been reported by numerous other sources.

When the author began teaching at a small liberal arts university in 2001 they were teaching both an introductory programming course using C++ and a web design course that taught some computer literacy and had students building simple web pages using HTML and JavaScript. The programming course was a required course for students interested in a major or minor in Computer Science, while the web design course was an elective course. While the students who

passed the programming course were clearly prepared to move on to the second computer course, about 40% of the students in the class either withdrew or failed the course. The web design course started with much simpler material, and did not go into as much rigor in programming assignments, but the author was somewhat surprised that the students in that course had less trouble than many of the students in the programming course had with some of the topics (in particular, writing functions). Several years later the department switched to using C# in the .Net framework for the programming course, but the department had difficulty in finding an acceptable textbook. Most of the C# textbooks that were available appeared to be C++ textbooks that had simply been converted to the new language with few changes in the opening chapters, and other textbooks were clearly written for students who already knew how to program in another computer language.

Mahmoud (2004) reported the results of a course redesign that used HTML, JavaScript, and then Java with some success. Java and C# are similar in many ways; each is a proprietary language (C# from Microsoft and Java from Sun Systems), each has basic syntax that is clearly derivative of C and C++, and each is an object-oriented language. The author's institution was already using Visual Basic .Net to teach its database course, so C# was a better fit for our students since the .Net architecture would be familiar when they got to Visual Basic .Net. That Fall the investigator taught a section of CPSC 1400 which began by teaching the students basic web page design using HTML, then moved on to interactive web pages using JavaScript, and then moved into C#. A different section started immediately with C#. While it clearly took longer to get to C# in the section of the course that had the additional material, since calculations, if statements and loops are similar in JavaScript and C#, the first class was able to move more rapidly through those topics and both sections of the course finished at approximately the same topic. The

experimental section had significantly lower withdrawal rates than the sections the investigator had taught before using the traditional approach.

Purpose of the Study

It is not the purpose of this study to compare the complexities of Java and C#, but simply to determine if a course redesign can improve student performance in a C# course as has been demonstrated to improve student performance in a Java course.

The previous study done by Mahmoud, et al, showed that student success rates could be improved by inserting a section of HTML and JavaScript in front of the introduction of Java as a programming language. The researcher will run a similar test using C# instead of Java to determine if the additional introductory material assists students in mastering a different object-oriented programming language; it is possible that the results that Mahmoud helped get students past some of the difficulties of dealing with Java. There are many problems in dealing with the complexity offered by Java when trying to teach the course to new programmers; as noted by Roberts (2004),

> It should be possible to convene a similar task force to define a simple, stable subset of Java and a set of supporting libraries that can meet the needs of our community…In the Fall of 2003, the ACM Education Board did indeed convene a task force along these lines with the following charter: "To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity."

Research Question

Does an approach of using HTML, JavaScript, and a visual approach to teaching an introduction to computer programming using C# .Net lead to higher rates of successful completion among students than teaching an introduction to computer programming using only C# .Net?

Significance of the Study

Computer Science programs are all facing the problem of declining numbers of students in upper division courses, and a large number of different approaches to teaching the introductory course are being tried. This study attempts to discover if one specific course redesign can lead to a higher level of student achievement in the introductory programming course. If it does, it could lead to an increase in the number of students that are eligible to take upper division courses making it at least possible to increase enrollments in those courses and an eventual increase in the number of students who graduate with majors or minors in Computer Science.

CHAPTER 2

REVIEW OF LITERATURE

The Computing Curricula 2001 project (CC2001) was a joint undertaking of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM) that sought to replace the earlier 1991 report by updating the material for the changes that had taken place in Computer Science. The primary change was the rapid growth of object-oriented programming languages (OOP). Universities dealt with the introduction of OOP in numerous ways, none of which had yet found total acceptance. CC2001 dealt with this controversy by proposing not a fixed curricula but rather a collection of knowledge units that could be introduced to students using one of six different strategies that introduced the knowledge units in different orders over several semesters:

[1] Imperative First. This was the traditional approach, and started out by teaching the syntax of a specific language. At the time of the report, the primary language used with this approach was C++.

[2] Objects First. This approach was favored by schools that had moved to a specific language that was object-oriented (primarily Java).

[3] Functional First. This approach was favored by schools that introduced programming using a function language such as LISP or Scheme.

[4] Breadth First. This approach begins by exposing students to the field of computer science rather than concentrating on programming.

[5] Algorithms First. This approach examines problems and how they can be solved by computers without actually programming the solutions; instead, pseudo-code that is never actually implemented is used.

[6] Hardware First. This approach starts by looking at the details of how computers work, and working up from logic gates to machine language code before moving to higher level languages.

The report went into considerable detail over the merits and drawbacks of introducing programming early in the sequence of computer courses; while noting that serious problems are created by early programming, it concluded that, "the programming-first model is likely to remain dominant for the foreseeable future."

The report summarized the challenge faced by Computer Science departments as follows:

> Because introductory programs differ so dramatically in their goals, structure, resources, and intended audience, we need a range of strategies that have been validated by practice. Moreover, we must encourage institutions and individual faculty members to

> continue experimentation in this area. Given a field that changes as rapidly as computer science, pedagogical innovation is necessary for continued success.

Rather than causing the field of Computer Science Education to come together in a wave of "New Computer Science" the way that schools embraced "New Math" in the 1960s, this report seems to have encouraged a wide variety of approaches to teaching Computer Science. An extensive survey of literature on the teaching of introductory programming by Pears (2007) found the many papers were motivated by the problems faculty members had with the programming language that was used.

> Dissatisfaction with the use of C++ or Java as the language in the first course has caused faculty to both look back at other languages used in the past and/or to consider adopting languages such as Scheme or Python.

Much of the turmoil followed the introduction of object-oriented languages. In spite of all the debate that continues on the topic of object-oriented programming, a study by Cooper (2003) found little difference:

> From the results of our research, we see that there is no significant difference in the overall achievements between the students who took the CS1 course with the traditional procedural approach and those who studied the Object Oriented paradigm.

Unfortunately, the result was a failure rate of over 50% on the exam that was used to measure student achievement; clearly switching to a different type of programming language is not enough to solve our loss problem.

Reges (2006) found problems with a course that was based on "objects first."

> We have gone back to procedural style programming. I was motivated to do this after attempting and failing to teach a broad range of introductory students at the University of Arizona. I found that my best students did just fine in the new approach, but the broad range of mid-level students struggled with the object concept.

Reges went on to design a course that used objects, but did not require students to define them, and found that his new approach was successful. The approach taken by the research in this paper is to introduce objects in HTML that are familiar to the students (action buttons, textboxes, radio buttons and check boxes) and have them write code that affects and is affected by these objects without requiring the students to write the code to create objects, simply to instantiate them.

A study by Wilson (2001) found that the factor with the highest correlation with student success in an introductory course was a student's comfort level (the second highest correlation was math background). Since students are likely to have more comfort with using a web browser to view web pages than any IDE, it seems reasonable to begin our course there. Gonzalez (2006) echoed the results that Wilson had, pointing out additional reasons to create a positive attitude amongst students:

> Lack of self-confidence and the competitive atmosphere fostered by the perceived notion of introductory courses as a way to "weed out" those not "fit" to be in science have been identified as a major contributing factor to the high drop-out rate in Science and Engineering courses, particularly among women and minorities.

An increasing number of courses that are based on the Breadth First model are using a new language called Alice. The purpose of the language is to meet the challenge of an objects first approach; as outlined in Cooper (2003),

> Our approach meets the challenge by:
>
> - Reducing the complexity of details that the novice programmer must overcome
> - Providing a design first approach to objects
> - Visualizing objects in a meaningful context
>
> Unfortunately, it does this by introducing students to an Integrated Development Environment and forcing them to learn to program immediately; it does, however, utilize a syntax that can be modified to visually "display Java-like punctuation to support a later transition to C++/Java syntax." I believe that we can accomplish all of these goals using HTML and JavaScript and web browsers that students are already familiar with.

Guzdial has written a series of papers related to the use of media computation to engage and motivate non-majors; since most students in out introductory course have not yet declared a major, it is likely that his findings apply to many of our majors as well. Guzdial (2001) was involved in the creation of a computer science course at the Georgia Institute of Technology ("Georgia Tech"). Using a language that accepted (but did not require) HTML commands, students created web pages that could include various types of media (pictures, sound, links, etc.)

> While there is a lot of speculation about the benefits of multimedia exploration, research on learning and technology suggest that the *creation* of media by students has an even greater benefit for students.

Later the course was redesigned to include Python, and Guzdial (2003) reported success in turning a course that was hated by many students into one that met with better success (only 2 students withdrawing out of 120).

> Computer Science Departments are not currently successful at reaching a wide range of students who are taking introductory computer science. The evidence for this statement includes…failure rates sometimes as high as 30%... Studies suggest that computing courses are seen as overly-technical and avoiding relationships to real applications, and are frankly boring and lacking opportunity for creativity.

Part of his solution to these problems is having students create web pages that they design themselves.

There are also positive aspects to using web pages as a starting place for an introductory course, as pointed out by Phillips (2003):

> The internet algorithmics framework makes a more convincing and exciting theme from the student perspective. And it has the distinct advantage of providing visual products right away, since experience tells us that students want to accomplish something right away, or they lose interest. That something must be something they can proudly *show* to others, such as through a web browser.

Kurkovsky (2007) reported on the results of a course named “Introduction to Internet Programming and Applications” that was “partly influenced by the work of Forte and Guzdial.” After an introduction to the Internet and the Web, the course teaches programming by first having students author web pages using HTML and then concludes with the use of JavaScript to introduce the core elements of programming. The course specifically avoids math-related examples in an effort to keep all the students engaged and motivated.

Another school that extended Guzdial’s work was the University of Illinois at Chicago. Sloan and Troy (2008) noted failure rates as high as 50% in introductory courses and retention rates from freshman to sophomore year of 40-60% and determined that part of the problem was the mixture of students with some computer programming background with a larger number of students who had little or no background. The latter group (which made up about 80% of the class) would struggle to keep up, and many would leave in frustration before the course was finished. Their solution was to create a CS 0.5 course. Students with a programming background could test out of the course, while other students would take a course that was designed to introduce students to programming using an adapted form of Guzdial’s course at Georgia Tech. The course from 2002 to 2004 used HTML and JavaScript, and in 2005 a switch was made to Python. In the final year using HTML and JavaScript the success rate had increased to over 93% and the retention rate to a second year was about 44% (success rates began decreasing when the switch was made to Python, but the retention rate increased). While this approach clearly has merit, it requires an extra semester for students, and was not an option at the investigator’s university; instead, a scaled-down introduction to HTML and JavaScript during the first few weeks of the course will be followed by the use of C# in the .Net environment.

David P. Ausubel (1963) pioneered the concept of advance organizers as a cognitive strategy to be used in improving the learning process.

> The short-term interference of similar elements, so crucial in rote forgetting, becomes relatively insignificant when meaningful materials are anchored to established subsuming concepts and progressively interact with them to the point of obliterative subsumption…
>
> The advantage of deliberately constructing a special organizer for each new unit of material is that only in this way can the learner enjoy the advantages of a subsume which both (a) gives him a general overview of the more detailed material in *advance* of his actual confrontation with it, and (b) also provides organizing elements that are inclusive of and take into account most relevantly and efficiently both the particular content contained in this material and relevant concepts in cognitive structure. It thereby makes use of established knowledge to increase the familiarity and learnability of new material.

Bergin reported on the results of studies done by John Carroll at MIT in the 1980s that lend support to the use of a constructivist approach to teaching computer science. Two approaches to technical teaching were used. The Objectivist approach had a logically partitioned sequence of material, while the minimalist approach gave the students tools and a series of tasks to complete without detailed instructions for completion (Bergin noted that the minimalist approach is "closely related to Constructivist educational theory"). The Objectivist theory stressed "the structure of the information itself," while the minimalist approach depended on

"active learners integrating new skills into what they already know." The experiments produced evidence that the Objectivist approach failed while the minimalist instruction succeeded.

Ben-Ari (1998) concluded that computer science education had not made as much use of constructivism as science and mathematics education, but felt that it was important.

> If the student does not bring a preconceived model to class, then we must ensure that a viable hierarchy of models is constructed and refined as learning progresses.

Students in an introductory computer programming course who have no previous programming experience to draw on are probably not familiar with the material that will be covered in even a general way. This problem has been pointed out by DuHadway (2002):

> CS1 tends to be demanding due to the amount of material covered and the workload. If students have not had previous programming experience, they are faced with the challenge of learning programming concepts, C++, and a development environment simultaneously. Having to learn in three different domains simultaneously often leads to cognitive overload. When the quantity of information is too large, students may be unable to synthesize all the information. They are then left with gaps in their understanding that make it more difficult to succeed as new material is presented.

What students *are* familiar with is the use of a web browser to view information (if nothing else, all of our students have used the school's email system). We can use this in combination with the work the students do building web pages to introduce advance organizers into the curriculum of the course and thereby hope to improve their retention and learning. We can help build scaffolds for the students by using simpler HTML and JavaScript concepts before moving to the harsher domain of a strongly-typed language such as C#.

The use of web page design to introduce programming is not a new idea.

Dale (2006) reported the results of a survey on which topics teachers thought were most difficult for students in CS1. The number one stumbling block identified was an inability to solve problems. While we cannot change the backgrounds of the students, many of whom have a lot of difficulty with any word problems, we can design assignments early on that give students an opportunity to develop some problem-solving skills. Instead of handing out complete specifications and showing students exactly how a program should look, we can give them descriptions of the problem the program is to solve and force them to go through the steps of defining the solution. The next biggest hurdle was the use of parameters in functions, followed by arrays and looping. Special attention can be given to the introduction of these features, using visual cues in HTML/JavaScript to ease understanding of them before covering some of these topics in more detail in C#.

According to Beaubouef (2005), a study by McConnell (2002) showed that textbooks are a contributing factor to the high attrition rates:

> Older textbooks (on procedural programming languages) averaged under 600 pages. For example, the average FORTRAN textbook was 379 pages. More modern textbooks are considerably larger, with the average Java text weighing in at a whopping 866 pages.
>
> Perhaps more alarmingly newer textbooks contain less coverage (on average) on simple data types, arithmetic expressions, relational and logic expressions, repetition statements, subprograms, and arrays.

Current textbooks seem to aim at also being complete reference books, leading to a large amount of material that students will never use; several books that exceed 1000 pages have shown up in the past few years. There is no textbook or combination of textbooks at present that would use the proposed approach to teach students, so an online textbook and a series of handouts were created to use for the course.

The following chart shows how topics in HTML and JavaScript will be used as advance organizers for the C# material:

HTML/JavaScript	C# .Net
Edit, View, Re-edit	Edit, Compile, Execute, Repeat
Container Tags for text markup	Block structures for programming
Tag Properties (better name for attributes)	Object Properties
String constants in quotations	String constants in quotations
Names for input boxes	Variable names
Input boxes, radio buttons, check boxes, action buttons	Objects
Assignment statements to change colors and pictures	Assignment statements
Dot Operator	Dot Operator
onClick events	Event procedures

Concatenation	String operations
Writing functions to improve readability and editing of code	Writing functions to create reusable code and modular programs
Passing values to functions to change colors	Passing values to functions to change results
Changing textbox values	Assignment statements/variables

Chapter Three

Methods

Subjects

The subjects of this study were students at a private university who were enrolled in an introductory programming course. Most of these students will be entering freshmen, but some upper class students may enroll in the course as well. Students will be randomly assigned to course sections that are one of two groups:

1. Students taking the course using the traditional approach of starting with C#.
2. Students taking the course using an approach that utilizes HTML and JavaScript before moving on to C#.

Test Instrument

Since the research is intended to measure student success as a reduction in withdrawals and failures, the only test instrument will be a notation of whether the student passed or did not pass the course. Withdrawals and failures will be grouped together since Federal guidelines for full-time students (a minimum of 12 semester hours completed each semester) cause some students who would otherwise withdraw from a course to simply stop attending class and accept a failing grade. Previous studies by Sloan (2008), Herrmann (2004), and Nelson (1990) have defined success in a course as a grade of A, B, or C and failure as a grade of D or F or withdrawal from the course before it was completed and a grade was assigned. Success will therefore be measured by counting the students who achieve a grade of A, B, or C while failure

in the course will be measured by counting the students who achieve a grade of D or F or who withdraw from the course without completing it.

Consideration of Other Factors

Students enrolled in the course were primarily entering freshmen, and the only information usually available to them when they register for a course was the time and days the class meets, reducing the chance that students pre-selected either the class section using the experimental approach or the class section using the traditional approach. Multiple teachers were involved in teaching the courses, but they worked together when the C# programming exams were prepared, using similar and/or identical questions on the exams to insure that the sections were covering and testing the same material. Any teacher that uses the experimental approach will have also taught using the traditional approach, and including this information in the study will help determine whether the teacher is having an impact on the results.

Since it is not possible to test identical groups, it may be that other factors enter into the results of the study. In addition to looking at student success based on the approach used to teach the material (the dependent variable), where available the following additional factors (independent variables) will be examined in determining student success:

- Gender
- Class standing – freshman versus upperclassman
- Experience in a programming class versus no experience
- SAT math and verbal scores
- High School class percentile rank: 100 * (Class size – rank) / class size
- High School class size

- Declared computer science major or minor
- High School GPA

Since the dependent variable is dichotomous (students either succeed or fail) and a number of independent variables exist that are of various types of data, a Binomial Logistic Regression can be used to identify the amount of variance that result from the various independent variables, leading to either acceptance or rejection of the thesis that the experimental approach improves student performance.

- Gender
- Class standing – freshman versus upperclassman
- Experience in a programming class versus no experience
- SAT math scores
- High School class percentile rank: 100 * (Class size – rank) / class size
- High School class size
- Declared computer science major or minor
- High School GPA

Regression tests will be run to determine if the experimental approach impacts success rates in the course.

Teaching Procedures

Both of the teachers who teach CPSC 1400 will be involved in the test, with one section taught using the traditional method while the other section is taught using an introduction to HTML before moving into C# programming. An online textbook will be used for the first few weeks in the latter section. This portion of the course will attempt to scaffold programming by focusing on the following:

- Easing into the edit/compile/test cycle by starting with an edit/view cycle.
- Introducing properties through the use of tag attributes.
- Introducing event programming using buttons and onClick events.

An Introduction to Computer Programming for Complete Beginners Using HTML, JavaScript, and C#

2007

BY REMBERT N PARKER

Introduction

You can't learn to ride a bicycle by watching somebody else do it. Oh sure, you can see how to hold the handlebars, where you feet go to pedal, and how to use the brakes, but until you actually get on a bicycle and try to balance yourself you have no idea what it really means to ride.

Computer Programming is the same way.

You can read a book, or view web pages, or watch somebody else write code, but you won't really learn what it is to program until you take a problem, devise a solution, code the program, and test and debug it (and then listen to a user complain about how it works and make changes). This textbook has short descriptions and examples of programming techniques, followed closely by some exercises. The exercises are almost more important than the descriptions and sample code, because only by doing the exercises will you actually learn how to program.

After a brief history lesson, we'll look at how to create simple web pages using Hyper Text Markup Language (HTML). I have a 1600 page book in my office that covers a lot of HTML, but it doesn't even come close to covering everything – it is not my intention here to teach you all the details of HTML, but rather to introduce you to a small subset that lets us create web pages.

The next section looks at JavaScript, a scripting language that lets you add program code to a web page so that it becomes interactive. A computer language can calculate anything that can be calculated by any computer or computer language as long as you have three things:

- A way to calculate values using simple arithmetic operations
- A way to decide which of two calculations to do based on a calculation
- A way to repeat one or more calculations

JavaScript has all three of these capabilities as well as a way to write functions which simplify our code. We will look at how to do some of these things in JavaScript before realizing that there must be a better way.

That better way is C#. This is a computer language that runs under an Integrated Development Environment called .Net that makes it easier to create interactive programs. Most books for C# are written for people who already know how to program in at least one other language, which puts most people at a distinct disadvantage; we will assume you know how to do simple things on a computer (edit files, copy files, move files, etc.) but that you are new to programming. Since this is new to you, if you feel lost, be sure to ask for help – you probably won't be the only one who feels that way.

This textbook is being provided to you for free as an online textbook (in part to save paper, and in part to allow us to update as changes become necessary), but it *is* copyrighted and you may *not* give copies to others, not even for free. You are being given the right to copy the pages onto a CD or DVD or flash drive so you can view them on other computers and put them on your own computer, but you are not allowed to copy them onto any media you do not own or to allow anybody else to copy them or display them on the web (or anywhere else). Since part of the reason for providing the textbook online is to save paper, we ask that you not print out a copy, even for your own use – since you can view the pages at any time, you shouldn't need a hard copy. That said, let's get to work!

An Introduction to Computing

1001010010101001010101010111110101101000101011010100010010010111010100010101 1

That string of 1s and 0s is exactly how something is stored in a computer! Whether the 1s and 0s are stored using lights, or capacitors, or vacuum tubes, or transistors, all a computer ever knows is whether a piece of data is a 0 or a 1. Each of these pieces of data, a 1 or a 0, is referred to as a **bit**, and by themselves individual bits are pretty much meaningless. Looking at the string of bits above, we can't tell if it's one or more numbers, a series of letters or words, a piece of music, a portion of a picture, the code for a program, or just random garbage that's been typed in by a monkey that was trying to type on a keyboard. The bits only change from being data to being information when we organize them in a specific structure.

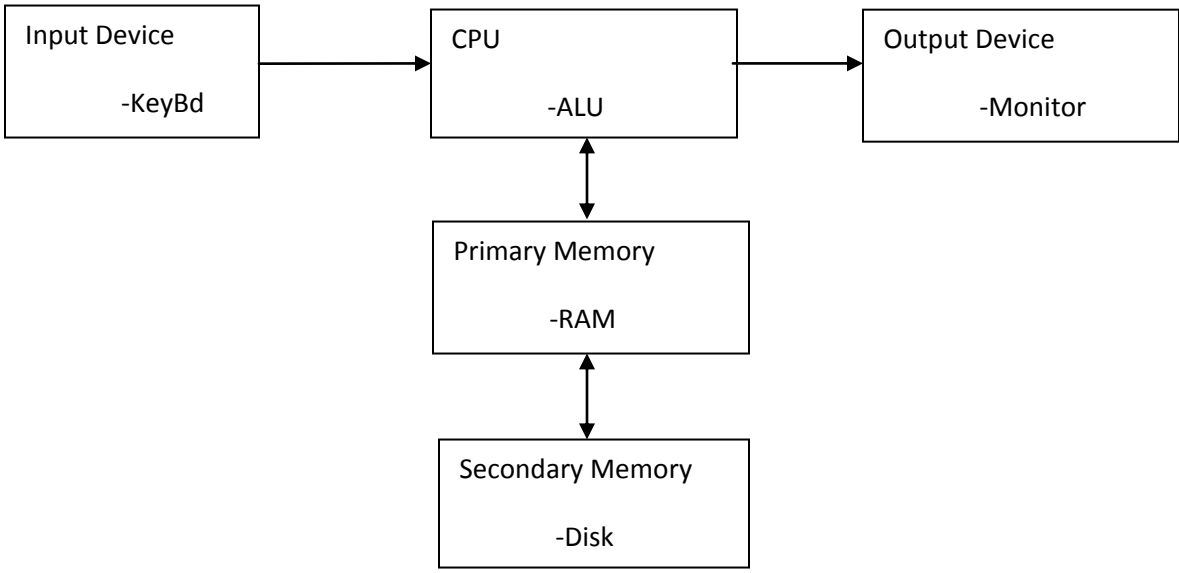
The next step up from a bit is a **byte**, a collection of eight consecutive bits. By using binary arithmetic a byte can store a number from 0 to 255. Alternately, by using an agreed-upon code a byte can store one of 256 different characters - enough to represent all the upper and lower case letters and numbers and special characters on the typical keyboard with plenty of codes left over. While early computers had hardware that could handle one byte at a time, later computers could handle two or four or more; this led to the concept of a **word**, which was a collection of bytes. While bits and bytes are the same from computer to computer, words are

not. IBM mainframes used an eight bit code called EBCDIC, while pretty much every other early computer used a seven-bit ASCII code to represent characters (the 7 bit code was later extended to eight bits). The EBCDIC and ASCII codes were not compatible at all. The computer industry currently has rallied around an extension of the ASCII codes called Unicode that defines 16 and 32 bit codes as well. These codes and what they represent are not that important to us since we will usually be looking at text, not internal computer codes.

Computer Hardware

The computer itself is referred to as hardware, a name that indicates you can physically touch the components: the monitor, the keyboard, a mouse, a printer, speakers, etc. The part of the computer that does all the work is the **Central Processing Unit** (CPU). The CPU contains a Control Unit that keeps track of what's going in the computer, an Arithmetic and Logic Unit (ALU) that does all the calculations, and special pieces of memory called registers that can hold data that the ALU can act upon. Changing values in the registers is the fastest thing a computer can do, but the registers themselves are very, very expensive, so the number of registers is usually very limited. Registers are usually only a few bytes long (most of them are exactly one word in length). Since registers cannot hold much data, computers also contain Random Access Memory (RAM). This memory is where both programs and data are stored while a program is running; the program instructions and data in RAM is copied into the CPU's registers and operated on there. RAM usually is erased when you turn off the power, so we need another way to store data that we don't want to lose, and this is done using disks, tapes, zip drives, CDs, and DVDs. The data on these **secondary storage devices** has to be transferred into RAM before

it can be used, so data may move many times during the course of a program. This arrangement of a computer is referred to as the **von Neumann architecture**. Because a program in a von Neumann architecture computer is nothing more than specialized data that is loaded into RAM, a general-purpose computer can be built without limiting the kinds of programs that it will be used to run.



When you turn on a computer, it loads up an **operating system**. This special software is responsible for the interface with the user (who is jokingly referred to as liveware). The operating system allows the user to select programs to run and enter data to the programs while they run. Early operating systems were simply scrolling lines of text, a far cry from the windowing interfaces we are used to now. You expect to run a program by clicking on an icon; not very long ago you had to memorize a large number of cryptic commands and type them onto a command line to get the operating system to load a program and run it for you. If you want to travel back in time, click on the Start icon, then click on Run, and enter cmd and press return – you will see a DOS box that looks a lot like a computer interface from the 1970s. Not pretty!

Computer Networks, the Internet, and the World Wide Web

In the 1950s and 1960s a number of companies produced mainframe computers for businesses that had the money to buy them. These machines cost up to millions of dollars, took up a lot of space, required special cooling (lots of air conditioning), and were typically less powerful than the computer you're reading this web page on. By the mid-1970s the minicomputer had shown up. These computers were smaller, less expensive ($50,000 and up), and less powerful, but were sufficient to support scientific programming and systems that did not need to deal with as much data as the mainframes. By the late 1970s microcomputers finally arrived. Initially they

were kits that had to be assembled and soldered together (much like a ham radio), but soon complete systems were available from Radio Shack, Apple, and finally IBM. These computers cost about $5,000, but at first they only had monochrome monitors, and diskette and cassette tape drives for input and output (hard drives came in the early 1980s). Printers were very expensive at that point, so small businesses that were using microcomputers wanted to find a way to share printers. In addition, it was necessary for several people to access the same data simultaneously on different computers, so it was imperative to find a way to share data that was on one computer with several other computers. Out of these needs we got the first local area networks that most people were aware of – small clusters of computers in a business or home that were hooked up to share printer, hard drive, and modems.

Computer networking was not a new idea in the 1980s; the US Government had established a network called ARPANET that successfully connected computers at four sites in 1969. Each of the four sites had a computer that was connected to the other three sites, and users who were connected to any of these mainframes could communicate with users at any of the other locations using live chats. In the early 1970s this network was used to transmit the first emails, and later it was used to transfer files as well. By the mid-1970s this network had escaped from government control and the term Internet was applied to it. We can't take a field trip to visit the Internet, because it isn't a single site; instead, it is a collection of computers that are connected to each other through a jungle of hardware that can switch packets of information between any two users. Using the Internet was a lot like being stuck back in the DOS box – users had to enter cryptic commands for even the simplest tasks, and use of the Internet was mostly for academics. In the late 1980s Internet Service Providers (ISPs) began to appear. These companies made it possible for anybody to get to the Internet by using a dial-up modem

and connecting to their ISP over a phone line. CompuServe was one of the early successful ISPs, but its interface was similar to the DOS model. America Online had a graphical interface, and was aimed at novice users. These and other companies made it possible for a much larger number of people to use email and join newsgroups, and they also made it possible to transact business over the Internet.

Wide-spread usage of the Internet had to wait a few more years. Apple computers had a feature called HyperCard which allowed a user who was reading text to click on certain phrases and open another card that contained more information. Tim Berners-Lee was a British scientist who took the idea of hypertext and designed a browser that would read files over the Internet…the program that was produced in 1991 was named WorldWideWeb, and it opened the Internet to browsing by anybody. Within a very short time other browsers had appeared. A web browser is simply a program that reads a file and displays the file's information for the user; its strength is that the file can be written using any computer and then be read on any other computer as long as the files follows the rules for creating a web page. We'll see some of these rules and create web pages in the very next chapter!

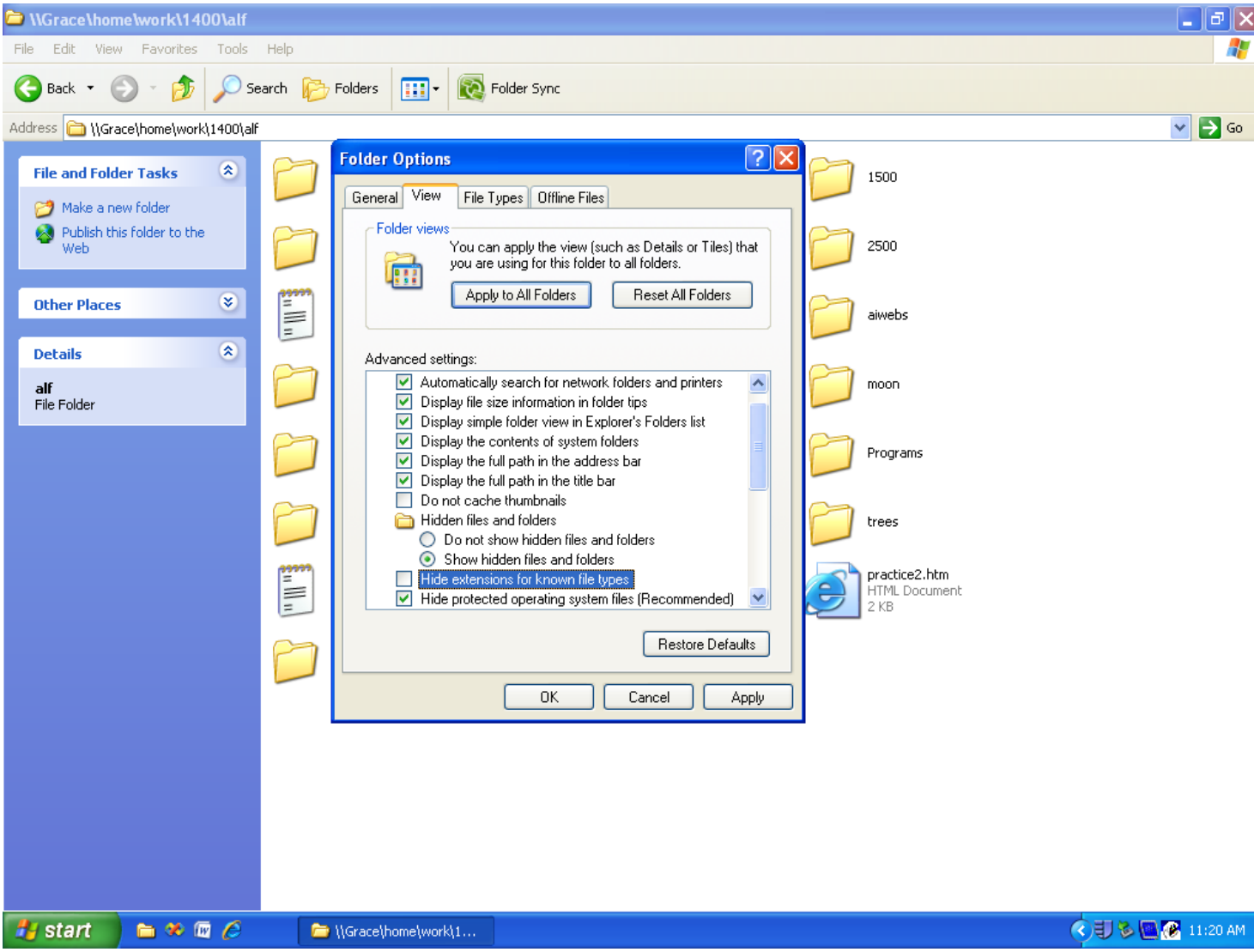
Chapter Two – Your First Web Page and a Few Tags

Early microcomputers didn't have hard drives for storage; my first microcomputer (a TRS-80 model 1) had only a tape recorder to use for reading and writing files. After I saved up for a few more months I was able to buy an expansion interface that let me add a 5 inch floppy diskette drive, and after even more savings I got a true luxury – a second diskette drive. A few years later I got my first IBM computer, and it came complete with two built-in diskette drives. So that you could tell the operating system which drive to read from, it was standard to name these drives a: and b:. A year or two later IBM announced the PC-AT, a computer that came with a built-in hard drive. Since almost all computers up until that time potentially had two diskette drives, c: was used to identify the hard drive. When local area networks were used to connect multiple computers to a central microcomputer, additional drive letters were used to identify the central computer's drives, printers, and/or modems. Even though very few computers are built that include diskette drives any more, we continue to identify the primary hard drive as drive c: and any additional devices using higher letters (it's more common now to see DVD and CD players in those slots). The network administrator can set up your login to the network to designate a specific drive (such as H: or I:) to map directly to a folder that is your home folder on a network drive so that you can get to it easily – if you click on Start and then My Computer you will see all the drives you have access to.

Navigate to your home area on the network by clicking on the appropriate drive, and then right-click on the window that opens for that drive and create a folder named Webs by selecting New

and then Folder. Double-click on that folder and you're in the area where we plan to save all our web works.

When the early IBM microcomputers were shipped, file names had a rigid format: each name could be only one to eight letters long, optionally followed by a period and a one to three letter **extension** (folder names could be up to eight letters long and could not have a period or an extension). The three letter extension was supposed to identify what kind of file it was – text files were .txt, executable files were .exe or .com, and so on. To insure that their programs were reading the correct type of file, software manufacturers began using specific extensions on the files their programs created, resulting in extensions such as .WP for Word Perfect files, .WKS for Lotus 1-2-3 files, and so on. The list grew rapidly, and in some cases more than one company was using the same extensions for completely different kinds of files. Current versions of Windows allow names longer than eight characters, and extensions longer than three characters, and each computer has a file where the operating system keeps track of what program the user wants to open for files with specific extensions. With the intention of making things easier for users, Microsoft added a feature that hides the extensions of files when you list them. While this may simplify things for some users, it will complicate your life a lot if you change file extensions of a file, which is something we are getting ready to do, so before you do anything else you need to check the folder that you have your files in to make sure that this option is turned off (be sure to check your home computer if you are working on programs there as well!) If you are looking at a folder, at the top there is a menu with lots of options – we want to click on Tools, then Folder Options, then View. This will bring up a long list of options, most of which you will want to leave alone. About half-way down the list is Hide Extensions for Known File Types – make sure that this option is NOT checked, and then click Okay.

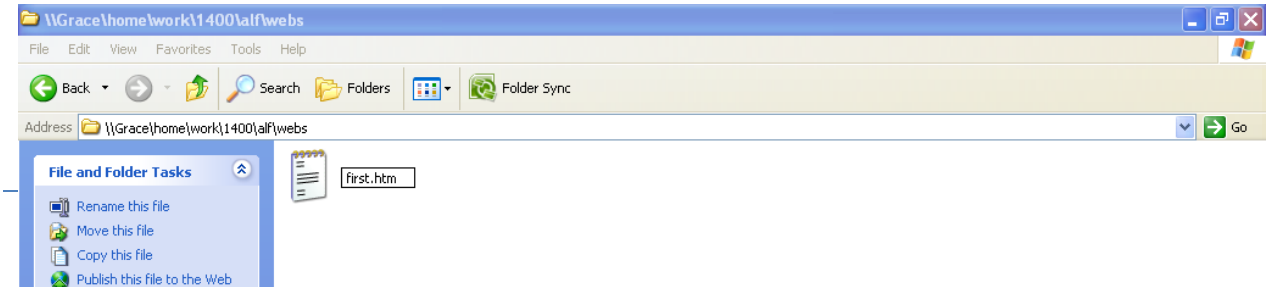

\\Grace\home\work\1400\alf
File Edit View Favorites Tools Help
Back Search Folders Folder Sync
Address \\Grace\home\work\1400\alf
Go
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Other Places
Details
alf
File Folder
Folder Options
General View File Types Offline Files
Folder views
You can apply the view (such as Details or Tiles) that you are using for this folder to all folders.
Apply to All Folders
Reset All Folders
Advanced settings:
Automatically search for network folders and printers
Display file size information in folder tips
Display simple folder view in Explorer's Folders list
Display the contents of system folders
Display the full path in the address bar
Display the full path in the title bar
Do not cache thumbnails
Hidden files and folders
Do not show hidden files and folders
Show hidden files and folders
Hide extensions for known file types
Hide protected operating system files (Recommended)
Restore Defaults
OK
Cancel
Apply
1500
2500
aiwebs
moon
Programs
trees
practice2.htm
HTML Document
2 KB
start
\\Grace\home\work\1...
11:20 AM

The easiest way to create a text file is to simply right-click on the folder and select New, and then select New Text Document. A file will show up, and we want to change the name of that file to First.txt. Once you've made this change, double-click on the file and Notepad should open up so that you can edit the file. Type in a few sentences, and then save the file. Examine the folder to make sure that the extension shows up; if it doesn't, go back and check your folder options.

Originally UNIX computers saved web page files with a **.html extension** (which stands for **Hypertext Markup Language**), but since early IBM-compatible microcomputers could only handle three letter extensions this was shortened to **.htm** on DOS and early Windows computers. While current Windows computers can handle longer extensions, old habits die hard, so most of us continue to use .htm extensions. Now for the trick – change the name of your file to First.htm. When you do, the operating system will jump in to make sure you know what you're doing by popping up a box that asks you a question:

> If you change a file extension, the file may become unusable.
>
> Are you sure you want to change it?

Since you're sure you want to change it, simply press yes, and two things should happen – the file extension will change, and if you're viewing the folder using icons you'll see a new icon for the file. Double-click on the file again, and this time instead of opening the file with Notepad, the operating system will open the file using your web browser – you've just authored your first web page!



Changing a Web Page

Normal web pages are nothing more than text files that have been read by a **web browser**, a special program that reads the text file and then displays it on your monitor inside a window. You can resize the window, and usually the web browser will automatically word-wrap the page so that you can still see the page (or, if the window is too small to see pictures or special text, the web page will allow you to use scroll bars to move up and down or side to side to see things). You can view the text of any page on the web by using the web browser's menu (if the menu is not visible, right-click on the upper section of the web page and click on Menu Bar to bring it back). From the menu, select View and then Source and Notepad will open up with the source file for the web page. Do this to the first.htm file, and you should see the source file you created. Add another sentence, and then save the file without closing Notepad. If you look at the bottom of your screen you should see boxes for each of the applications you have open – you can switch back and forth between your Notepad program and the web browser by clicking on these boxes. Alternately, you can minimize and restore the various windows by using the icons in the upper right-hand corner of each window.

To see your changes, click on the green icon at the top of the web browser to refresh the screen (or you can pretend you're a dinosaur programmer and simply hit the F5 key; we'll see that key be useful through the entire course).

Once that works, go back to the open copy of Notepad – do NOT click View-Source again! If you do View-Source a second time you will open a second copy of Notepad, and it may be easy for you to get confused on which copy of Notepad has the most recent copy of your file…or worse still, you may end up saving an early copy of your file over the top of your final file and lose all your changes. Go to the last sentence you entered and press return a few times to start up a new paragraph and type in another sentence or two; save the file and then refresh the web browser.

Uh-oh. The web browser only has one paragraph instead of two! Go back and add five or six spaces between the first and second word in the first sentence, save the file, and refresh the web browser; nothing changed, all the extra spaces have disappeared!

This demonstrates the way that a web browser wipes out **white space**, characters that would normally be used to skip spaces or lines in a display. Samples of white space are the space, a tab, or a line feed that results from hitting the enter key. This is done, in part, because when a web page is designed, we don't know if the user will be using a web browser on a 12 inch or 24 inch monitor or in a full-screen or partial screen window, so if the creator of the web page puts line feeds on a page they may show up at the wrong time (word wrap on a 12 inch monitor would clearly be different than word wrap on a 24 inch monitor). To go the next line we **mark up** the page by inserting special **tags**.

An **HTML** tag is a way to send a code to the web browser program to give it special instructions about how to display the web page. The simplest tag is one that causes a line break; all tags

consist of the opening symbol < followed by a single word that gives the command to the web browser and finally ending with the closing symbol >. The break command is therefore

Go back to your source file and add the
 command at the end of first paragraph. This command can go on the same line, on the next line by itself, or at the start of the second paragraph; it is up to you to select which way you want to use the break tag, but try to be consistent. Save the text file and refresh the web page and you should see two paragraphs. If you want more space between the two paragraphs, simply put two or more breaks in there (each break tells the browser to go to the next line before displaying more data).

A few rules about tags:

- You cannot put any spaces between the < symbol and the command word. If you do, the command will be ignored.
- You can put one or more spaces after the command word and the web browser will still follow the command.
- You can put one or more spaces or line feed before or after the tag so that it is easier to read.
- Upper and lower case letters are the same.
 and
 and
 and even
 will all do the same thing. It's a good idea to be consistent about how you capitalize commands; I tend to use all lower case letters, other programmers use all upper case letters or simply capitalize the first letter only.

Tags do NOT print – when the web page is displayed it will not show the tags, it will simply implement their commands.

Here's a second tag you can use to separate sections of a web page: <hr> This probably stands for horizontal rule, and creates a horizontal line on the web page. The line is put on a line by itself, so it automatically will separate two paragraphs.

Viewing Somebody Else's Web Page Source File

Now that you know how to view and change your source files for web pages, you may think about looking at or changing other people's web pages. Go to any web page on the web, and you can probably use the View-Source trick to see the source code for the web page. The first thing to notice is that when Notepad opens the file it changes the name to add [1] to the end of the file name – this alerts you that while you can change the text in your Notepad window, you will NOT change the original file that you viewed. This happens because you do not have rights to change the file, so Notepad was opened with a copy of the file. You can save this copy to a folder where you do have rights (making, for example, a copy of the file and saving it to your hard drive) but you cannot change somebody else's file.

Sometimes when you use the network you will open your own files using an alias address with a web browser but you will not have rights to save the file – be alert to the signal [1], and be ready to navigate to the folder that the file is in rather than getting to it through the network. This is not likely to happen on your own computer, only when you are in a computer lab.

Container Tags

Tags can also be used to affect how text looks. The most common changes we make to text are to make it bold, or underlined, or italics. Pick an important word in your file that might look better if it was bold and put the bold tag before that word: Save the file and refresh the web browser and you'll discover that *all* the words from there to the end of the document became bold. Not exactly what we had in mind, but how is the web browser to know when to stop making words bold? Web browsers could have been written to bold only one word after the bold tag, but then we would have to put a bold tag in for each word when we wanted to bold a phrase, and that would be a lot of extra work. Instead, we have to tell the web browser when to stop using bold type. Tags that have a beginning and an end are called **container tags**; they affect all the text that is between the opening tag and the closing tag. To close a container tag we put a second tag that is made up of the < symbol, a / symbol, the word that we used to start the opening tag, and a closing > symbol. To close the bold tag we would use . Thus, we might have a sentence like

You can use container tags to hightlight text

Which would show up as

You can use **container tags** to hightlight text

In addition to bold, we can create underline and italics text by using these three tags:

	bold	
<u>	<u>underline</u>	</u>
	italics	

So the sentence

> We can then make sure that the reader <u> pays attention </u>
>
> to parts of a sentence by using container tags .

Shows up as

We can then **make sure** that the reader pays attention to parts of a sentence by using *container tags* .

Try using all the tags in your first.htm web page, and then save it.

Chapter Three – Tags Have Properties

Web page files should have some housekeeping tags that help the web browser determine how to display a web page. These tags are all container tags, and some of them are contained inside other container tags. You can visualize container tags as various sized bowls and cups and that will make it easier to see how some containers fit inside other containers. Suppose we had a sentence that had both underlined and bold type:

> When creating a web page, one important thing to consider is **who is going to look at the web page and what *they* want to find out from the web page**.

For this example, we have a section of text that is bold and a section of text that is italics and bold, so the italics section is fully contained inside the bold section. Just as a cup cannot be half inside and half outside another cup, you should not overlap tags. If you wanted this passage:

aaaa **xxxxxx** **yyyyyy** zzzzzz bbbb

this set of tags would be unpredictable:

aaaa xxxxxx <u> yyyyyy zzzzzz </u> bbbb

It might work, and on other browsers or other web pages or other days it might not. If you really wanted to get that effect, you would have to have one underline passage that is inside the bold tag container and a second passage that is outside of it:

aaaa xxxxxx <u> yyyyyy </u> <u> zzzzzz </u> bbbb

Now none of the container tags "cross" each other, and we should get the effects we want. This isn't very important when we are simply doing bold and underlined, but it becomes very important with other container tags such as the housekeeping tags.

The first housekeeping tag is the **<html>** tag - this simply informs the web browser that the file it is opening is designed to be read as a Hypertext Markup Language file. If this tag is not there, the web browser has to assume the file is an html file and keep reading, looking for tags to tell it what to do with the text in the file. The html tag contains the set of instructions to the web browser.

The next two housekeeping tags are **<head>** and **<body>.** Since a head is usually on top of a body, it comes first inside the html container, and then the body tag follows. Inside the head tag we put things that describe the web page or how it should be accessed, while inside the body tag we put the actual data that will be displayed on the web page.

The last housekeeping tag we are going to use is the **<title>** tag. When you look at a web page, at the top of the window is a line that shows the name of the file that you are viewing (and, if you're using Internet Explorer, it then shows the words "Windows Internet Explorer"). Since things like chapter03.docx don't mean much to most people, we can use the title tag to show something other than the file name. Since the title tag describes the title of the web page rather than some of the data that is actually on the web page, it goes in the head section of the source file. Here's what a sample file would look like with all of these housekeeping tags:

```
<html>
  <head>
    <title>  The Title Goes Here  </title>
  </head>

  <body>
    The web page goes here

  </body>

</html>
```

Several things to note about the style I have used to create this page:

- I indented the opening and closing tags for the head and body
- I indented the tags and text that went inside the head and body containers

- I used blank lines to make the various containers stand out from each other

Remember that the web browser removes all of the white space when it reads a file, so all of the extra spaces I put in do not affect how the web page looks, they only affect how easy it is to read and modify the source code for the web page. While computers are expensive, programmers are much more expensive, so it is important to find ways to make programmers more efficient, and using spaces and blank lines to make code easier to read is a very simple way to help them.

Go ahead and create a file named **default.htm** that contains those lines (you can do this by creating a textfile named default.txt, cutting and pasting the text into the file, saving the file, and then changing the name to default.htm).

Every time we want to create a webpage file we would have to key in these housekeeping tags, but this is a lot of busywork. Programmers work hard to create **reusable code**, and this is a simple example of a place where we can write code and then copy it instead of writing and typing it over and over again. Every time we want to create a web page we can now simply copy the default.htm file and change its name, copying all of these housekeeping tags and then simply changing the title and replacing the text “The web page goes here” with the text we want to show on the web page.

Once you have the default.htm file built, copy it and rename the copy props.htm so you have a file to play in.

Formatting Tags

Some tags can be used to format the text that is displayed on the web page; they are all container tags. Here are some of the tags that you can use in the body of the web page:

<H1> <H2> <H3> <H4> <H5> <H6> These tags create headings that are big and bold or smaller and bold. These tags automatically put the text that is inside of them on a line by itself, and makes it bold. Tags H1 through H4 are generally larger than the regular text, and H5 and H6 are often smaller.

<pre> This tag creates preformatted text that shows up as Courier font. The courier font is a fixed font, which means that all characters take up the same amount of space. When the pre tag is used the text that is inside of it is reproduced without removing white space, so it can be used to create columns of numbers or names that will line up properly.

<blockquote> Indented text is usually used for quotes, poetry, etc. When you use blockquote the text that is inside the container is indented from both margins so that it will stand out.

Tag Properties

Another way to define how a web page should look is to use properties (most other sources use the word “attributes” instead of “properties”, but a property by any other name will still change

how the web page looks). Each property has a default value, and if we don't give them a value in a tag the default value is used. When you look at a web page, it normally shows up with black text on a white background, but sometimes we want to change the colors to make the page stand out. We can do this by changing the properties that control the background color and the text color, but only if we know the names of the properties and where to change them. Names of properties are like magic spells – there are special words, and only trained mages, er, programmers, know these names. The properties are usually changed inside the tag that they are changing. Properties that change an entire document are usually changed inside the body tag, while properties that change a specific section of the web page are usually changed inside the tag that contains that section of the web page.

To change the background color of the entire page, we can modify the body tag:

```
<body  bgColor="Tan" >
```

Go ahead and change the body tag in your props.htm file, then save the file and reload it in your web browser to make sure your change worked. This simple tag shows the way all property values are changed:

- Leave one or more spaces after the tag name.
- Type in the name of the property you are changing followed by an assignment operator (that's the programmers' secret name for something you've always called a equal sign).
- Type in the new value you want the property to have; if it is a number, you can just type in the number, but if it is text you should put the value inside quotes.
- Close the tag with the > symbol.

- Caution: no spaces around the = symbol. If you put a space before or after it the assignment won't work.

After you get this to work, let's try to change the text color as well. You can assign new values to more than one property by putting one or more spaces between the assignments:

```
<body bgColor="Tan"  fgColor="White">
```

Hmm….that didn't work, the text is still black. When you key in a program change you should always **test** the change to see if it works; if it doesn't work, the program contains something wrong, something that can be referred to as a **bug**…and if there is a bug in the program, it is time to **debug** the change so that the code will work properly. When looking for bugs, the first thing to check is your typing, since computers have no common sense and even one letter typed wrong will confuse them. In this case, what's wrong is that we used the wrong name for the property for the text color; while JavaScript uses the property fgColor for foreground color, HTML uses the property text in the body tag, so our code should read

```
<body bgColor="Tan"  text="White">
```

There are books and web pages full of the details of property names and valid values for them, and you should always check on a property name before you use it: it is easier to prevent an error than it is to correct it. If we had looked up the property name before we typed it in, our program would have worked on the first try. Other errors can be more subtle than using the wrong property name or mistyping something. There is a specific list of supported color names; if we use a color that isn't on the list we will get unpredictable results.

The horizontal rule tag has a number of properties we can change – width, align, size, and color. Here's an example of a horizontal rule tag with several properties:

Example: <hr width=75% align="left" color="#729985" >

The width indicates how far across the screen the line will go, with a default width of 100%. Since we don't know how wide the monitor or window is where the web browser will be running, we can use a percentage to define how far across the screen we want the line to go. Alternately, we can define the width using a number of pixels:

```
<hr  width=400>
```

If you look very, very closely at a monitor you can usually see the little dots on the screen that are different colors (even lines are actually collections of dots). Each of these dots is referred to as a **pixel** (a picture element), and if we use a number for a width we are telling the web browser how many pixels wide something is. You need to be very careful when defining sizes using pixel counts since not all monitors have the same number of pixels, and a line that looks okay on one monitor might be too long or too short on another monitor. When you use a percentage the web browser will make a line that is that percentage of the current size of the window that the web page is being displayed in. The number of pixels or the percentage does not have to be inside quote marks, but if you put them in quote marks your page will still work.

Alignment of the line can be left, right, or centered (the default is centered). If you designate an alignment of left, the line will start at the left margin and go across the page for the width you have defined. If you designate an alignment of right, the line will end on the far right-hand side of the web page, and it will start at a location that results in the width you defined.

Since there are only about 215 colors defined by name in web browser programs, and there are millions of other colors, we need a way to define all the other colors. You'll recall that one of the ways data can be organized is by bytes. The strange-looking color is a way of representing a

color using a certain amount of red light, green light, and blue light (RGB). Each of these colors can have an intensity from 0 to 255, the numbers that can be stored in one byte. Since it gets messy to show those numbers in base 10, we instead show them using base 16. The # symbol indicates that the number that follows is in base 16 instead of base 10, and we represent the three bytes for RGB colors by using pairs of base 16 numbers. Each of the six digits that follows the # symbol must be either a number from 0-9 or a capital letter from A to F. A number of 000000 would be completely black (no light of any colors) while FFFFFF would represent white (maximum light from each color). You can easily find charts on the web that show you what all the various colors are, or you can simply play with a web page and change the numbers to see what they look like as colors. Of course, you can also simply use some of the names that the web browser program recognize (you can easily find a list of those colors online).

Some other useful container tags and their properties include:

<p> paragraph tag: align can be left, right, center, or justify (justify is the way books are usually created – both the left-hand and right-hand margins are straight lines).

<font> font tag: size, color, and face can be set to vary the look of text within the container tag

Example: <font size=12 face="wingdings" color=tan> The size and face are the same as the codes you will find in word processors such as Word.

Exercises

[1] Copy a large chunk of text from another web page into props.htm (simply highlight the text with the mouse, click copy, and then click paste into the source file). Use H1, H2, pre, and block quote tags to organize the page, and then use the properties from the body, hr, p, and font tags to modify how the page looks.

Chapter Four – Putting Pictures on a Web Page

When the Internet first started, it was possible to copy files from one computer to another, but it was necessary to use one program to view text and a different program to view pictures – you could not easily see both at once. One of the strengths of the Web is the ease with which you can display pictures and text together with the same program (the web browser).

There are a number of different formats for picture files, each of which has a different extension. There are quite a few of them, but web browsers tend to support three important formats:

.bmp files – these files usually store the color for each pixel separately. Since there are a lot of pixels in most pictures, these files are very large, but they are also the most detailed. They take longer to download than the other formats, but look much better.

.jpg files – also known as jpeg files (pronounced "jay-peg"), these files contain a small amount of compression when pixels next to each other are the same or similar colors. These files can be smaller than .bmp files, and look almost as good.

.gif files – these files only support 256 colors, changing the colors of the picture to the color that is closest to it. This reduction in the number of colors makes pictures that are noticeably different when compared to other types of files, but they are also significantly smaller, making web pages that use them load much faster. If you are using a small picture which does not have much detail, .gif files are perfectly acceptable.

You can create your own pictures using a graphics editor and save them in one of these formats or draw a picture on paper and scan it into your computer. Of course, if you're as untalented as I am, drawing pictures isn't an option, and you are reduced to taking pictures with a camera and transferring those pictures to a computer, after which you can load the picture into a graphics editor and possibly change the format of the picture.

Another option is to use pictures you find on the web, but you need to be careful when doing this – many websites have copyrights on their pictures, and will not appreciate it if you simply copy their work without paying for the privilege. There are websites that license clip art for you to use, and there are many websites that make their images available for free as long as you credit them on your website or link back to their website. If you find a picture that you want to use on the web, you can save it to your computer by right-clicking on the picture and selecting Save Picture As – you can change the name of the picture when you save it, but do NOT change the extension or it will not display properly when you use it.

The tag to put an image on your website is the image tag, and it requires at least one property:

```
<img   src="picturename">
```

SRC is an abbreviation for source. The picture name should always be enclosed in double quote marks and must include the extension (some examples would be src="Roberts.gif",

src="Atlanta.bmp", and src="toons.jpg"). The picture should be in the same folder as the web page file; if the picture is in another folder, you must give directions to the web browser telling it where to look for the file. A name followed by a / symbol and a filename tells the web browser to go down to a folder and look for the picture there, and two periods followed by a / symbol tells the web browser to go up a level. Suppose we have a folder named webs than includes folders named pictures and pages, and that the pages folder contains our web page and a folder named cities. This leads to names like

src="cats.jpg" if the picture is in the same folder as the web page (the easiest setup).

src="cities/Atlanta.bmp" If the picture is in the sub-folder named cities.

src="../Roberts.gif" if the picture is in the webs folder.

src="../pictures/toons.jpg" if the picture is in the pictures folder.

When you put the image tag on your web page you should see the picture you saved, but you may see a small red x instead – this means that the web browser could not find your picture. The most common error when this occurs is that you typed "scr" instead of "src". If you got the spelling correct, then you should check to make sure the picture is in the folder you told the web browser to look in and that the extension of your picture file matches. Sometimes you should simply re-copy the picture.

Image Tag Properties

There are a number of other useful properties in the image tag.

Width

Width can be defined as either a number of pixels (width=500) or a percentage of the width of the web page (width=75%). The default is to use the number of pixels in the width of the actual picture (which you can see by looking at the properties of the picture file).

Height

Height can also be defined as either a number of pixels (height=500) or, much less frequently, a percentage of the width of the web page (height=75%). The default is to use the number of pixels in the height of the actual picture (which you can see by looking at the properties of the picture file).

If you supply either the width or the height for a picture (but not both), the web browser calculates the other item by using a ratio of the actual width and height from the picture file itself and then stretching or shrinking the picture to fit that calculated size. If you supply both values and they have the identical ratio as the actual values, the web browser will resize the picture and it will look fine, but if you supply both values and they are slightly out of sync your picture will look like it has been sent through a fun house mirror. To avoid problems you can supply only one of the values, but to speed up the loading of your web page you should resize your picture using a graphics editor so that the web browser doesn't have to do any work to modify the picture. If you use lots of pictures on a page you can supply the exact height and width values so the web browser can leave space for the pictures, load all the text immediately, and then go back and fill in the pictures.

Align

The align property has five possible values which can be used to tell the web browser how to flow text around a picture.

Three of these values line up one line of text with the picture, and then move under the picture for the rest of text; in each case text is put on the page until the image tag is reached, then the picture is put on the page, more text is put on the page to the end of the line, and then the rest of the text is put on the page under the picture. The line of text is lined up with the top, middle, or bottom of the picture:

align=”top”

align=”middle”

align=”bottom”

The other two values for the align property put the picture on the page and then scroll text around the picture. The picture is placed either at the left-hand side of the web page or the left-hand side of the web page:

align=”top”

align=”middle”

There is no align=”center” tag! You can use the center tag to center text, a picture, or pretty much anything that that is inside the center tag:

```
<center>
        You can put text here, or <img src=”toons.jpg”> or most anything
</center>
```

You can also use
 tags to separate the text from a picture. As is true quite often when you are working on a computer project, it may take a little trial and error to get the result you want. In a later chapter we will see another way to control the layout of text and pictures by using the table tag.

Border

You can add a black border around a picture by using a border tag. The width of the border is defined by the size of the border:

<img src="rr.jpg" border=10> creates a border that is 10 pixels wide

<img src="rr.jpg" border=30%> creates a border that has a border that is 30% of the size of the picture

Hspace, Vspace

In addition to adding a border, you can vary the amount of space between a picture and the text near it by adjusting the horizontal and vertical space on each side of the picture.

<img src="rr.jpg" Hspace=10> creates a white space border that is 10 pixels wide on either side of the picture but does not affect the vertical space.

<img src="rr.jpg" Hspace=10 Vspace> creates a white space border that is 10 pixels wide on all sides of the picture.

There are, of course, lots of other properties.

Exercises

[1] Create a web page named pictures.htm with at least five pictures and several sentences of text for each picture. Include the use of each of the five Align attribute choices at least once; be sure to include enough text to clearly show how each one works. You will likely need a paragraph or more for each picture to get the text to flow properly.

Chapter Five – Links, Lists, and Tables

One of the features of web pages that made them so easy to use was links, highlighted text that opened another web page when you clicked on the text. We use a link to attach to another web page by using the <a> tag. This container tag has three parts: the anchor tag, the text that will show up highlighted, and a closing tag:

```
<a  href="targetpage.htm">
    This text will be highlighted
</a>
```

You can put these three pieces together or on separate lines since the white space will be removed by the web browser.

The href property designates the web page to link to using either relative or absolute addresses – relative addresses are like those we used to find pictures in the same or nearby folders, while absolute addresses give the entire address of the web page starting with http:// If the web page we are linking to is in the same folder as the current web page, we can simply put the name of the file:

```
Click <a   href="next.htm">  here </a> to see the next page.
```

If we wanted to link to a page that is already on the web somewhere, we would use an absolute address:

```
Take a look at the </a   href="http://www.anderson.edu"> Anderson University </a>
page
```

Ordered and Unordered Lists

It is useful to be able to put a list of items on a web page using bullets, letters, or numbers. This can be done using the <ul> tag for an unordered list (bullets) or the <ol> tag for an ordered list (letters or numbers). These tags contain a series of list items, each of which will be marked and put on a separate line. We identify the list items by using <li> tags at the start of each item.

This is an example of a list of fruits using bullets:

Here are some popular fruits:

```
<ul>
        <li> bananas
        <li> mangos
        <li> limes
</ul>
```

This would show up as

Here are some popular fruits:

- bananas
- mangos
- limes

If you didn't like filled in bullets, you could use the type property to change the bullets:

```
<ul   type=”square”>
```

Other possible values for the type are disk and circle.

Ordered lists work exactly the same way, substituting ol for ul:

The top three vegetables I don’t like:

```
<ol>
  <li> okra
  <li> eggplant
  <li> mushrooms
</ol>
```

Which would show up as

The top three vegetables I don’t like:

1. okra
2. eggplant
3. mushrooms

There are a number of choices for type for ordered lists. Instead of numbers (a type of “1”), you can get upper case or lower case letters by using a type of “A” or ”a” or you can get Roman numerals by using a type of “I” or “I”.

If you remember doing outlines of papers back in grade school, we used a mixture of letters and roman numerals, indenting as we moved to more detail, and that’s exactly what we get if we

nest lists – the web browser will change the symbol used when it does a sublist. Let's build a list of pets by type of pet; the best way to do this is to first set up (and inspect) the list of types of pets:

```
Pets we have at home
<ul>
        <li> dogs
        <li> cats
        <li> birds
</ul>
```

We can save this list and view it to make sure it works the way we wanted, and then we can go back and add a list of pet names under each type of pet.

```
Pets we have at home
<ul>
        <li> dogs
          <ol>
                <li> Tinkie
                <li> Flash
          </ol>

        <li> cats
          <ol>
```

```
                <li> Hammer
                <li> Chewie
                <li> Pugs
            </ol>
        <li> birds
            <ol>
                <li> JB
                <li> Silent Bob
            </ol>
</ul>
```

If your list doesn't look right when you view it, the most likely reason is that you forgot to close a tag, so

when you use a container tag, it's often a good idea to key in the opening tag and the closing tag and then go back and type in the text that goes between them. This will help you avoid forgetting to close the tags.

The result of this long set of lists would be

Pets we have at home

- dogs
    1. Tinkie
    2. Flash

- cats
    1. Hammer
    2. Chewie
    3. Pugs
- birds
    1. JB
    2. Silent Bob

The web browser will automatically indent sublists for us, but the source code in the example above has indented text just to make it easier for us to read it. We can nest lists much deeper than two levels if we want to, and we can freely switch back and forth between ordered lists and unordered lists.

Tables

Tables can be used to control the layout of a page or to display data in the format similar to a spreadsheet. Tables are slightly more complex than lists since there can be multiple cells on each row of the table.

```
<table   border=3 >
  <tr>
        <td> Bob  </td>
        <td> Junior </td>
        <td> Math Major </td>
```

```
    </tr>

    <tr>
        <td> Sally </td>
        <td> Sophomore  </td>
        <td> Computer Science Major  </td>
    </tr>
</table>
```

Bob	Junior	Math Major
Sally	Sophomore	Computer Science Major

The table tag organizes each row as a series of cells going across the page – never forget that even though we type the cells going down the page, they display across the page until a new row tag appears. Each row tag <tr> contains a series of table data tags that define what goes inside each cell on that row.

The table tag has a large number of useful properties:

border=3	if you leave out the border tag, there is no border; this is done when you are using the table tag to layout a page, letting you put pictures and text where you want them
align=”center”	the align tag lets you align the table to the left-hand side of the page, the center, or the right-hand side of the page
bgColor=”lightblue”	Table text and background colors are not affected by the colors you set in the body tag, but fortunately the background color for all the cells in a table can be set in the table tag. You cannot set the text color for the table in the table tag; it is necessary to change them in each individual cell using <font> tags.

The table row tag is used primarily to group the cells for a row, and you will rarely need to use any of its properties.

The table data tag is a container tag that in many respects defines a mini-web page. Each cell can contain text, pictures, lists, even more tables…or combinations of any of these items. If you don’t set the width of the cells, each column is set to the size of the largest cell in that column. It is common to use bgColor, width, and height properties for the <td> tags, but many more properties are available.

```
<table>
  <tr>
```

```
        <td>    cell data goes here      </td>
        <td>    and another cell         </td>
        <td>  <ol>
                        <li> Ball State
                        <li> Indiana University
                        <li> <a href="http://www.anderson.edu"> Anderson University</a>
                  </ol>
         </td>
   </tr>

   <tr>
        <td>    row two cell data goes here      </td>
        <td>    nothing to see here  </td>
        <td width=30%>  <img src="valley.jpg"> <br> A picture of the campus  </td>

   </tr>
</table>
```

Exercises

[1] Create an ordered list of the top ten things about Anderson University in a file named Top10.htm. Feel free to include such important items as (ahem) adequate parking and the dreaded valley squirrels.

Your page should have the Anderson University logo centered at the top, a reasonable size heading, and an ordered list. One or more additional pictures would help as well. This will give you a chance to practice using list tags, image tags, header tags, etc.

[2] Create a nested list of the books of the bible. Obviously you'll divide them into New Testament and Old Testament, but you should have some subdivisions in those as well (we'll leave the divisions up to you). You should use lists that are contained inside other lists, not a series of lists (thus, there will be a list for books of the bible, inside of which there will be a list of old testament books and a list of new testament books – there will NOT be two lists).

Your page should include a changed title, a few appropriate pictures, and you should control the colors of the page as well as the fonts using the appropriate tags.

The assignment should be in your webs folder, and should be entitled bible_xxx.htm (where xxx is your initials).

[3]Create a webpage named States.htm with a table that contains at least four rows (one state per row) and various things about each state in the cells of that row:

Name of the state – use a heading tag for the name of the state; this name should link to a web page about the state

State Capital

Picture of the state flower

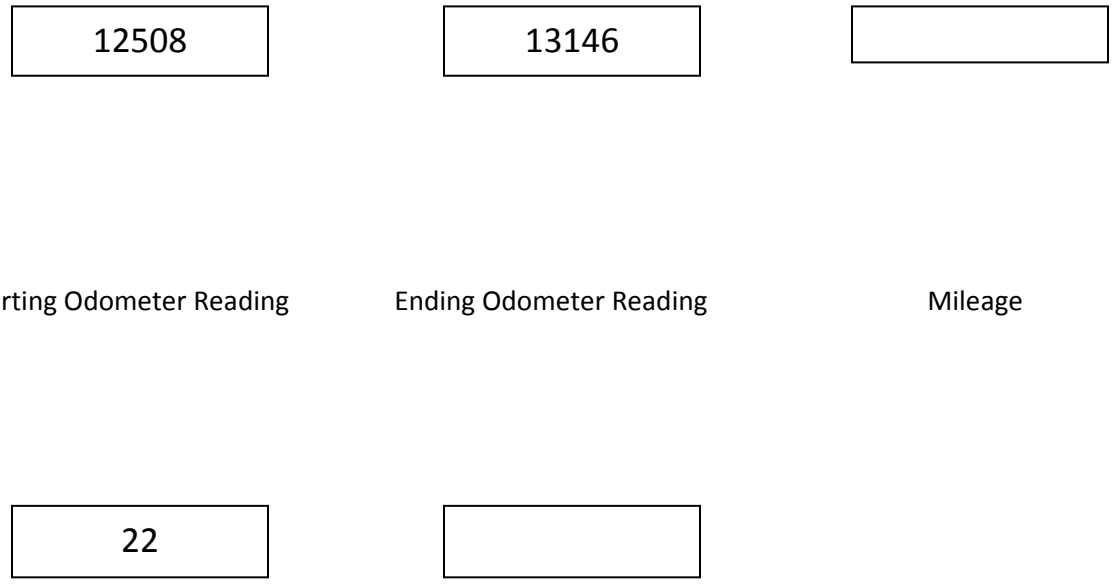
Picture and name of the state bird in the same cell

An ordered or unordered list of at least four major cities in the state

Above the table there should be a heading of some sort that explains what the table is.

Chapter Six – Naming Objects

All of the webpages we have looked at so far have only been static – other than linking to other webpages, the pages simply sat there without changing. One of the strengths of the Web has been the ability of the programmer to make webpages interactive. Let's look at the following simple example of an interactive webpage:



This webpage has several textboxes (solid rectangles that we can type into) and several labels (the text that isn't inside boxes). These textboxes and labels are generically referred to as objects – and other objects on a webpage might include check boxes and radio buttons and action buttons. We'll learn how to put all of these objects on a webpage in the next chapter.

This simple webpage is designed to calculate the miles per gallon for your car based on the starting and ending odometer readings and the number of gallons of gas that you used. As shown above, the user would input their information into three of the textboxes and then press another object (an action button) somewhere on the webpage to get the page to calculate the information needed for the other two textboxes and then fill them in with the correct amounts.

Before we start writing the code that would make this magic happen on the webpage, we should first design the **algorithm** that we want to use for the calculation. An algorithm is a step-by-step set of instructions that describe exactly how to do something; it is usually written in clear, concise English, not in some cryptic computer language - the purpose of the algorithm is to describe to a person how to do something. Using the algorithm, you should be able to accomplish the task that the program has in front of it without using a computer at all. Explaining the steps to the computer comes later.

An algorithm for this task might look like this:

> (1) Calculate the miles traveled by subtracting the number of miles in the Starting Odometer Reading textbox from the number of miles in the Ending Odometer Reading textbox, and display the result in the Mileage textbox.

(2) Calculate the miles per gallon by dividing the number of miles in the Mileage textbox by the number of gallons in the Gallons textbox, and display the result in the Miles Per Gallon textbox.

These instructions would be sufficient for most students to complete the task using a calculator, but they aren't what a computer needs to accomplish the task. While it is clear to us from a glance which textbox goes with which label, it isn't at all clear to a computer. We could rewrite the second instruction as

(2) Calculate the miles per gallon by dividing the number of miles in the textbox above the label that reads "Mileage" by the number of gallons in the textbox above the label that reads "Gallons", and display the result in the textbox above the label that reads "Miles Per Gallon".

It turns out that coding even that algorithm is very difficult since computers are not very good at understanding descriptions of locations and have no common sense – as an example, the computer might very well decide that since the Starting Odometer Reading textbox is above the label that reads "Gallons" it should use 12508 as the number of gallons.

To simplify the code, we have to complicate things for the programmer by giving each object on the webpage a name. If we name each textbox with a different name (for this example we could use textBox1, textBox2, textBox3, textBox4, and textBox5), we can tell the computer specifically which textboxes to use in our calculations.

(2) Calculate the miles traveled by dividing by the number of miles in the textbox named textBox3 by the number of gallons in the textbox named textBox4, and display the result in the textbox named textBox5.

Assuming we get the textbox names correct when we make up our algorithm and code the algorithm, there will be no question about which numbers to use in the calculations.

Rules for Names

There are a few rules for making up names for objects that you *must* follow. Failure to follow these rules means that your webpage might not work properly. These rules are non-negotiable!

[1] Names can only include letters, numbers, and underscores. You can use up to 31 characters altogether with little risk; while some web browsers will let you use longer names, why would you want to do that much typing?

Names can include upper or lower case letters A-Z and a-z, the numbers 0-9, and the underscore (the character _ which is usually found next to the key for the number 0 on the top of your keyboard). No other characters can be used. In particular, note that you cannot use a dash (–). That's the character on the same key as an underscore, but shows up in the middle of the line instead of on the bottom like the underscore. More importantly, you cannot use any spaces. That's important enough to repeat: names cannot contain any spaces. No other special characters can be used either, but those two need special attention since it's so easy to make that mistake.

[2] Names cannot start with a number. While it may be tempting to use a name like 3rd_generation, you'll have to settle for third generation. If we let names begin with a number, the computer would have difficulty telling the difference in a number and a name (keep in mind that scientific notation has numbers like 3.56E12, and the easiest way to differentiate that

number from a name is by looking at the first character). Can names start with an underscore? Yes, but it's really not a good idea – compilers use names that start with underscores for special purposes, and you can possibly confuse the poor computer if you start your name with underscores, so it's best to stick with letters.

[3] Some names are reserved, and you can't use them. These names are words that compilers or Scripts use for specific reasons, and include words like:

if else for function public private case double new return

There are fewer than 100 reserved words, and the list is slightly different for C# than it is for JavaScript. If this worries you, keep in mind two things: no reserved words include numbers, and no reserved words include underscores.

[4] Upper case letters are not the same as lower case letters, and names with upper case letters are not the same as the same names with all lower case letters. This rule is described as names being **case sensitive** – the names textBox1, TextBox1, and textbox1 are all different. Does this mean you can use all three of those names on the same webpage? Well, yes, but just because you *can* do something doesn't mean that you should. Some web browsers will work just fine if you do that, but other web browsers will get confused – you should use a different name for each object, and the names should never be different based only on whether the name contains capital letters.

Agreements on Name Styles

The rules given above are the only ones you *have* to follow, but many programmers have other agreements on how they will make up names. These differences in style are mostly the result of which programming languages each programmer started out with, but following them can make programs easier to maintain. Almost all programmers start out names for objects and variables with lower case letters, reserving names that start out with upper case letters for naming functions and class names.

Programmers who have used languages such as COBOL and PASCAL are used to making up names that separate words with dashes or underscores, so they would use variable names like starting_odometer_reading and miles_per_gallon.

Camel notation, a more modern approach to making up names, does not use underscores. The first word in a name starts out with a lower case letter, and each additional word in the name starts out with an upper case letter. This leads to variable names like startingOdometerReading and milesPerGallon.

Which style should you use? If you are a working for a company, you should use the style that the company demands. If you are writing for yourself, you can use the style that suits you. The important thing is to use a consistent style. The more consistent your names are, the easier it will be to maintain your programs.

Meaningful Names

In our example above, we used names like textBox1. While this works fine, using numbers to differentiate textboxes starts to break down if we get twenty or thirty textboxes on a webpage and forget whether the cost per widget was in textBox27 or textBox28. It's much better to use names that describe what the object is instead of simply numbering our objects. While it may be tempting to use short names (or even single letters!) for names, it can lead to confusion when you come back to change the program later.

In our example above, we could name our textboxes startingMiles, endingMiles, mileage, gallons, and milesPerGallon. If you rewrite the algorithms using these names you'll see how much simpler they are to understand and implement.

Programmers often use the single letters from i through n as counting variables; this is a throwback to the FORTRAN language that has stuck with us through the years. Since it is likely that another programmer would expect a specific use for the name i, it is best to avoid using those single letters to name an object.

Special note for future C# programmers: you should also avoid using the single letter names e and g since these have special uses for events; you can use them, but it will complicate your code, so it's best to avoid using them to name objects.

Exercises

[1] Indicate which of the following names are valid and which are invalid. For those that are invalid, indicate the reason they are invalid, and suggest an alternate name.

Chapter Seven – Putting Input Objects on a Web Page

Interactive web pages have all sorts of "objects" on them – input text boxes, buttons, drop-down menus, radio buttons, and check boxes. Each object needs its own name using the rules for names from chapter six. Now let's see how to create a few objects!

Input Text Box

<input type=text name=xxxxxx> This creates a box on the screen that the user can use to enter data. You can change how wide the box is by using the size property:

```
<input type=text name="address" size=60>
```

Input Check Box

```
<input type=checkbox name=xxxxxx>
```

This creates a small square box on the screen that the user click on to check and uncheck the box. The user can check or uncheck any number of checkboxes at once.

You can put a name after the box so the user knows what the box is for:

Indicate which pets you have:

```
<input type=checkbox name="cb_cats"> Cats
<input type=checkbox name="cb_dogs"> Dogs
<input type=checkbox name="cb_birds"> Birds
```

You can group and space out buttons by putting them into tables.

Radio Buttons

<input type=radio name=xxxxxx> This creates a small circle on the screen that the user can click on to check and uncheck. The user can only check one radio button at a time. To put radio buttons in groups so that only one gets checked at a time, buttons that are related should have the same name.

Which is your favorite meal?

<input type=radio name="rb1"> Breakfast

<input type=radio name="rb1"> Lunch

<input type=radio name="rb1"> Dinner

What helps most with homework?

<input type=radio name="rb2"> the web

<input type=radio name="rb2"> my roommate

<input type=radio name="rb2"> a tutor

If we didn't use separate names for the two sets of buttons, only one of the six buttons could be checked at a time…and if we used six different names, they could all be checked but could not be unchecked.

Action Buttons

When we want to do something on a webpage, we give the user a button to click:

<input type=button value="What shows on the button" onClick='wait until chapter eight'>

Action buttons don't need names, but you can give them one if you like.

A sample button (which even works!) would be:

<input type=button value="Blue background please" onClick='document.bgColor="lightblue" '>

Special Note: be careful with the use of single and double quotes on this last line! It's best to consistently use single quotes for the onClick property so that you can put other things in double quotes inside them.

Exercises

[1] Create an order page for a pizza. The page should have:

- text boxes for a name and phone number
- radio buttons to choose a small, medium, or large pizza
- check boxes to select several different toppings
- radio buttons to select dine-in, carry-out, or delivery
- an action button that reads "Calculate the Cost"
- a text box labeled Total Cost

Chapter Eight – Making Web Pages Interactive

If webpages were static (never changing), they would be little more than electronic books. Their real strength is the ability to change based on actions by the person who is viewing the webpage, usually in response to a mouse click of some kind. Responses are made to **events**, something that happens while the webpage is active. When an event occurs, it signals the webpage that some kind of change may be required, and it is up to the programmer to describe these changes to the web browser.

Changes are made using **assignment statements**, which look a lot like algebra. To show how these statements work we first need to set up a webpage with an image named image1 that loads a picture named dog1.jpg; you would start by copying our default webpage file, and then add a tag for the image:

```
<img  name="image1"  src="dog1.jpg" >
```

In addition, put a button on the page that says "Show a Cat" using an input tag:

```
<input type=button value="Show a Cat">
```

The purpose of this webpage is to allow the viewer to look at different pictures that we have saved; the page starts out showing a picture of a dog, but in case the viewer is a cat person we want them to be able to change the picture to cat1.jpg. What we need is a way to tell the web browser that we want to display the picture cat1.jpg in the image named image1. We do this by using the **dot operator** and an **assignment statement**.

The Dot Operator

The dot operator is used to allow us to select a specific property of an object. In this example, we want to change the source property of the image:

image1.src

You would read this as “the source property of image1”. Here are some other examples:

address.width	the width property of address
address.value	the value property of address
button1.value	the value property of button1

The name of an object is the one we gave it in the tag, while the property names are the specific properties that each object is permitted to have – not all tags can have all properties.

Using the dot operator gives us an easy way to tell the web browser exactly what we want to change.

The Assignment Statement

Once we can identify a property that we want to change, we need to have a way to tell the web browser to change that property. We do this with a statement that looks a lot like algebra. Here’s the statement we would use to tell the web browser to change the picture that is displayed in image1:

image1.src = “cat1.jpg”

When reading this statement, you should translate the = to be the words “is changed to” and *not* “equals”. This would mean we would read the statement as

The source property of image1 is changed to “cat1.jpg”

An assignment statement does not set two things equal to each other – it changes the value of something! Here are some other examples:

address.width = 20	the width property of address is changed to 20
address.value = “5 Ava Drive”	the value property of address is changed to 5 Ava Drive
button1.value = “Dog”	the value property of button1 is changed to Dog

You probably noticed that there are spaces around the assignment operator. While we could write the code without the spaces, the spaces make it easier to read and change the code. While tag properties cannot have spaces around the assignment operator, the assignment statement allows them.

The onClick Event

In addition to having properties, tags can have code that tells the web browser what to do when certain **events** take place. An event is something that happens because of an action the viewer

has taken – pressing a key, moving the mouse, and clicking a mouse button are the most common events.

Buttons have a special event that the web browser recognizes and reacts to – the onClick event. This occurs when the viewer uses the mouse to click on the button, and the programmer can tell the web browser what to do by putting code into the button's tag. The programmer puts their code inside the tag, and identifies the event by using the word onClick= followed by the instructions for the event. These instructions have to be inside quote marks, which can lead to a slight problem. Suppose we wanted to change the image to cat1.jpg; using the assignment statement above, we would want to tell the web browser:

onClick="image1.src = "cat1.jpg" "

How can the browser know that we aren't finished with the onClick instruction when it gets to the second quote mark?

onClick="image1.src = "

This wouldn't work at all since it would leave an incomplete statement

image1.src = "

We get around this problem (as mentioned before) by using single quotes to enclose the onClick event:

onClick='image1.src = "cat1.jpg" '

The entire tag for that button would therefore read

<input type=button value="Show a Cat" onClick='image1.src = "cat1.jpg" '>

Changing The Web Page Colors

As you may have discovered from some fumbling typing errors, if you put an invalid property into a tag, the web browser simply ignores it. Not all tags can have names; you cannot, for example, change the properties of a table or a list by giving it a name and using an assignment statement. While this won't be a problem for us (we will concentrate on changing input objects), one special tag does get a fixed name – the body tag is named document. This name persists even if you try to change it. This means that if we wish to make a change to the entire web page, we can use the properties of the body tag to make the change. The simplest examples of this are changing the background and foreground colors of the webpage using onClick events:

```
<input type=button value="Go Tan" onClick='document.bgColor= "Tan" '>
<input type=button value="Go Tan" onClick='document.fgColor= "White" '>
```

Multiple Instructions

Quite often we will want to do more than one thing when an event occurs. We do this by using the **statement separator**, which is the semi-colon. If we only have one statement, it ends when we put in the closing quote tag, but if we have more than statement we put a semi-colon before

each additional statement. If we wanted to change both the foreground and the background colors on the webpage, we could do that with only one button using this trick:

```
<input type=button value="Go Tan and White"
        onClick='document.fgColor= "White"; document.bgColor= "Tan" '
```

It's very, very important to always close a quote mark on the same line where you open it; things in quotes cannot continue onto the next line. As shown above, it's okay to have a tag break onto more than one line, but only if you are careful with the quote marks.

Important Note: to get long lines using Notepad you may have to turn off word wrap.

Exercises

[1] Create a webpage that has some text and a picture and at least four different buttons, each of which changes the picture to something different.

[2] Add several buttons to your web page that change the size of the picture (hint: it is the image tag that has a width that determines how big the picture will be on the web page).

[3] Add several buttons to your web page from exercise [1] that change the background and text colors to different combinations.

Chapter Nine – Changing Textboxes

In the last chapter we learned how to write onClick events to get the web browser to execute assignment statements for us. This chapter we are going to use these statements to change the values of textboxes.

One of the types of objects we learned to put on a page is the textbox – it can hold a series of letters, numbers, and special symbols…anything you can type can go into a text box. We can put something specific in a textbox by using the value property:

```
<input type=text  value="The capital is Indy"  name="capital">
```

If we put this line into a web page, we would get a textbox, but it would start out with a visible string instead of blanks. The user can always type in the textbox to change its value, but we can use assignment statements to change the value as well. We can assign the value of a textbox a string, we can copy a value from one textbox to another, or we can put a number into a textbox.

To change the value in a textbox to a new string, you would use the same kind of assignment statement we saw in the last chapter:

```
capital.value = "Albany"
```

The quote marks are very important; if you leave them out, the results may not be what you expected.

To copy a value from one textbox to another, you would also use an assignment statement:

```
textBox1.value =  textBox2.value
```

This statement would copy whatever is in textBox2 into textBox1 and they would then both have the same value.

To put a numeric value into a textbox, you would simply set the value equal to the number using an assignment statement:

```
textBox1.value = 436.72
```

If you wanted to, you could put the numeric value into the textbox using a string; the result would be the same:

```
textBox1.value = "436.72"
```

In the next chapter we'll see how to do calculations rather than simply putting a specific value into a textbox.

Exercises

[1] Add several buttons to your web page that change the size of the picture (hint: it is the image tag that has a width that determines how big the picture will be on the web page).

[2] Add several buttons to your web page from exercise [1] that change the background and text colors to different combinations.

Chapter Ten – Do The Math…er, Arithmetic

Assignment statements can be used to do calculations, allowing us to turn a $2,000 computer into a $10 calculator. While this isn't cost effective, it will show us how to get a computer to do the math for us. While computers have no common sense, they are amazingly accurate – as long as we give them the correct values, and the correct instructions, they will always get the same correct answer.

We normally think of four operations we learned to do in elementary school – add, subtract, multiply, and divide. Computer languages use four specific symbols to tell programs to do these operations. Look at the numeric keypad on your keyboard, and you will notice that there are four symbol keys surrounding the numbers:

/ * - +

These are the keys that generate the correct symbols for division, multiplication, subtraction, and addition.

The x key produces a letter, not a symbol, so computer languages don't use the x for times, they use the * symbol for multiplication. There is no key for a horizontal line with dots on either side of it, so we use the / key for division.

Building a Calculator

We can build a simple calculator by setting up a web page with three textboxes, four buttons, and a little bit of text. Here's the code that should go inside the body:

```
<input   type=”text”      name=”first_number”   value=”32”   > <br>
<input   type=”text”      name=”second_number”   value=”4”   > <br> <br>

<input   type=button     value=”multiply” >
<input   type=button     value=”divide” >
<input   type=button     value=”subtract” >
<input   type=button     value=”add” >            <br><br>

<input   type=”text”      name=”the_answer”     value=” ”   > <br>
```

Of course, your webpage will look much nicer than this one because you'll include one or more pictures and either background colors or wallpaper, and you might even use a table to carefully space out the buttons and line things up in the middle of the page.

We should be able to make the first button work by adding an event to its tag:

```
onClick='the_answer.value = first_number.value * second_number.value'
```

Add this inside the multiply button's tag, making sure to stay inside the tag markers, and be careful not to go on to a second line (remember to turn off word-wrap if you're using Notepad). Once you have this in the tag, reload your page and click on the multiply button to see what happens – if the number 128 shows up in the answer textbox you've got it! To make sure the button really works for more than one calculation, you can change the values that are in the first two textboxes and click the button again to check the results.

Once you're sure the button works properly, set up the other three buttons as well, replacing the multiplication symbol with the appropriate symbols for division, subtraction, and addition, and check the results…but don't be too surprised when the addition button doesn't work properly.

Concatenation

Textboxes were designed to show strings – the value they contain is a string, and it is displayed inside the box that shows up on the webpage. When we tell the web browser to multiply, divide, or subtract the values that are inside two textboxes, it knows that what we really want to do is convert the strings to numbers and then perform the calculation. When we use the + sign, however, it tells the web browser to do something a little bit different, something called concatenation. This operation takes two strings and appends the second one to the first one. If, for example, the first two textboxes contained "34" and "57", the result of concatenation would be "3457". If the textboxes contained "house" and "fly", the result of pressing the concatenation button would be "housefly".

To avoid more confusion, go back and change the value of the last button from Addition to Concatenation. When you test this button, it should copy the strings from the first two textboxes and put them together and store the result in the answer textbox.

So how do we do addition? The best answer would be to use a different symbol for concatenation than we use for addition, but apparently that didn't occur to the early programmers – some languages use & for concatenation and + for addition, and that solution

works fine. To get the web browser to do addition, however, we have to convert the strings in the two boxes into numbers. There are several different ways to do this, but the easiest way is to multiply each of the strings by 1. Since the web browser knows to convert strings to numbers to do multiplication, this will get it to convert the numbers for us (we could also use conversion functions, but that's a lot more typing). Note that you have to convert *both* of the strings before you can do addition – if you only convert one of them, it's entirely likely that the web browser will convert it back to a string and perform concatenation. Add a fifth button for addition, and try multiplying each of the textbox values by 1 before you add them:

```
onClick='the_answer.value = 1 * first_number.value  +  1 * second_number.value'
```

Who would have expected that addition would be more complicated than multiplication or division? Make sure your web page works okay, and then do the exercises.

Chapter Eleven – Simplifying Your Typing With Functions

A few chapters ago we learned how to have multiple statements executed when an event occurs using the statement separator (the semi-colon). Since our statements had to be inside a set of quotes, and since quotes cannot extend past the end of a line, this limits how many instructions we can give the web browser. It also results in some lines of code that are difficult to read and even more difficult to change. In order to make our web pages more manageable, we can make use of functions.

A New Default File

Functions can be placed at various places on a webpage, but that would mean looking all over the code for the webpage if we needed to make changes to a function. A better idea is to always put them in the same place. We noted earlier that the Head section of the code is used to describe the webpage and the Body section of the code is used to describe the content, so it will be easier to maintain our functions if we put them in the Head section. To do this we will be adding a new tag: the Script tag.

Copy your default.htm file and rename the copy jsdefault.htm. Then go in and add lines that indicate we will be using the JavaScript language to write our functions. This new file should look like this:

```
<html>
 <head>
  <title>  The title goes here  </title>
  <script language=”JavaScript”>
        // functions go here
  </script>
 </head>
 <body>
        // the webpage goes here
 </body>
</html>
```

From now on you should start out new webpages by copying jsdefault.htm instead of default.htm.

Writing Your First Function

Functions aren't very different than the assignment statements we've been writing, they're just a way of making them easier to maintain. Each function consists of two parts, the header and the body (sound familiar?)

The header of the simplest function simply indicates that you are writing a function and it also tells the web browser what the name of the function is. Names follow the same rules we've seen for names of objects on our webpage, with special emphasis on one important rule: never use the same name twice. If you use a name for a textbox, you cannot use that same name again for a function. Each function must have its own unique name. Most programmers start functions with a capital letter so that they can be identified as function names and not object names.

After the name of the function we put a pair of regular parentheses; these are place holders for arguments, which we'll get to later. All functions *must* have these parentheses after the function name.

The body is a series of lines of code, each one ending with a semi-colon. When we use a semi-colon in this way it is called a **statement terminator** rather than a statement separator. It's usually a good idea to put each statement on a separate line so that we can easily read and change them when necessary – just think of how difficult it was to read onClick commands that all had to come within a single line of code and you'll appreciate the freedom to use multiple lines.

In order to show where the lines of code start and end we use a pair of curly braces: { } There is no limit to the number of lines that can be in a function, they simply have to be inside the braces. To make sure that you don't forget to put in the closing brace, it's a good idea to key in the function header, key in the open and close braces on separate lines, and then go back and add the code that goes between them.

Earlier we saw how to write the code to change the colors of a webpage; here's the code that would do this in a function named Change_Colors:

```
function Change_Colors (  )
  {
        document.fgColor = "White";
        document.bgColor = "Tan";
  }
```

We have indented the various portions of the function to make it easier to read, but that isn't necessary – the function will work fine if you line up everything on the left, or put everything on one line, or even split some of the statements onto multiple lines, but it won't be as easy to read and change.

The tag for a button that would call this function might look like this:

```
<input  type="Button"  value="Change to White and Tan" onClick='Change_Colors(  )" >
```

Exercises

[1] Create a webpage called names.htm that uses an unordered list to show the rules for making up names; be sure to include the special rules for functions. The webpage should start out without any special colors for the text or background. Add the button and the function shown above so that the user can change the colors.

[2] Add at least two more buttons and two more functions that change the colors to different color combinations so that the user can select how the webpage looks when they read it. Be sure to give these new functions different names!

[3] Create a webpage called rectangle.htm that has input boxes for the width and the height of a rectangle; be sure to label which textbox is which, and include a picture of a field or a pool or something that is shaped like a rectangle. There should also be two more boxes labeled area and perimeter and a button labeled Calculate. Write a function named Do_Calculations that calculates the area and perimeter and puts those values into the output textboxes. Hint: The area is simply the width times the height; the perimeter is the sum of the four sides, which is also equal to two times the width plus two times the height. Your function only needs two lines of code in the body.

Chapter Twelve – Passing Arguments to Functions

If we wanted to have a function to change the background color to tan, it might look like this:

```
function Change_Background_To_Tan (  )
  {
        document.bgColor = “Tan”;
  }
```

and the onClick command would simply be

```
onClick=’Change_Background_To_Tan( )’
```

We then would add a Change_Background_To_Green, a Change_Background_To_Blue, and so on and so on until we had covered all the colors we wanted to allow. All of these functions would be identical except for the name and the color we set the background to. Computer programmers should be lazy – why code something a dozen times when you can code it only once? Fortunately there is a way to write code that allows the program to change to the color of our choice – we can pass values to the function to tell it what color we want to change the background to by using arguments.

Passing Values To a Function

For each of the color changing functions, the value we want to set the color to is a string, so if we can pass the function a string that tells it which color to use we can write only one function

and call it differently for each color. Up until now when we wrote or called a function we had an empty pair of parentheses – this is where we put the value that we want to pass to the function. We can call the function many times using different colors as arguments to the function:

```
onClick='Change_Background_Color( "Tan")'
onClick='Change_Background_Color("Geen" )'
onClick='Change_Background_Color( "Blue")'
```

These function calls each have one argument inside the parentheses, the color that we want to change the background to. We have to be careful to put the color in quote marks because we are passing a string to the function.

Of course, in order for this to work, we have to change our function as well – if we plan to call the function with an argument, we have to write the function in a way that accepts the argument. We do this using a variable – think of it as a name for a value in an invisble textbox. When we call the function it places the values we are calling it with into the invisible textbox, and we can then use that value in the function. Here is the code that would implement this new function:

```
function Change_Background_Color( new_color )
  {
        document.bgColor = new_color;
  }
```

Since it isn't really a textbox, we don't have to use the .value property – the variable has nothing but a value, so it can go on either side of an assignment statement, just like a variable in an algebra statement. When the function is called, the value that was inside the parentheses is said to be passed to the function; the value is taken out of the calling code and placed into the variable that is listed in the function. Variables have names that follow the same rules as the names for objects and functions, and usually start with a lower-case letter.

When we call a function, we can call it with the value from a textbox instead of a fixed string. This code would let the user input the color they wanted for the background while calling the same function we already wrote:

```
<input type=text  name="color_choice">
<input type=button value="Change to your color"
                onClick='Change_Background_Color( color_choice.value)'>
```

This would call the function using the string that is in the textbox named color_choice; that value would be passed into the variable new_color and would be the color that shows up. The danger in allowing the user to input the color is that they input an invalid color, either by selecting a color that isn't supported (such as "Marble") or by accidentally mis-typing the color (such as "Purpel"). The results may be unpredictable!

Passing Multiple Values to a Function

The function we wrote in the last chapter changed two colors, the background color and the text color. If we wanted to replace a lot of functions from the exercises in the last chapter with a

single function, we could do it by passing two values instead of one. As always with functions, we start by writing the onClick commands so we know what we want our function to look like:

```
onClick='Change_Both_Colors( "Tan", "White")'
onClick='Change_Both_Colors( "LightGreen", "Brown")'
onClick='Change_Both_Colors( "White", "Black")'
```

We could write the code for the function like this, but it would be a mistake:

```
function Change_Both_Colors( c1, c2)
  {
      document.bgColor = c1;
      document.fgColor = c2;
  }
```

If we came back to maintain this code a year from now, we might not remember whether c1 was the foreground color or the background color, and we might make a mistake when we changed the code.

In each case we are passing the background color first and the text color second, so we want to be sure to make up variable names that reflect these choices. This is referred to as **self-documenting code** since the variable names tell us what the variables are for. Here is a better function that does the same thing:

```
function Change_Both_Colors( new_fgcolor, new_bgcolor)
  {
```

```
    document.bgColor = new_fgcolor;
    document.fgColor = new_bgcolor;
}
```

Exercises

[1] Create a webpage called names.htm that has the rules for making up names that starts out without colors for the text or background. Be sure to add the special rules for functions. Add the button and the function shown above so that the user can change the colors.

[2] Add at least two more buttons and two more functions that change the colors to different color combinations so that the user can select how the webpage looks when they read it.

Chapter Four

RESULTS OF STATISTICAL ANALYSIS

A total of four classes that contained 71 students were reviewed in this study. The classes can be broken down into three distinct categories based on the method used to teach the course:

Method 1: Experimental method teacher 1 (one section of the course)

Method 2: Traditional method teacher 1 (one section of the course)

Method 3: Traditional method teacher 2 (two sections of the course)

Student achievement in the course was used to divide the students into two groups: Success for students who earned an A, B, or C and Non-Success for students who earned a D or an F or who withdrew from the course.

The raw data for these classes was as follows (success in the course is the dependent variable):

Table 1 –Student Success Totals

Results	Method 1	Method 2	Method 3	Total
Success	11	7	31	49
Non-Success	1	9	12	22
Total	12	16	43	71

Simple raw percentages shows a success rate of 92% using the experimental approach compared to an average of 64% for the other sections of the course.

Data were also collected to classify students as male or female and as freshmen or upperclassmen. Where available, each student's high school GPA and SAT scores were recorded. For some students an SAT score was not available, but the student did have an ACT score. The College Board and the ACT periodically publish a Concordance that shows corresponding scores for the SAT total score and the ACT total score; in some instances where a total SAT score was unavailable a score was taken from the Concordance by using the total ACT score.

The following table shows how the 71 students were distributed using the three groupings of Gender (male or female), Status (freshman or upperclass), and teaching method (experimental, traditional with teacher 1, and traditional with teacher 2). Each of the three major classifications is shown separately.

Table 2 – Distribution of Students

		Frequency	Parameter coding
		(1)	(1)
Gender	1.00	60	1.000
	2.00	11	.000
Status	1.00	27	1.000
	2.00	44	.000
Method	1.00	12	1.000
	2.00	16	.000
	3.00	43	.000

The crosstabulation of student success by method produced the following result:

Table 3 – Crosstabulation of Student Success

		Method 1	Method 2	Method 3	Total
Success	Count	11	7	31	49
	Expected Count	8.3	11.0	29.7	49
	% within Success	22.4%	14.3%	63.3%	100.0%
	% within Category	91.7%	43.8%	72.1%	69.0%
	% of Total	15.5%	9.9%	43.7%	69.0%
Non-Success	Count	1	9	12	22
	Expected Count	3.7	5.0	13.3	22.0
	% within Success	4.5%	40.9%	54.5%	100.0%
	% within Category	8.3%	56.3%	27.9%	31.0%
	% of Total	1.4%	12.7%	16.9%	31.0%
Total	Count	12	16	43	71
	Expected Count	12.0	16.0	43.0	71.0
	% within Success	16.9%	22.5%	60.6%	100.0%
	% within Category	100.0%	100.0%	100.0%	100.0%
	% of Total	16.9%	22.5%	60.6%	100.0%

The Chi-Square results for this classification is shown below:

Table 4 – Chi-Square Test

	Value	df	Asymp.Sig. (2-sided)
Pearson Chi-Square	7.846	2	0.020

For the logistic regression, the categorical variables used were the method, the class standing, and the gender. The following table shows the changes to the model as less significant factors were removed:

Table 5 – Variables in the Equation

		B	S.E.	Wald	df	Sig.	Exp(B)
Step 1(a)	Gender(1)	1.289	1.151	1.256	1	.262	3.631
	Status(1)	1.192	.618	3.716	1	.054	3.293
	Newmeth			7.765	2	.021	
	Newmeth(1)	-1.529	.776	3.884	1	.049	.217
	Newmeth(2)	1.525	.553	7.603	1	.006	4.596
	Constant	-2.673	1.139	5.511	1	.019	.069
Step 2(a)	Status(1)	1.270	.609	4.353	1	.037	3.561
	Newmeth			8.494	2	.014	
	Newmeth(1)	-1.681	.758	4.918	1	.027	.186
	Newmeth(2)	1.575	.541	8.457	1	.004	4.829
	Constant	-1.608	.509	9.971	1	.002	.200

a Variable(s) entered on step 1: Gender, Status, Newmeth.

The percentages of students correctly classified in each category as a result of the logistic regression is shown below:

Table 6 – Classification Table

Observed			Predicted		
			Success		Percentage Correct
			Correct	Incorrect	
Step 1	Success	1.00	43	6	87.8
		2.00	14	8	36.4
	Overall Percentage				71.8
Step 2	Success	1.00	49	0	100.0
		2.00	18	4	18.2
	Overall Percentage				74.6

As can be seen, after gender was eliminated from the equation, the model was over 70% correct in classifying students who were successful into the success category, but misclassified some of the students who weren't successful as being successful. Overall, the final model was 75% accurate in classifying the students.

Complete records for GPA and SAT scores were not available for all students, so those factors were not included in the analyses shown above. However, using the available information shows the scores by treatment method.

Some simple statistical comparisons were done between the experimental class and the traditional classes. The first showed that there was little difference in the high school GPA for the two different groups:

Table 7 – High School GPA By Teaching Method

	Mean	Standard Error	95% Lower Bound	95% Upper Bound
Experimental	3.519	.157	3.202	3.835
Traditional	3.530	.141	3.245	3.815

Similar tests on the SAT scores revealed that the more successful experimental group had lower Total SAT scores (where scores were available):

Table 8 – SAT Scores by Teaching Method

Newmeth	Mean	Std. Error	95% Confidence Interval	
			Lower Bound	Upper Bound
1.00	1033.000(a)	55.817	920.434	1145.566
2.00	1203.750(a)	71.329	1059.901	1347.599
3.00	1126.966(a)	41.954	1042.357	1211.576

a Based on modified population marginal mean.

Chapter Five

CONCLUSIONS AND FUTURE RESEARCH

Materials Developed

The materials developed for this study differed from traditional introductory computer programming classes by using the Ausubel model of scaffolding student learning. This was accomplished by inserting a segment of materials at the beginning of the course that focused on introducing students to concepts using a text editor and web browser they were familiar before proceeding to the more difficult topics that followed. In this case, the introductory portion of the course taught students to build increasingly more challenging web pages using HTML and JavaScript before moving on to C#. In this material students were introduced to the concepts of objects, properties, variables, and functions and an edit/view/debug cycle in preparation for the more involved edit/compile/debug cycle used with .Net and C#.

The use of an Integrated Development Environment makes writing, compiling, testing, and debugging easier for accomplished programmers, but requires students to master a tool they probably have no previous exposure to. In the HTML portion of the course students used a simple text editor (Notepad) to create text that would show up in a web page that was viewed by a web browser. Since all the students had experience using both of these tools, they had no difficulty creating initial web pages. The course then proceeded to build more and more complex web pages using lists, tables, and graphics, and as students used these tools their web pages often did not look the way they had envisioned. This required the students to examine the output, review their text file, and make changes to the text to fix the problems, after which

they could look at the new web page. This introduced the students to an edit/view/debug cycle that was easy for them to understand; changes could be a simple as fixing spelling errors or as complicated as fixing tags.

Web pages are simple text that is formatted using tags. While some tags are stand alone, many useful tags are container tags (tags that must be used in pairs to show where the effect of a tag both begins and ends). When working with lists and tables, container tags often contain other tags, and learning to cope with these combinations of containers when doing lists and tables prepared students for the block structure nature of C#.

Tags can be used to create objects (such as textboxes and pictures that appear on the web page), and these tags can modify the size or colors of these objects by defining the attributes of the objects in their tags. The course simply renamed these attributes, calling them properties to match up with the nomenclature used in Java and C# and other modern languages. Students created web pages that used and modified visual properties such as text, color, and size of buttons and pictures so that their first experiences with objects and properties were with tangible objects and they could see the results of changing the properties.

Students spent several days putting objects onto web pages before they did any programming with them. This allowed them to become familiar with the concepts of textboxes, radio buttons, check boxes, and images and their properties without the need to do any programming. The concept of a string was introduced so that text could be placed on buttons and in textboxes, again, without any programming.

Once students were comfortable with objects, and could easily add them to their web pages, the OnClick events for action buttons were introduced. Initially simple single assignment

statements were used to change colors or sizes or even pictures by simply changing existing properties of the objects that were on the form. In order to select which object was to be changed, students learned to give names to the objects (using the same naming conventions that apply in C# programs). Changing properties of specific objects also required the use of the dot operator (but only for properties, not for methods).

After learning to change the text property in a textbox, students learned to build more complex strings by using concatenation and by doing implicit conversions from numbers to strings. The dangers of implicit conversions were discovered by using arithmetic on textbox strings, leading to concatenation instead of addition. Explicit conversions were introduced to avoid these mistakes.

OnClick events that were more complex were introduced, requiring students to separate statements with semi-colons (the statement terminator which is also used in C#). OnClick events must be contained in a single delimited string which must be on a single line, leading to some very long text lines of code even for something as simple as changing both background and text colors on a web page. Students were generally relieved when JavaScript was introduced to allow them to move code from OnClick events to functions that could span multiple lines. This response was significantly different from the response students in the traditional classes had to the introduction of functions, which made their programming more difficult without any apparent payback. The ability to pass values to functions to reduce the number of functions that needed to be written also made programming easier for the students (the primary example was passing the colors the students wished to use on the page, again allowing a visual result immediately).

At this point the code for the web pages was beginning to be significant, and most students were dealing with more complex debugging problems. Each student was then given a web page that contained numerous errors and asked to fix the page. The errors included unmatched tags, missing or mismatched quote marks, misspelled names, incorrect punctuation, incorrect names for picture files that were to be included on the web page, and concatenation where addition would have been appropriate. Students were given a class session to do their best to repair the page, and it proved to be a frustrating task for them (most of the page didn't even show up until an initial error was fixed). Each student was asked to make a list of the kinds of errors they found, and to make suggestions on what the web browser should do when it encountered errors. From these lists, the class produced a list of things that they wished the web browser would fix or point out; this led to a discussion of compilers and why we have them and how useful it would be to have a tool that could identify where errors in a program were. At this point the class was ready to move on to coding in C#.

After a quick tour of the .Net IDE and the coding of a simple "Hello World" program (which only requires one line of code that is similar to what was done in JavaScript), the students spent several days building forms. After learning how to place action buttons, text boxes, labels, check boxes, and radio buttons on a form, students were given several assignments that essentially rebuilt non-functional web pages they had built before. Overall they were pleased at how much easier it was to control the objects, where they appeared on the form, and how the form looked.

C# is a strongly-typed language, so the first stumbling block students faced was the use of explicit conversion functions to take numbers out of textboxes or to put numbers into

textboxes, but having seen the potential for mistakes with JavaScript they already understood the need for the conversions. Several programs were written doing assignments.

If statements were introduced that used checkboxes and their checked property rather than the traditional mathematical comparisons. After students were comfortable with checkboxes and simple if statements, the radio buttons were used to introduce the else statement and the construction of a series of if…else if statements. A typical program that used checkboxes and radio buttons was a program that calculated the cost of a pizza using radio buttons for the size and checkboxes to add extra toppings. After students created several programs that did arithmetic based on collections of checkboxes and radio buttons, the other comparison operators were introduced.

While, do while, and for loops were done next, using a similar approach to that used in the traditional sections.

Finally, the semester ended with several weeks of writing user-defined functions. The initial functions were void functions that simply moved code from an event to a function, similar to the way functions were used on the web pages. Some functions were done that passed variables by value (starting with the same color-changing function done before), and finally passing variables by reference.

Results

The experimental group was made up of students who were randomly assigned to one section of the course while other students were assigned to a section that was taught using the traditional approach (additional archival data was added for two similar sections of the course for each teacher from a prior year that each used a traditional approach to teaching). The C#

assignments for the two sections were similar, and the final exams were also similar and included a number of nearly identical questions.

The experimental group taught under like conditions was more successful than the group taught using the traditional approach. Statistical studies verified that differences in success rates were probably not a result of differences in male/female ratios in the classes, class standing (freshmen versus upper class students), high school GPA, or SAT scores.

Students for all sections filled out surveys at the end of each course that indicated their reaction to the classes, and the students in the experimental section gave the teacher better ratings than students taught in the traditional course sections, and also showed more interest in continuing to take computer science courses.

While it is too soon to measure eventual success in a computer science major or minor for the students, a higher percentage of the students in the experimental section took at least one more computer science course.

Table 9 – Student Retention By Teaching Method

	Experimental	Traditional
Total	12	59
Continuing	7	21
Percentage	58%	36%

Further Study

The most important research that could be done would be additional studies that test to see if the results shown here can be verified.

The study by Qusay (2004) showed positive results from using HTML/JavaScript prior to teaching Java, and this study showed improvements in student success using HTML/JavaScript prior to teaching C#. Java and C# are very similar languages, so similar results are not a surprise. It might be revealing to follow a similar course design using C++ or Visual Basic or even a functional language such as LISP to see if some of the scaffolding nature of the introductory section of the course improves student learning.

While statistical analysis such as was done in this dissertation can identify correlations, it cannot prove cause and effect. There could be other reasons that the delay before beginning work with C# had a positive effect on the students, such as the reduction of new information at the start of a semester when most students are adjusting to a new setting. Testing to see if this delay is the primary benefit of the experimental approach used could be done by replacing the HTML/JavaScript portion of the course with an appropriate alternative: (1) using material on the nature of computer science without any programming content (many CS0 courses cover this type of material); (2) an introduction to using word processors, spreadsheets, and databases; or (3) an introduction to discrete mathematics topics that are related to computer science.

It may also yield important information to follow the long-term progress of students who receive the experimental treatment. Due to time constraints, retention is currently based on whether or not students in the experimental section took an additional computer science course in the two semesters after the introductory course. Comparisons of student success in these additional courses will indicate how well these students were prepared for additional studies. In addition, information will be tracked to determine what percentage of these students complete a major or minor in Computer Science compared to students who came through the traditional sections.

References

Ben-Ari, Mordechai (1998). "Constructivism in Computer Science Education", *The Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education,* 257-261. New York City, New York: Association for Computing Machinery.

Bennedsen, Jens, Caspersen, E. Michael (2007). "Failure Rates in Introductory Programming", *Inroads*, *39* (2), 32-36.

Bergin, Joseph, Kelemen, Charles, McNally, Myles, Naps, Tom, Goldweber, Mike, Power, Chris, & Hartley, Stephen (2001). "Non-Programming Resources for an Introduction to CS: A Collection of Resources for the First Courses in Computer Science", *ACM SIGCSE Bulletin*, 33(2), 89-100.

Cooper, Steven, Dann, Wanda, & Pausch, Randy (2003). "Teaching Objects-First In Introductory Computer Science", *ACM SIGCSE Journal*, 35(1), 191-195.

Dale, Nell B. (2006). "Most Difficult Topics in CS1: Results of an Online Survey of Educators", *Inroads,*38(2) 49-53.

DuHadway, Linda P., Clyde, Stephen W., Recker, Mimi M., & Cooley, Donald H. (2002). "A Concept-First Approach for an Introductory Computer Science Course", *Journal of Computing Sciences in Colleges*, 18 (2), 6-16.

Pears, Arnold, Seidman, Stephen, Malmi, Lauri, Mannila, Linda, Adams, Elizabeth, Bennedsen, Jens, Devlin, Marie, & Paterson, James (2007). "A Survey of Literature on the Teaching of Introductory Programming", *ACM SIGCSE Bulletin*, 39(4),204-223.

Phillips, Andrew T., Stevenson, Daniel E., & Wick, Michael R. (2003). "Implementing CC2001: A Breadth-First Introductory Course for a Just-In-Time Curriculum Design", *ACM SIGCSE Bulletin*, 35(1) 238-242.

Mahmoud, Qusay H., Dobosiecwicz, Wlodek, & Swayze, David (2004). "Redesigning Introductory Computer Programming with HTML, JavaScript, and Java", *The Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education,* 120-124. New York City, New York: Association for Computing Machinery.

Gonzales, Graciela (2006). "A Systematic Approach to Active and Cooperative Learning in CS1 and its effects on CS2", *ACM SIGCSE Bulletin*, 38(1), 133-137.

Guzdial, Mark (2001). "Use of Collaborative Multimedia in Computer Science Classes", *ACM SIGCSE Bulletin*, 33(3), 17-20.

Guzdial, Mark (2003). "A Media Computation Course for Non-Majors", *ACM SIGCSE Bulletin*, 35(3),104-108.

Hermann, Nira, Popyac, Jeffrey L., Char, Bruce, & Zoski, Paul (2004). "Assessment of a Course Redesign: Introductory Computer Programming Using Online Modules", *The Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education,* 66-70.

Kurkovsky, Stan (2007). "Making Computing Atractive For Non-Majors: A Course Design", *Journal of Computing Sciences in Colleges*, 22(3), 90-97.

Nelson, C.V., & Neff, K.J. (1990). "Comparing and Contrasting Neural Net Solutions to Classical Statistical Solutions". Paper presented at the Annual Meeting of the Midwestern Educational Research Association, Chicago, IL. (ERIC Documentations Reproduction Service No. ED 326 577).

Reges, Stewart (2006). "Back to Basics in CS1 and CS2", *ACM SIGCSE Bulletin*, 38(1), 293-297.

Roberts, Eric (2004). "The Dream of a Common Language: The Search for Simplicity and Stability in Computer Science Education"*, The Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education,* 115-119.

Sloan, Robert H. & Troy, Patrick (2008). "CS 0.5: A Better Approach to Introductory Computer Science for Majors", *Proceedings of the Thirty-Ninth SIGCSE Technical Symposium on Computer Science Education*, 271-275. New York City, New York: Association for Computing Machinery.

Vilner, Tamar, Zur, Ela, Gal-Ezer, Judith (2007). "Fundamental Concepts of CS1: Procedural vs. Object Oriented Paradigm – A Case Study", *ACM SIGCSE Bulletin*, 39(3), 171-175.

Wilson, B.C. & Shrock, S. (2001). "Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors", *ACM SIGCSE Journal*, 33(1), 184-188.