

PROG 110 C# Programming Book

For use at Bellevue College

3000 Landerholm Circle SE Bellevue, WA 98007-6406

Version 2.00

March 18 2018

COPYRIGHTS AND USAGE RESTRICTIONS

* Approximately a fourth of the content of this book was derived from parts of the first two chapters of Rob Miles' great Yellow Book, V8.2 November 2016. It has been modified extensively by Kurt Friedrich for use at Bellevue College, with V1.01 first used in my Prom 110 course starting in January 2018. V2 includes all the updates I made while teaching that course. Rob Miles graciously granted me the right to modify and extend his book to create my own version more tailored to our courses at Bellevue College. My primary goal was to create a free book for BC students that matched our program and courses.
* © Kurt Friedrich 2018, all rights reserved.
* Individual copies can be printed for personal use. No large-scale reproduction, copy or transmission of this publication may be made without written permission from the author. The author can be contacted at: kurt.friedrich@bellevuecollege.edu

Contact

If you have question, suggestions, or corrections, please send them to: kurt.friedrich@bellevuecollege.edu

Contents

[Chapter 1 – Programming with C# 5](#_Toc509154831)

[Types of software 5](#_Toc509154832)

[The C# language 5](#_Toc509154833)

[Ready, Shoot, Aim 6](#_Toc509154834)

[What Comprises a C# Program? 7](#_Toc509154835)

[Class, Object, Method, Property 8](#_Toc509154836)

[Names, Identifiers, Key Words, and Strings 10](#_Toc509154837)

[Check your understanding 11](#_Toc509154838)

[Chapter 2 – Integrated Development Environment 13](#_Toc509154839)

[Note for the entire book about cutting and pasting code 18](#_Toc509154840)

[Check your understanding 18](#_Toc509154841)

[Chapter 3 – Our First Useful Program 19](#_Toc509154842)

[Putting in more detail 20](#_Toc509154843)

[What the program needs to actually do. 20](#_Toc509154844)

[Real code for our window problems 21](#_Toc509154845)

[Comments (and proper code formatting) 24](#_Toc509154846)

[Debugging 25](#_Toc509154847)

[Check your understanding 25](#_Toc509154848)

[Chapter 4 – Storing Data in Variables 27](#_Toc509154849)

[Variables and Data 27](#_Toc509154850)

[Storing real values 29](#_Toc509154851)

[Storing Text 31](#_Toc509154852)

[Storing strings 32](#_Toc509154853)

[Storing Booleans 32](#_Toc509154854)

[Choosing the type of your variables 33](#_Toc509154855)

[Naming Variables: Identifiers 33](#_Toc509154856)

[Giving Values to Variables 34](#_Toc509154857)

[Expressions 34](#_Toc509154858)

[Changing the “Type” of Data 35](#_Toc509154859)

[Constant variables 38](#_Toc509154860)

[Check your understanding 39](#_Toc509154861)

[Chapter 5 – Making Decisions 42](#_Toc509154862)

[Conditional Execution - if 42](#_Toc509154863)

[Conditions and Relational Operators 43](#_Toc509154864)

[Combining Logical Operators 46](#_Toc509154865)

[Using the else and if else structures for better logic flow 48](#_Toc509154866)

[Applying our new knowledge to our window program 49](#_Toc509154867)

[Range Checking with if statements 51](#_Toc509154868)

[Using the switch statement to make decisions 51](#_Toc509154869)

[Check your understanding 54](#_Toc509154870)

[Chapter 6 - Loops 57](#_Toc509154871)

[while loop 57](#_Toc509154872)

[do while loop 59](#_Toc509154873)

[for loop 63](#_Toc509154874)

[Nested Loops 66](#_Toc509154875)

[Variables and Scope 69](#_Toc509154876)

[Practice with a Dice Game 73](#_Toc509154877)

[Check your understanding 76](#_Toc509154878)

[Chapter 7 – Writing to the Console and Basic String Usage 79](#_Toc509154879)

[Concatenation 79](#_Toc509154880)

[Comparing two strings 79](#_Toc509154881)

[Length property 80](#_Toc509154882)

[string Substring method 80](#_Toc509154883)

[string Contains, EndsWith, and IndexOf methods 81](#_Toc509154884)

[Better ways of writing output to the Console 82](#_Toc509154885)

[Dealing with bad user input 84](#_Toc509154886)

[Check your understanding 85](#_Toc509154887)

[Chapter 8 - Arrays 87](#_Toc509154888)

[Parallel Arrays 91](#_Toc509154889)

[Using the Sort(), Reverse(), and BinarySearch() Methods 93](#_Toc509154890)

[Two Dimensional Arrays 94](#_Toc509154891)

[An Array of Arrays (Advanced Concept) 97](#_Toc509154892)

[Arrays of the char data type 98](#_Toc509154893)

[Check your understanding 99](#_Toc509154894)

[Chapter 9 – Adding Methods 104](#_Toc509154895)

[Why methods are good 105](#_Toc509154896)

[The mechanics of writing methods 106](#_Toc509154897)

[Implementation Hiding 112](#_Toc509154898)

[Method Overloading 112](#_Toc509154899)

[Parameter Passing By Value and By Reference 113](#_Toc509154900)

[Check your understanding 114](#_Toc509154901)

[Chapter 10 – Windows Forms 122](#_Toc509154902)

[The utility of Windows Forms 122](#_Toc509154903)

[Quick Start, Our First Forms Program 122](#_Toc509154904)

[Pizza Parallel Arrays, Our Second Forms Program 131](#_Toc509154905)

[Forms Summary 136](#_Toc509154906)

[Check your understanding 137](#_Toc509154907)

# Chapter 1 – Programming with C#

### Types of software

It is useful to divide computer software, or programs, into two broad categories.

1. System Software, which generally is designed to hide complexity and differences in the computer hardware and to provide nicer, more powerful abstractions for application writers. For example, it takes about 39 assembly language instructions to ask a user for two numbers, add them, and write out the answer. Assembly language is the low lever code that PC computers actually execute. An example of System Software is a C# compiler. A compiler lets us write much more powerful single instructions and then it converts this more English looking code into the many lines of complex assembly code. The same program in C# requires about 5 lines of code. Examples of System Software also include operating systems, database engines, and networking software.
2. Application software, which is generally software that a person wants to use to solve a real-world problem. This includes office programs like word processors and spreadsheets, business applications, and of course, games. Application software is easier to write because it takes advantage of System Software. For example, a common thing done in an application program is to save some information to a file on a disk. An application programmer would generally take advantage of libraries of code that come with the C# language and the Windows operating system. In this manner, an application coder can save information to a file by writing about 10 lines of code. This code will in turn make calls to, and use, many thousands of lines of system software code.

Application software engineers use System Software to create useful applications and games. The people who write system software are generally Computer Science majors.

Programming is defined by Rob Miles as deriving and expressing a solution to a given problem in a form which a computer system can understand and execute.

**Two things fall out of this definition:**

* **You need to understand how to solve the problem yourself before you can write a program to do it.**
* **The computer has to be made to understand what you are trying to tell it to do**.

### The C# language

The focus of this book is writing application software using the C# language. There are a great many programming languages around, and during your career you will have to learn more than just one. C# is an excellent, modern language to start programming in, but do not think that it is the only language you will ever learn.

C# is a very flexible and powerful programming language with an interesting history. It was developed by Microsoft Corporation for a variety of reasons, some technical, some political and others marketing. C# bears a strong resemblance to the C++ and Java programming languages, having borrowed (or improved) features provided by these languages. The origins of both Java and C++ can be traced back to a language called C, which is a highly dangerous and entertaining language which was invented in the early 1970s. C is famous as the language the UNIX operating system was written in, as it was specially designed for this.

I referred to C as a dangerous language. So, what do I mean by that? In programming terms, what this means is that C lacks some safety features provided by other programming languages. This makes the language much more flexible. However, if you do something stupid C will not stop you, so you have a much greater chance of crashing the computer with a C program than you do with a more modern, safer language.

The C# language attempts to get the best of both worlds in this respect. A C# program can contain **managed** or **unmanaged** parts. The managed code is fussed over by the system which runs it. This makes sure that it is hard (but probably not impossible) to crash your computer running managed code. However, all this fussing comes at a price, causing your programs to run just a bit more slowly. To get the maximum possible performance, and enable direct access to parts of the underlying computer system, you can mark your programs as unmanaged. An unmanaged program goes faster, but if it crashes it is capable of taking the computer with it. C# is a great language to start learning with as the managed parts will make it easier for you to understand what has happened when your programs go wrong. In this course, we will stick with managed code.

The C# language is **Object Oriented**. Objects are an organizational mechanism which let you break your program down into sensible chunks, each of which is in charge of part of the overall system. Object Oriented Design makes large projects much easier to design, test and extend. It also lets you create programs which can have a high degree of reliability and stability. While object oriented programming is very important, we won’t get deep into that aspect of C# for this course as there are some fundamental programming concepts which need to be learned before we make more use of object structure in our programs. (At BC, this is done in the course PROG 120)

C# is a compiled programming language. The computer cannot understand the language directly, so a program called a compiler converts the C# text you write into the low-level instructions which match the hardware level instructions that the CPU chip actually executes. For our purposes, this just means we need to compile our program before we try and execute it. To write and run our programs, we will be using the **Visual Studio** IDE (Integrated Development Editor). This is a sophisticated editor that understands the C# language and provides considerable help as we write and debug our code. It is a bit like how the Microsoft WORD program helps you by finding spelling and grammar errors. Visual Studio (henceforth generally just called **VS**) provides considerably more help for us then does WORD. One such example is, within the VS IDE, when you select “run your program”, it will automatically first call the compiler for you.

A compiler is a large program which knows how to decide if your program is legal. The first thing it does is check for errors in the way that you have used the language itself. Only if no errors are found by the compiler will it produce any output. The compiler will also flag warnings which occur when it notices that you have done something which is not technically illegal, but may indicate that you have made a mistake somewhere. An example of a warning situation is where you create something but don't use it for anything. The compiler would tell you about this, in case you had forgotten to add a bit of your program. The C# language is supplied with a considerable library of existing C# code which lets our programs do things like read text from the keyboard, write on the screen, set up network connections and the like. These extra features are available to your C# program but you must explicitly ask for them. They are then located automatically when your program runs. Later on, we will look at how you can break a program of your own down into a number of different chunks.

### Ready, Shoot, Aim

While it is possible to start VS and just start writing code, this is generally a bad idea. Before you attempt to solve a problem using a computer program, it is better if you first make sure you clearly understand everything you want the program to do, as well as clearly thinking though the things it will not do. These are called the **requirements**. Getting the requirements clearly documented and agreed to by the programmers and whoever is paying for the software is absolutely critical to the success of any project.

Next you must think through, probably using good old pencil and paper, how you are going to structure the program. If you sit down with a pencil and work out the general structure, flow, and logic first, you will probably get to a working system in around half the time.

### What Comprises a C# Program?

If your friend wanted to tell you how to make their favorite fruitcake they would write the recipe down on a piece of paper. The recipe would be a list of ingredients followed by a sequence of actions to perform on them. A program can be regarded as a recipe, but written for a computer to follow, not a cook. The ingredients will be values (called variables) that you want your program to work with, including any input that the program might ask the user to supply. The program itself will be a sequence of actions (called statements) that are to be executed by the computer. Rather than writing the program down on a piece of paper you instead put it into a file on the computer, often called a **source file**. (Just like WORD saves your document in a filename.docx file.) This is what the compiler reads before outputting another file, an executable file. A source file contains three things:

* Instructions to the compiler
* Information about the structures which will hold the data to be stored and manipulated.
* Instructions which manipulate that data

The C# compiler needs to know certain things about your program. It needs to know which external resources your program is going to use. It also can be told about any options for the construction of your program which are important. Some parts of your program will provide this information to tell the compiler what choices you have made.

Programs work by processing data. The data has to be stored within the computer while the program processes it. All computer languages support **variables** of one form or another. A variable is simply a named location in memory where a value is held while the program runs. C# also lets you build up structures which can hold more than one item, for example a single structure could hold all the information about a particular bank customer. As part of the program design process you will need to decide what items of data need to be stored. You must also decide on sensible names that you will use to identify these items.

The actual instructions which describe your solution to the problem must also be part of your program. A single, simple, instruction to do something in a C# program is called a statement. A statement is an instruction to perform one particular operation, for example add two numbers together and save the result. A powerful aspect of programs is that some statements can change which statement is performed next, so that your program can look at things and decide what to do. This is often called branching, as you might climb a tree and step by step as you ascend, you make decisions if and which branch to go out on.

With C#, you put a logical collection of statements together to form a chunk of a program which does one particular task. Such a chunk is called a **method**. A method is very similar to what other programming languages call a function. A method can be very small, or very large. It can have any name you like but it is best if you try to make the name of the method fit what it does, for example ShowMenu or SaveToFile. You can break your program into as many chunks (methods) as you see fit. One method may “call” others to have them do a sub-task. Methods that are called by another method may or may not return a value. For example, one method might call and pass to another method a set of temperatures, and ask that 2nd method to return the highest temperature found in the set. The C# language also has a great number of libraries with many pre-written methods which you can use. These save you from "re-inventing the wheel" each time you write a program.

### Class, Object, Method, Property

I’m afraid we have a bit of the famous chicken-and-egg problem. We really can’t start writing programs until you understand something about these four critical C# structures. On the other hand, it’s pretty hard to explain what they are until you have learned to write at least some simple code. So what I am going to do is give just a very basic definition of these four items, leaving out many aspects and powerful concepts for now. Then as we go through the chapters, I will keep referring to these four structures and eventually, they will make pretty good sense. However it is extremely important that anytime I use any of these four words in this and the following chapters, if the word does not make sense to you in that context, you ***must*** come back to this sections and re-read this section. If you skip over my usage of these words and don’t understand them, you will just be more and more confused as the book progresses.

These four items are used for two important reasons. First, to organize your programs into logical sections and second, to make it convenient to model real world things that your program wants to deal with. In the case of a class, sometimes a class is defined to accomplish only one or the other of these two goals, and sometimes a class will help achieve both of these goals.

#### **Class**

A class is a container that holds related code. You could a write very large program and put all the code into one single class. In fact, this entire book, we will in fact put all our code in one class because we will have very small programs. But as things get bigger, it makes much more sense to break the code into more than one class. My kitchen has a series of many drawers, and I keep related things in particular drawers. *I could* instead have one giant drawer and throw everything in there, but that would sure make it hard to find things.

For now, we will talk about only two things that we will use in our basic classes; **methods and properties**.

#### **Methods**

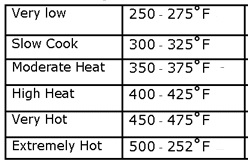
As mentioned above, all our executable code statements will be in containers called methods. A class might have none, one, or many methods. Each method is a set of instructions that achieve one logical function. That is what a well written method does. It is possible to have one method that does several quite different functions; the compiler will not say this is illegal C# code, but hopefully your teacher will! Good coding practice dictates that one method should be for one function. Add a second or more methods if you want the class to hold instructions for more than one function.

It might be useful to think of a cook book as a simile for a class. Let’s further clarify that this class is a cook book for desserts. The class (the book) is one place to hold a collection of “instructions for making particular desserts”. Maybe our desserts cook book has 20 recipes for 20 different desserts. Each recipe will correspond to what we call a method; a set of instructions to do one thing. And just like with a method in a class, you COULD write a single recipe that describes how to make both an apple pie and how to make chocolate fudge, but that is not good practice. It would be much better to split those instructions into TWO recipes, one for the pie and one for the fudge.

Think of methods as verbs, they are things your code does.

#### **Properties**

Properties are **values** (information) defined in the class that you desire to share with code in ***other*** classes. They are very similar to what we define as variables, the difference is, variables hold values to be used exclusively within the particular class where they are defined. Properties are sort of special variables which you code to expose these values to code outside of the class in which they are defined. If our desserts cook book is our model of a class, think of one of the first pages in the cookbook which might contain this information:



Our cook book is letting any reader know the value of six properties, one named “Very Low”, one “Slow Cook”, etc.

Properties generally look more like nouns or values, and not verbs. You read or write them.

#### Using statements

Using statements are something we will have to use right away, and it also allows me to show some examples of a class and methods. Using statements are a way of letting our code make use of code that other people wrote and placed in a library. C# comes with a very large library of code for us to use, so that we don’t have to write everything ourselves. This library is generally referred to as the .NET library. Each of these many libraries are collections of related classes, and each one contains one or more useful methods. One library that we will always use is called System. So our first line of code in all our programs will always be:

using System;

By adding that line to our programs, we have access to all the classes (and their methods and properties) defined in this System library. In the System library is a class called Console with some useful methods and properties. We will be using this class in our first program and every one thereafter. Console is a class that has been written as a software model of the hardware computer screen and keyboard. We write code using the Console methods and that code will write our messages out to the computer screen and also read back what the user types for our program.

So we will write code such as:

Console.WriteLine("Welcome to my program!");

Console.ForegroundColor = ConsoleColor.Blue;

Console.WriteLine("Please type your name.");

string userName = Console.ReadLine();

With those four code statements, twice we “called” the WriteLine method of the class Console and asked it to write a message to the screen. Then the 4th line used the ReadLine method of the class Console to ask the method to take whatever message the user types and save it for us in a variable named userName. (More on that later.) The 2nd line is setting a value (ConsoleColor.Blue ) of our Console class’s property named ForegroundColor. Our first line of text will be white text on a black screen, but because we changed the value of the Console class’s property, the next message will be written with blue text.

#### **Objects**

Imagine you wrote your own Desserts-Cook-Book with your own paper and pen. You decide you like it so much, that you write a second cook book, an Appetizer-Cook-Book, again, using your own paper and pen. You have now defined two classes, each containing several methods (recipes) and sets of values called properties. In C# you reference a method or property by stating the name of the class, then a dot, and then the name of the method or property. (See the four lines of code just above.) So you now have two classes (books) and if that’s all you want to do, you are done.

But what if you decide you want to make copies of your Desserts-Cook-Book and give them to your friends. And imagine your friends love the cook book, but because of personal preferences, they want to make some changes. Maybe they decide that the fudge should be cooked for 35 minutes instead of the 30 minutes you have in your original cook book (class). And maybe another friend likes there fudge cooked for only 25 minutes. So what you have now is a series of exactly the same Desserts-Cook-Books, but with different values stored in their properties. In C#, we call these copies of a class “objects”. And each object gets a unique name. So now there is Joe’s-Desserts-Cook-Book, and Sue’s-Desserts-Cook-Book, and Chen’s-Desserts-Cook-Book. You can no longer refer to Desserts-Cook-Book.Chocolate-Fudge as that is ambiguous, since there are now many different Desserts-Cook-Books. So now when you want to reference an object’s method (a cook book’s recipe) you must specify the name of the object that was made from the original (class). Chen’s-Desserts-Cook-Book. Chocolate-Fudge

Again, if you write a class with some methods and only want one copy, then you reference the methods and properties of that class using its class name. You actually have to put a label on your class, the keyword static, which says there can only be one of these things (this class). But if you write a class with the idea that you will make many copies, you do not mark it static, and the copies of the class are called objects, and each object gets its own unique name. You reference the methods and properties of these objects using the object’s name.

For the first few chapters, all our code will be written inside of one particular method, the Main() method. When you tell VS to run your program, the system starts your program by looking for a particular method with a special name, **Main**. This method is called automatically when your program starts running, and when Main finishes, your program ends. When we write our first few programs, all our code will be inserted into this Main chunk (method). After we learn the fundamentals of programming we will begin to add more methods and then in later courses, more classes.

*That section is pretty dry and boring, but it is critical that you become very familiar with these terms and what they mean. Please refer back to this section when you encounter these terms and they don’t seem to make sense.*

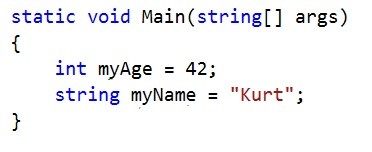
### Names, Identifiers, Key Words, and Strings

Everything, such as classes, methods and variables, gets a name. The names that you invent to identify things are called **identifiers**. When you want to save a number or a string of text, you save them in a **variable**. Variables must have names so we can get back the values that we saved in them. For example, woodLength might be a good name choice for a variable where we want to hold the length of wood required. Later on, we will look at the rules and conventions which you must observe when you create identifiers. There are also some key words that the C# language reserves for its own use. In a recipe, a keyword would be something like "mix" or "heat" or "until". They would let you say things like "*heat* sugar *until* molten" or "*mix* *until* smooth". In fact, you'll find that programs look a lot like recipes. Keywords in this text and in VS will be colored blue. It is a bad idea to give names to the things you create such as variables or methods that are also used by C# as keywords. Class is a good example. You might be writing a program about school courses, and you might want to refer to a courses with a variable named class. That is not a good idea, as it will be confused with the C# key word class. Instead make up a similar name, theClass or aClass would be fine.

There are two kinds of text you will use in your program. There are the **instructions** that you want the computer to read and perform and there are the **messages** that you want the program to actually display to the user. You might add the following instruction to your cake recipe: **Now write the words “Happy Birthday” on top of the cake in pink icing**. We are using double quote characters to mark the text that is to be drawn on the cake. The recipe is not telling *you* to have a merry Christmas, it is giving you a sequence of letters that it wants you do use in the construction of the cake. C# works in exactly the same way. “Merry Christmas” is not part of the instructions; it is what needs to be written to the computer screen. Text used this way in C# programs is colored red in this book and in VS and is referred to as a string. A string is a collection of letters, and each one of the letters is called a character.

Please look at the VS screen shot image below where I demonstrate some key words and defining some variables.

* The words static, void, string, and int are all reserved key words. You cannot use those words for your identifiers (names).
* The 2 code statements are contained in our one method named Main, which is delimited with an opening **{** and closing **}** brace. The statement int myAge = 42; will be the first line of code executed when this program is started
* I created two variables, and after declaring what kind of data they may store (an integer and a text string), I assigned them initial values.



## Check your understanding

**Questions 1:** Are we learning to write application software or system software? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Questions 2:** Before starting to write code, you must fully understand what the code should do, document it, and get the customer to agree with your document. This type of document is called a \_\_\_\_\_\_\_\_\_\_\_\_ document.

**Questions 3:**  A named location in memory where a value is held while the program runs is called a \_\_\_\_\_\_\_\_\_\_\_.

**Questions 4:** We covered two structures that hold chunks of code. They are called a \_\_\_\_\_\_\_\_\_\_\_ and a \_\_\_\_\_\_\_\_\_\_\_\_\_. The latter one is a subdivision of the former, so it is always found inside of the former.

**Questions 5:** If we want to use methods that live in .NET classes, we have to add what kind of statement at the top of the program? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Questions 6:**  What is the difference between a class and an object?

**Questions 7:**  Name 2 reserved C# words that we cannot use as names for our variables. \_\_\_\_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

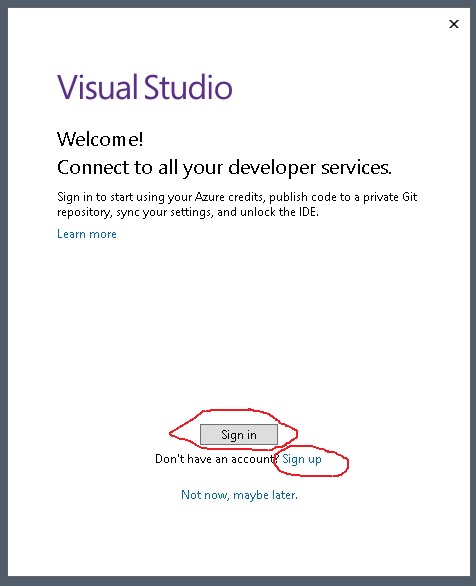
# Chapter 2 – Integrated Development Environment

As mentioned in Chapter 1, this book assumes you will be using the **Visual Studio** IDE (Integrated Development Environment). This is a sophisticated editor, henceforth just called **VS**, that understands the C# language and provides considerable help as we write and debug our code. As we learn the C# language, we will write code to demonstrate how it works in VS. This chapter is focused on getting you started with VS. It is best that you open VS when you study from this book, and cut and paste and then try the code statements found in this book.

Microsoft offers several versions of VS, most of which cost money, but the version we will use is free. Even though it is free it is still a highly functional version. Search the web for “visual studio 2017 community download”. This site link below is the correct one at the time of this writing:

<https://www.visualstudio.com/vs/community/>

This version is the community edition, which means the sources are available so that people can extend it to support many other features and frameworks. The 2017 version made a significant change from prior versions in which parts of VS are installed and how you add needed other components.



After you install it and start it up, VS wants you to log in with a Microsoft account. This is not required, and you might get through the course without doing that. However I highly recommend it, and I require it in *my* class. This is NOT and must not be your Bellevue College email account. It is an account that Microsoft will use to recognize you if you use any of their online products, some of which we will use in later courses. Examples include:

* Logging into the Microsoft Azure Cloud
* Logging into Microsoft’s Visual Studio Team Services
* Collecting points when you search with Bing (which are actually useful if you like free coffee!)
* Others

If you already have a Microsoft account, such as a Hotmail or Outlook.com email account, you can use that. If you don’t have one, click on the “Sign Up” and either create a new Microsoft account or you can use an existing other email such as gmail or yahoo mail. However I think that leads to confusion about account names and passwords, so I suggest you just create new Microsoft account. It won’t hurt!

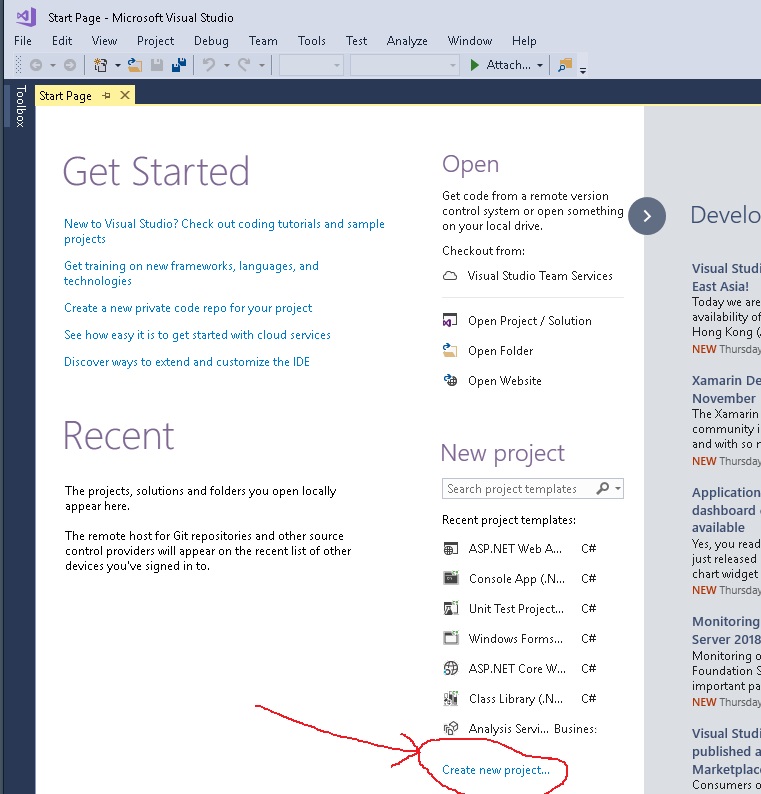
Once you are done with that, you should see this screen. We will always either open an existing project, which will show up under the **Recent** area, or we will create a new project.

Unless you change it, all your projects will end up in a folder:

C:\Users\xxxxxxxx\source\repos

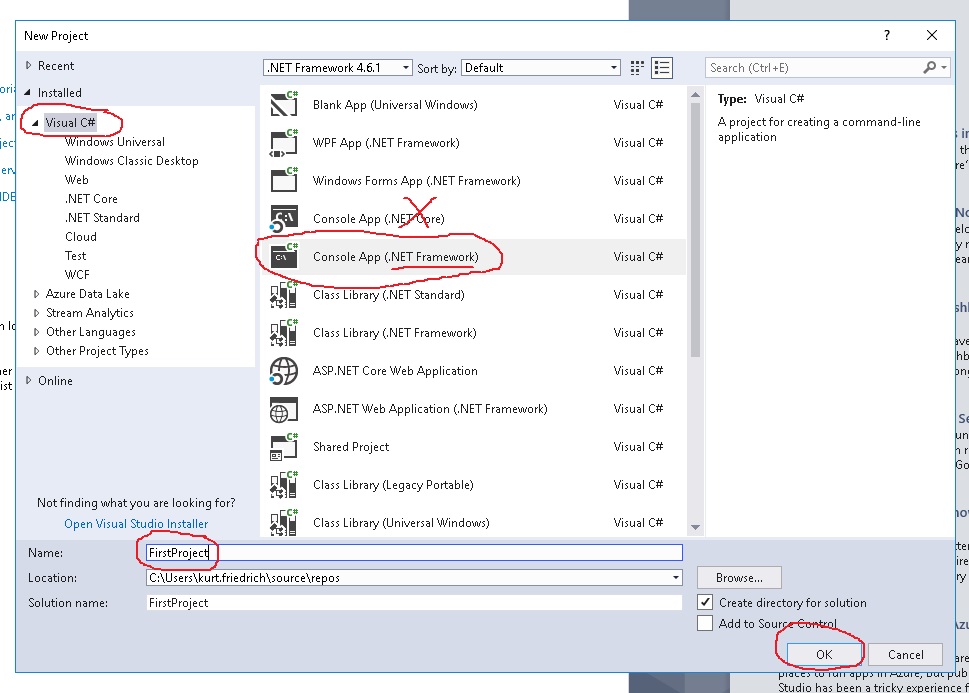
Where xxxxxxxx is your username on that computer.

Now click the Create a new project, as shown in the next image.

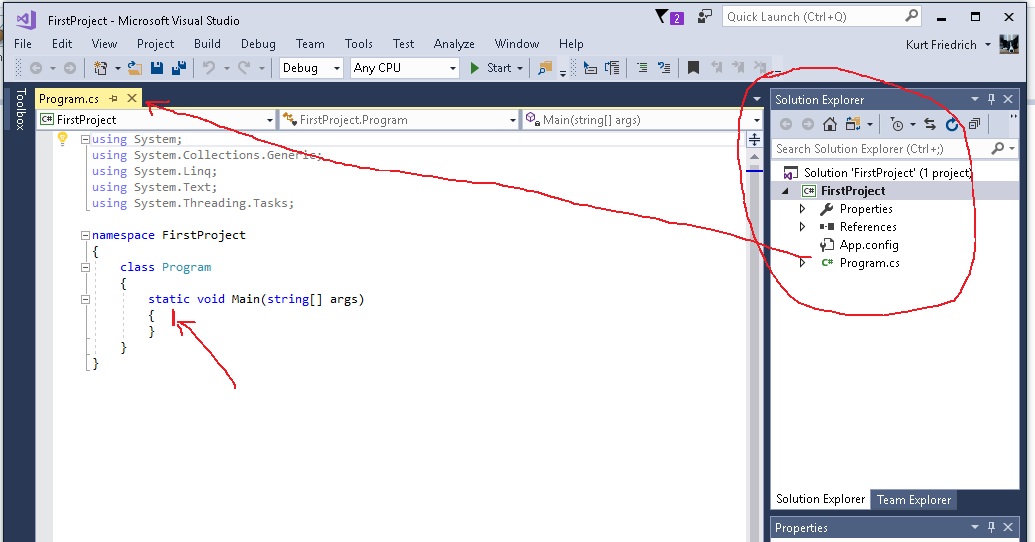


In the next screen, VS is offering to start one of many different kinds of projects. For this course, ***always*** choose the left-most “**Visual C**#” choice, and then in the middle column, select **Console App (.NET Framework)** choice. Do not choose Console App (.NET Core). At the bottom of the form you enter a name for your Project, which will also be the name for your Solution. (In general, a Solution can have more than one Project, but in this course, we will only have one.)

After you enter that name, click the ok button.

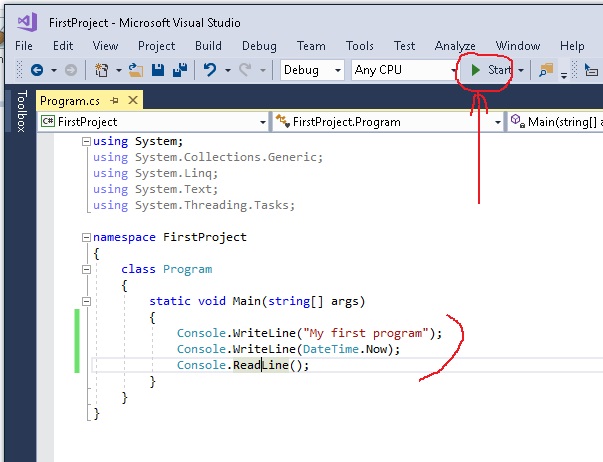


Now you will see the normal “Project Screen”. You will be spending much time in this view. The large left window is an editor window, where you edit which ever file you have selected from the right window, which is the Solution Explorer window. Your C# code will always be entered and saved into a file within the solution folder called Program.cs.



What you see in the above picture, is a complete C# program that VS created for you. It starts with several lines that begin with using xxxx. These are libraries of code that the system is assuming you might want to make use of. Lower down, inside of the class Program, is a method definition called **Main**. We can ignore the words static void, and (string[] args) for now. The important thing is there is a method named **Main**. When you want to run your project, the system will always look for the Main method inside of a class Program as the correct location to start running your program. As VS coded this for you, there are no instructions inside of **Main**. The program is runnable, it just won’t do anything. So let’s add some real C# code to try it.

Type in the 3 lines of code as shown below.



Console.WriteLine (*some string*) is using a library method named WriteLine in another class named Console. This method will write out to a “console window” whatever string is inside of the parentheses. So the first line will cause the words “My first program” to be written to this console window. The next line uses another library method, Now, in the class called DateTime to get the current time, and then the WriteLine method writes that value out to the console. The third line will wait until the user pushes the Enter button on the key board before proceeding. After that, the program is over and the console window will close. After you type in those 3 lines of code and click on the green arrow “start”, the console window will open as shown below. Of course your date and time will be different.



After the program executes the first 2 lines of code, the information appears in the window. The program then did the 3rd line, and the program is still running. It is waiting for you to type the Enter key, when you do, the window will go away as the program is done.

Congratulations, you have just written and run your first C# program!

Now go back and read the section on [Class, Object, Method, Property](#_Class,_Object,_Method,) in chapter 1. It will make a little bit more sense now that you see some of those terms in use.

### Note for the entire book about cutting and pasting code

WORD, PowerPoint, and Visual Studio all like to autocorrect our text. Generally they do a good job, but when you move back and forth between them, they don’t always agree. One of the biggest problems is the quote characters. It turns out there is a left quote “ and a right quote ” and there is a neutral quote "

Visual Studio (C#) wants ONLY the neutral quote ", but when I cut from Visual Studio and paste code examples into WORD, WORD loves to convert my quote character into either the right or left version. For all the sample code in this book, I have tried to un-do the corrections that WORD has done to my quotes, but should you ever cut and paste code from this book into Visual Studio, and it does not work, the first thing I suggest you look at is any quotes in the code. Make sure they are all the neutral quote, and not the left or right. WORD also likes to insert a space character that it thinks makes better English, but breaks C#, so watch for those also!

## Check your understanding

**Questions 1:**Of the many project types that Visual Studio supports, what is the one we will always use in this course? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Questions 2:**In Visual Studio, the window that shows all the files in your solution is named \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_

**Questions 3:** What is the shape and color of the icon you click to have Visual Studio compile and then run your program? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Questions 4:** The method WriteLine is contained in a class named \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Chapter 3 – Our First Useful Program

*Note: occasionally when I use an example program to explain a new C# construct, it is much more convenient to use some structure or part of the C# language that we haven’t covered yet. Rather than make the example programs extremely awkward, I will sometimes use something from a later chapter in an earlier chapter. Whenever I do that, I offer a brief explanation of this new concept and point out that we will cover it fully in a later chapter. When this happens, I suggest you just accept on blind faith that my code is correct, and try and understand the effect of the code item I am using, but don’t worry if you don’t fully understand it*.

Consider the scenario; you are sitting in your favorite chair contemplating the universe when you are interrupted in your reverie by a friend of yours who sells double glazed windows for a living. He knows you are a programmer of sorts and would like your help in solving a problem which he has. He has just started making his own window units and is looking for a program which will do the costing of the materials for him. He wants to just enter the dimensions of the window in meters and then get a print out of the cost to make the window, in terms of the amount of wood in feet and glass in square feet required.

The first thing you need to do is find out exactly what the customer wants you to do. These are called requirements.

***Specifying the Problem***

When considering how to write the specification of a system there are three important things:

* What information flows into the system?
* What flows out of the system?
* What the system does with the information.

There are lots of ways of representing this information in the form of diagrams, for now we will stick with written text when specifying each of the stages:

***Information going in***

In the case of our window problem we can describe the information going in as:

* The width of a window in meters.
* The height of the window in meters.

***Information coming out***

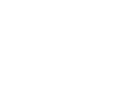
The information that our customer wants to see is:

* The area of glass required for the window in square feet
* The length of wood in linear feet required to build a frame.

You can see what we need if you take a look at the diagram below:



Width of Window



Height of

Window

The area of the glass is the width multiplied by the height. To make the frame we will need two pieces of wood the width of the window, and two pieces of wood the height of the window.

### Putting in more detail

We now have a fairly good understanding of what our program is going to do for us. Being sensible and far-thinking people we do not stop here, we now have to worry about how our program will decide when the information coming in is actually valid.

This must be done in conjunction with the customer, he or she must understand that if information is given which fits within the range specified, your program will regard the data as valid and act accordingly.

In the case of the above we could therefore expand the definition of data coming in as:

* The width of the window, in meters and being a value between 0.5 meters and 3.5 meters inclusive.
* The height of the window, in meters and being a value between 0.5 meters and 2.0 meters inclusive.

Note that we have also added units to our description, this is very important. Having written this all up in a form that both you and the customer can understand, we must then both sign the completed specification, and work can commence.

### What the program needs to actually do.

Remember, you have to know how you are solving the problem before you write code. The program can derive the two values according to the following equations:

glass area = width of window \* height of window

wood length = (width of window + height of window) \* 2

When we start with our window program we now know that we have to:

read in the width

verify the value

read in the height

verify the value

calculate ( width + height ) \* 2 \* 3.25 and print it

calculate (width \* 3.25) \* (height \* 3.25);and print it

### Real code for our window problems

While it’s too soon to fully understand our program, here is what the code would look like. This is followed with an explanation of many of the syntax items (language rules) that are required. The lines in green font are called comments. The compiler ignores them, they are just there to help anyone trying to understand how the program works.

using System;

class Program

{

static void Main()

{

/\*

\* This program asks the user for the size of a window.

\* It then calculates the total length of wood needed for the frame

\* and the total area of the glass.

\* \*/

double width;

double height;

double woodLength;

double glassArea;

string widthString;

string heightString;

// read in the width

Console.WriteLine("Please enter the width.");

widthString = Console.ReadLine(); // get the value from the user

width = double.Parse(widthString);

// need to add code later to verify the value

// read in the height

Console.WriteLine("Please enter the height.");

heightString = Console.ReadLine();

height = double.Parse(heightString);

// need to add code later to verify the value

// calculate ( width + height ) \* 2 \* 3.25

**woodLength = 2 \* (width + height) \* 3.25;**

// calculate width times height

**glassArea = (width \* 3.25) \* (height \* 3.25);**

// and print it

Console.WriteLine("The length of the wood is " + woodLength + " feet");

Console.WriteLine("The area of the glass is " + glassArea + " square feet");

// this line prevents display window from closing until you type the Enter key

Console.ReadLine();

}

}

This is a valid program. If you enter this code into VS, you can run it. The actual work is done by the two lines that I have bolded. Broadly speaking the code before these two lines is concerned with setting things up and getting the values in to be processed. The code after the two lines is concerned with displaying the answer to the user.

We can now go through each line in turn and try to see how it fits into our program. It might be good to go back and review this section now: [Class, Object, Method, Property](#_Class,_Object,_Method,)

#### **using System;**

This is an instruction to the C# compiler to tell it that we want to use code from a library named System. We don’t have to write every line of code ourselves, we can take advantage of many methods that are defined in many code libraries in the .NET library. Visual Studio has these libraries available for us. In this case, System library includes the code to allow us to write to, and read from the display window, which is called the console. We use several methods in this class called the Console class. The Console class provides a software model of the display and keyboard. We don’t have to know the details of how to get information to appear on the screen or read back from the keyboard. All we have to know is how to call two methods.

#### **Class Program**

A C# program is made up of one or more *classes*. A class is a container which holds data and program code to do a particular job. In the case of our double glazing window calculator the class just contains a single method which will work out our wood lengths and glass area, but a class can contain much more than that if it needs to.

You need to make up an identifier for every class that you create. I've called ours **Program** since this is the default that VS starts you out with when you create a new program. For all the chapters in this book, we won’t create any other classes, but we will use methods in classes that are already defined in the .NET library.

There is a convention that the name of the disk file that saves our code which contains a particular class should match the class itself, in other words the program above should be held in a file called **Program.cs**, which is what in fact, VS does for us by default.

#### **Static**

This keyword generally means “there can be only one of these”. As we discussed back in Chapter 1, some classes allow you to make many copies, called objects. If you want there to be only one copy, you mark a class static. Likewise, if there can be only one method with a particular name, you mark the method with the static keyword. For now, just know that the system expects there to be only one place where your program can start, and that is a method named Main, and we must mark it static to tell the system, this is the one and only place to start executing my code.

#### **Void**

A void is nothing. In programming terms the void keyword means that the method we are about to describe does not return anything to any code that calls this method. This method will execute its code and then finish. In some cases we write methods which return a result (in fact, we will use such a method later in this program).

However, in this case for our Main method, we are explicitly stating that it does not return anything. When you write methods, you must either state exactly the type of data a method returns, or you must use the keyword void, to say nothing. The C# language likes things to be very clear, no ambiguity.

#### **Main**

You choose the names of your methods to reflect what they are going to do for you. Except for Main. As I said above, this Main method (and there must be one, and only one such method) is where your program starts running. When your program is loaded and run the first method given control is the one called Main. If you don’t have a Main method the system quite literally does not know where to start

#### **( )**

Our method name Main ends with a pair of brackets enclosing nothing. This tells the compiler that the method Main can do its job without be handed any extra information. Often you will want to give a method some particular pieces of data to work on. These pieces of data are said to be “passed into” the method as arguments. If you had a method, for example static float sin(int angle); it is indicating that it needs to be passed in a particular value of some angle, and that value must be an integer. It would then calculate the trigonometric function sine on that angle, and then return that answer as a float. float indicates another type of variable, a floating point number, as often used in scientific calculations. We will cover methods in more detail later in this book.

#### **{ }**

This is a set of braces. As the name implies, braces come in packs of two, i.e. for every open brace there ***must*** be a matching close. Braces allow programmers to lump pieces of program together and put boarders around the code. Such a lump of program is generally called a block. The particular block we are looking at now is a block of code living inside the Main method. So the closing brace is the end of the method Main. A method’s block can contain the declaration of variables used within it, followed by a sequence of program executable statements which are executed in order. When the compiler sees the matching close brace at the end it knows that it has reached the end of the method.

#### **double**

The double keyword says that the variable it describes is a “*double precision floating point number"*. We will look more into variable “*types*” later, the system provides many different ones, each one particularly suited to hold a specific *type* of data. The use of the word *type* in computer languages is quite specific and an important concept. You will develop a good feel for this concept as we go thought the first few chapters.

Our program needs to remember certain values as it runs. Notably it will read in values for the width and height of the windows and then calculate and print values for the glass area and wood length. C# calls the places where values are saved in computer memory *variables*. It is useful to think of variable names as pointers to a particular addressable location in the computer’s memory where it holds the information. When it goes to that location to read or write that value, it must understand how to interpret the value. Is it a date? An integer, such as the number of students in a class? Is it a string, such as a person’s name? For that reason, each time you creates a new variable (a place to save data), it must have a unique name, **and** you must specify what *type* of data it will store. It is generally a good practice to define any new variables defined in a method at the beginning of the method. We did this in our example with the variables width, height, woodLength, glassArea, widthString, and heightString.

Notice carefully that **every** line of code ends with either a semicolon, or else it is followed with no semicolon but instead has a code block marked with a set of matching braces.

int size = 4**;** // declare an int varaible

int power = 3**;** // declare an int varaible

int answer = (size + 2) \* (power - 1)**;** // executable statement

if (answer < 14) // no semicolon, instead, a code block marked with { }

**{**

// execute more code here if answer is less than 14

**}** // this closing brace is the end of the if statement

### Comments (and proper code formatting)

When you are paid to write programs, it is generally the case that the expectation is the program will be used for months, even years. Normally, as time goes by, bugs will be reported and need to be fixed, as well as requests for more features. You should write all your code assuming that someone else, at a later date when you may not be anywhere to be found, will need to make changes to your code. Therefor professional programmers will consistently write programs that:

* Should be easy to read. At no point should the hapless reader be forced to backtrack or brush up on knowledge that the writer assumes is there. All the names (identifiers) should impart clear meaning and be distinct from each other (more on this next).
* The various components should be organized in a clear and consistent way and look good in the VS editor. A good program is well laid out. The different blocks should be indented and the statements spread over the page in a well formed manner. Note, if you “cut” all your code and then “paste” it back in, VS will do a perfect job of indenting to make your code clear.
* It should be clear who wrote it, and when it was last changed. If you write something good you should put your name on it. If you change what you wrote you should add information about the changes that you made and why. This can be done as a set of comments at the top of the program.

A big part of a well written program is the comments that the programmer puts there. Comments are stored in the file with your code, but the compiler ignores them. They are there strictly to make notes so that when other people read your code, they can make better sense of it, and more quickly. You do not need to add comments on *every* line of code as C# statements are fairly easy to understand one at a time.

goatCount = goatCount + 1; // add one to goatCount <<< this comment is not needed!

However each section of logic should have comments that explain what you are doing and how you are doing it. If you do anything “tricky”, that should definitely be commented. There are 2 ways to add comments. If you are going to add multiple lines, such as at the beginning of the code to explain it, you start with a /\* and then VS will add a \* to each new line, and they will all be considered comments, until you end the block of comments with a closing \*/

Here is the example from the above code, notice how VS makes all comments a green font:

/\*

\* This program asks the user for the size of a window.

\* It then calculates the total length of wood needed for the frame

\* and the total area of the glass.

\*/

You can also add one line of comment, either on a line by itself, or as an ending to a line of code, by starting the comment with // here are examples:

// read in the width

Console.WriteLine("Please enter the width.");

widthString = Console.ReadLine(); // get the value from the user

We covered a lot of details in this chapter so far to give you a quick, big picture view of what a C# program looks like. We will be going back through all these concepts in more detail in the following chapters.

### Debugging

Debugging is the process of removing all syntax and logical errors from the program. Syntax errors are errors where you entered code that the compiler does not understand, or does not allow. Here are two examples of something the compiler does not allow:

int myAge = "forty";

dinner = "spaghetti";

The first line is not allowed because you declared the variable myAge to be an int, which means it can only hold integer numbers such as 5, 42, or 231235. But then you tried to set myAge to be a string value. The second line is a line the compiler does not understand, because you never told the compiler what dinner is. You must declare it and say what kind of data it is allowed to hold before you try and assign it a value. Visual Studio will detect most of these kinds of errors and highlight them for you.

Logic errors are when the compiler accepts your program and does exactly what the code says it should do, but the code expresses flawed thinking. Here is an example using variables that hold floating point numbers (numbers which can have decimal points).

float a = 3.14F;

float b = 6.2F;

float c = 14.89F;

float average = (a + b + c) / 2;

The compiler is happy to allow you to add the 3 numbers and divide by 2. But that is not the correct way of calculating the average of 3 numbers.

The normal process for removing bugs from a program is to first, fix all the syntax errors Visual Studio finds. Then run the program to make sure it runs with no problems. Lastly, test the program by giving it a wide range of inputs and verifying it gets the correct results. Visual Studio supports an automated system for testing, which is covered in the PROG 120 course.

## Check your understanding

**Questions 1:** When you are specifying the problem to be solved, what are the 3 aspects of information that need to be clearly documented?

**Questions 1:** Why do we start our programs off with the line “using System “?

**Questions 2:** As we use the word void, what is a good synonym for it in our context?

**Questions 3:** Visual Studio will start every Console Program by starting to execute code in a method named?

**Questions 4:** What does the compiler do with your program comments in your C# source file?

**Questions 5:** What are the 3 stages of testing our programs?

**Questions 6:** Paste the program from this chapter into VS and get the program to work.

# Chapter 4 – Storing Data in Variables

### Variables and Data

In this section we are going to take a look at how we can write programs that manipulate data, how values can be stored, retrieved and used. This provides us with the ability to perform the data processing part of programs.

In the window program above we decided to hold the width and the height of the windows that we are working on in variables that we described as double. Before we can go much further, we need to consider just what this means, and what other types of data we can store in programs that we write.

Programs operate on data. A programming language must give you a way of storing the data you are processing. What the data actually means is something that you as a programmer decide.



A variable is a named location where you can store something. You can think of it as a box of a particular size with a name painted on the box. You chose the name to reflect what is going to be stored there (we used sensible names like woodLength in the above program). You also need to choose the type of the variable from the range of storage types which C# provides. Type is a very important concept in C#. Some computer languages force you to be very careful about data types and they are called “strongly-typed” languages. C# is one. Others do not and they are called weakly-typed languages. JavaScript is an example. In C#, you place a restriction on each box that stores something (each variable you define), stating that only items of a certain “type” may be stored in that box. Would you add more sugar into your flour canister? In C#, if you define a new variable and specify it will hold letters (string) and then try and store a number in this “box”, the compiler will complain.

string firstName;

firstName = 5280; // compiler will not allow this

The argument in favor of strongly-typed languages is they help protect you from creating bugs, such as polluting your flour with sugar. The argument for weakly-typed languages is they are more flexible, and you can do cool and crazy things with this freedom. (Maybe you want to hide that last piece of candy in the flour so no one else can find it!)

As I said earlier, it is good to think of a variable name as an address, or pointer, to the location in the computer’s memory where it will remember the value you assign to the variable. When you open the sugar canister, you know its sugar, even if it was not labeled. But the computer really can’t go to a memory location, read the bit pattern stored there (for example, 01100111001010011001110110011001 ) and make any sense out of it UNLESS you have told it how to interpret it. That is why when we declare a new variable, we not only give it an address (name), we also mark its **type** (string, int, float, etc).

You will see shortly, that if you want to store a small integer, or a big integer, or a really big integer, you have three choices; Int16, Int32 (which is the same as int), or Int64. We are able to tell the computer not only what kind of data we are storing, but what size box we need. Most computers now have memory that is made up of chunks (called words) of 64 bits. If you say that your variable is an Int16, that lets the computer store 4 such variables in one memory location.

Programs also contain **literal values**. A literal value is just a value (usually expressed as you would think of it) in your program which you use for some purpose. You are not specifying a code name for something, you are saying “literally, this is the value I want the code to use.” For each type of variable the C# language has a way in which literal values of that type are expressed. Here are two examples.

string firstName;

firstName = "Sam";

Sam is not a code for anything, it IS the name of the person I want the computer to save. Sam is a string literal.

int myAge = 39;

39 is not code to be interpreted, it is a real integer, it is “literally” my age (Oh, well, it is the age I wish!)

Now let’s dig into the many data types C# provides for us to save things in memory.

#### **Storing Numbers**

When considering numeric values there are two kinds of data:

* Nice chunky individual values, for example the number of sheep in a field, teeth on a cog, apples in a basket. These are referred to as integers.
* Nasty real world type things, for example the current temperature, the length of a piece of string, the speed of a car. These are referred to as reals.

In the first case we can hold the value exactly; you always have an exact number of these items, they are integral.

In the second case we can never express what we are looking at exactly. Even if you measure a piece of string to 100 decimal places it is still not going to give you its exact length - you could always get the value more accurately. These are real. A computer is digital, i.e. it operates entirely on patterns of bits which can be regarded as numbers. Because we know that it works in terms of 1.s and 0’s it has difficulty holding real values. To handle real values the computer actually stores them to a limited accuracy, which we hope is adequate, and in fact is most often able to store more precision than however we take the measurements.

This means that when we want to store something we have to tell the computer whether it is an integer or a real. We also need to consider the range of possible values that we need to hold so that we can choose the appropriate type to store the data. (The optimal box size, or, how much computer memory to use.)

#### **Storing integer values**

Integers are the easiest type of value for the computer to store. Each value will map onto a particular pattern of bits. The only issue is one of range. The bigger the value you want to be able to express, the larger the number of bits that you need to represent it. C# provides a variety of integer types, depending on the range of values you would like to store:

|  |  |  |
| --- | --- | --- |
| Name | # bits | Size it can hold |
| sbyte | 8 bits | -128 to 127 |
| byte | 8 bits | 0 to 255 |
| short | 16 bits | -32,768 to 32,767 |
| ushort | 16 bits | 0 to 65,535 |
| int | 32 bits | -2,147,483,648 to 2,147,483,647 |
| uint | 32 bits | 0 to 4,294,967,295 |
| long | 64 bits | -9,223,372,036,854,775,808 to  9,223,372,036,854,775,807 |
| ulong | 64 bits | 0 to 18,446,744,073,709,551,615 |
| char | 16 bits | 0 to 65,535 |

The standard integer type, int, can hold frighteningly large numbers in C#, in the range -2,147,483,648 to 2,147,483,647. If you want to hold even larger integers than this (although I've no idea why you'd want this) there is a long version. Notice that some of these use 1 bit to allow for negative numbers, and others do not.

An example of an integer variable would be something which kept track of the number of sheep in a field:

int numberOfSheep;

This creates a variable which could keep track of over two billion sheep! It also lets a program manipulate "negative sheep" which is probably not meaningful (unless you run a sheep bank of course and let people borrow them). Remember that the language itself is unaware of any such considerations. If you want to make sure that we never have more than 1,000 sheep and the number of sheep never goes negative you must specify this behavior yourself.

When you edit your program source using Visual Studio (or another code editor that supports syntax highlighting) you will find that the names of types that are built into the C# language (such as int and float) are displayed in blue, as shown above.

Something else which you should bear in mind is that a program will not always detect when you exceed the range of a variable. If I put the value 255 into a variable of type byte this is OK, because 255 is the biggest possible value of the type byte can hold. However, if I add one to the value in this variable the system may not detect this as an error. In fact this may cause the value to "wrap round" to 0. Which could cause a program big problems. In, general, especially now that memory is so cheap, you want to pick types that are safely going to hold the biggest number your program might encounter. Later, when you learn about Try Catch structures, you will learn how to detect these out of range incidents. (PROG 120)

#### **integer literal values**

An integer literal is expressed as a sequence of digits with no decimal Point:

23

This is the integer value 23. I could use it in a program as follows:

numberOfSheep = 23 ;

### Storing real values

"Real" is a generic term for numbers which are not integers. They have a decimal point and a fractional part. Depending on the value the decimal point floats around in the number, hence the name float. C# provides one box size which can hold a real number. A standard float value has a range of 1.5E-45 to 3.4E48 with a precision of only 7 digits (i.e. not as good as most pocket calculators). An example would be 3.124512 \* 10 to the 48 (48 zeros).

If you need more precision (although your programs will use up twice as much computer memory and run more slowly) you can use a double box instead (double is an abbreviation for double precision). This has a range of 5.0E-324 to 1.7E308 and a precision of 15 digits. An example would be 3.12451284937542 \* 10 to the 308 (308 zeros).

Example float variables could be something which held the average price of ice cream, or the distance to the moon in meters:

float averageIceCreamPrice;

float moonDistance;

An example of a double variable could be something which held the width of the universe in light years:

double univWidthInLightYears;

Finally, if you want the **ultimate in precision** but require a **slightly smaller range** you can use the decimal type. This uses twice the storage space of a double and holds values to a precision of 28-29 digits. It is used in financial calculations where the numbers are not so large but they need to be held to very high accuracy.

decimal robsPayCheck;

#### **real literal values**

There are two ways in which you can store floating point numbers; as float or as double. When it comes to putting literal values into the program itself the compiler likes to know if you are writing a floating point value (smaller sized box, less memory) or double precision (larger sized box, more memory).

A float literal can be expressed as a real number with an **f** after it:

2.5f

A double literal is expressed as a real number without the f:

3.5

You can also use exponents to express double and float values:

9.4605284E15

This is a double precision literal value which is actually the number of meters in a light year. If you put an f on the end it becomes a floating point literal value.

Unlike the way that integers work, with real numbers the compiler is quite fussy about how they can and can't be combined. This is because when you move a value from a double precision variable into an ordinary floating point variable some of the precision is lost. This means that you have to take special steps to make sure that you as a programmer make clear that you want this to happen and that you can live with the consequences. This process is known as casting and we will consider it in detail later.

### Storing Text

Sometimes the information we want to store is text. This can be in the form of a single character; at other times it will be a “string of characters, called a string. C# provides variables for looking after both of these types of information:

#### **char variables**

A char is a type of variable which can hold a single character. A character is what you get when you press a key on a keyboard or display a single character on the screen. C# uses a character set called UNICODE which can handle over 65,000 different character designs including a wide range of non-English characters. An example of a character variable could be something which held the command key that the user has just pressed:

char commandKey;

You express a character by enclosing it in **single** quotes:

'A'

This means "the character A". It is what your program would get if you asked it to read a character off the keyboard and the user held down shift and pressed A. VS will highlight a character literal in red.

#### **Character Escape Sequences**

This leads to the question "How do we express the ' (single quote) character". This is achieved by the use of an escape sequence. This is a sequence of characters which starts with a special escape character. Escape in this context means "escape from the normal char processing and consider this *sequence* as a single character". The escape character is the **\** (backslash) character. Possible escape sequences are:

|  |  |
| --- | --- |
| Character | Escape Sequence name |
|  |  |
| \' | Single quote |
| \" | Double quote |
| \\ | Backslash |
| \0 | Null |
| \a | Alert |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical quote |

These conventions go all the way back to the 1920’s when devices called Teletype Machines were replacing the Telegraph. The effect of these escape sequences depends on the device you send them to. Some systems will make a beep when you send the Alert character to them. Some clear the screen when you send the Form feed character.

You can use them as follows: char beep = '\a' ;

Note that the a must be in lower case. Even though it looks like you are putting 2 characters into one char variable, you are not. \a is a code for a special single char value that the computer understands.

### Storing strings

A data type (kind of box) which can hold words and sentences is a string. In C# a string can be very short, for example "Rob", or it can be very long, for example "War and Peace" (that is the entire book, not the three words!).

An example of a string variable could be something which holds the line that the user has just typed in:

string commandLine

A string literal value is expressed enclosed in double quotes:

string userMessage = "Good afternoon" ;

The string can contain the escape sequences above:

string userMessage = "Good \a afternoon" ;

If we print this string it would print out:

Good afternoon

and it would try to ring the bell between writing Good and afternoon. If I am just expressing text with no escape characters or anything strange I can tell the compiler that this is a **verbatim string**. I do this by putting an **@** in front of the literal:

string userMessage = @"Good \a afternoon" ;

If I print this string, the @ tells it to ignore escape characters, so I will get:

Good \a afternoon

This can be useful when you are expressing things like file paths; c:\users\joe\documents\dog.jpg

The verbatim character has another trick, which is that you can use it to get string literals to extend over several lines:

@"The quick brown fox

jumps over the lazy dog"

This expresses a string which extends over two lines. The line breaks in the string are preserved when it is stored.

### Storing Booleans

A bool (short for boolean) variable is a type of box which only takes one byte which can hold whether or not something is true. Sometimes that is all you want. If you are storing whether or not a subscription has been paid or not there is no need to waste space by using a type which can hold a large number of possible values. Instead you just need to hold the states true or false. These are the only two values which the bool type allows.

An example of a bool variable could be one which holds the state of a network connection:

bool networkOK;

Boolean literal values are easily expressed as either true or false both of which are reserved C# words:

networkOK = true;

This would be a mistake as the string true is not the same as the reserved work true

networkOK = "true";

### Choosing the type of your variables

Choosing the right type for a variable is something of a skill. When considering how to store data it is important to remember where it comes from. In the example above I've used the double type to hold the width and height of a window. This is a waste of memory. It does not take into account the precision required or indeed how accurately the window can be measured. I would suspect that my glazing salesman will not be able to measure to accuracy greater than 1 mm and so it would make sense to only store the data to that precision.

This illustrates another metadata consideration, when you are given the prospect of storing floating point information; find out how it is being produced before you decide how it will be stored. As an example, you might think that being asked to work with the speed of a car you would have to store a floating point value. However, when you find out that the speed sensor only gives answers to an accuracy of 1 mile per hour, this makes the job much simpler, as you could use an Int16

### Naming Variables: Identifiers

In C# an identifier is a name that the programmer chooses for something in the program. The name of a variable is more properly called an identifier. We will see other places where we create identifiers. C# has some rules about what constitutes a valid identifier:

* All identifiers names must start with a letter or the underscore "\_" character.
* After the letter you can use both letters, numbers, and the underscore "\_" character.

Upper and lower case letters are different, i.e. Fred and fred are different identifiers.

Here are a few example declarations, one of which are not valid (see if you can guess which one and why):

int **fred29** ;

float **jim\_bob** ;

string **29yesitsme** ;

One of the golden rules of programming is always give your variables meaningful names. Storing a person’s first name in a variable called **x**, or even **fn** is cryptic. Months later, when you are trying to find a bug in your program, names like **x** will make it much harder to remember what the code is doing. A much better variable name would be **firstName**. A common convention you should follow is to name variable with names starting with a lower case letter, and then capitalize the first letter of each word that you glue together to make a descriptive identifier (you cannot put spaces in an identifier). **billingDayOfTheMonth** is an example of good name. As you will see when you use VS, once you have created a nice identifier name, you will never have to type it all out again, as VS will auto-complete it as soon as you type the first few letters.

### Giving Values to Variables

Once we have our variable we now need to know how to put something into it (write it), and get the value out (read it). C# does this by means of an **assignment statement**. There are two parts to an assignment, the thing you want to assign and the place you want to put it, for example consider the following:

static void Main()

{

int first;

int second;

int third;

first = 1;

second = 2;

third = second + first;

}

The first part of the program should be pretty familiar by now. Within the Main function we have declared three variables, **first**, **second** and **third**. These are each of int type.

The last three statements are the ones which actually do the work. These are **assignment statements**. An assignment gives a value to a specified variable. The value that you enter on the right side must be compatible with the data type of the variable you are storing it into. The value to be assigned is calculated with what is called an expression. The ***equals*** sign in an **assignment statements** divides the left from the right. The computer will do everything you direct it to in the code on the right side (the expression), and the results of that computation will then be stored in the variable on the left side. So after the statement is executed, the variable is holding a value ***equal to*** the results of the code expression on the right. Remember, the computer will work from right to left, as in our last statement, where the computer first adds the two values on the right, and then it stores the answer into the left.

### Expressions

An expression is something which can be evaluated to produce a result. We can then use the result as we like in our program. Expressions can be as simple as a single value (e.g. 34) and as complex as a large calculation. They are made up of two things, ***operators*** and ***operands***.

#### **Operands**

Operands are things the operators work on. They are usually literal values or the identifiers of variables. In the program above first, second, third are identifiers and 2 is a literal value. A literal value is something which is literally there in the code. A literal value has a type associated with it by the compiler.

#### **Operators**

Operators are the things which do the work: They specify the operation to be performed on the operands (the data). Most operators work on two operands, one each side. In the program above ***+*** is the only operator we used.

Here are a few example expressions with the operands in black, and the operators in red.

**2 + 3 \* 4**

**-1 + 3**

**(2 + 3) \* rate**

These expressions are worked out (evaluated) by C# moving from left to right, just as you would yourself. Again, just as in traditional math, all the multiplication and division is performed first in an expression, followed by the addition and subtraction.

C# does this by giving each operator a priority. When C# works out an expression it looks along the expression for all the operators with the highest priority and does them first. It then looks for the next ones down and so on until the final result is obtained. Note that this means that the first expression above will therefore return 24 and not 20.

If you want to force the order in which things are worked out you can put parenthesis around the things you want done first, as in the final example. You can nest parenthesis inside parenthesis, provided you make sure that you have as many open ones as close ones. I strongly suggest you make your code very clear to yourself and the poor souls who come after you to fix your bugs, by putting brackets around everything. ***Relying on correct operator priority is just risky.*** It requires both you and anyone working on your code later to remember the precedence and get it correctly. Just make it clear; use parenthesis all the time.

Here is a list of common operators and what they do. I am not indicating their priority because we don’t care about the priority! We are good programmers and we always are explicit by using parenthesis.

|  |  |
| --- | --- |
| - | unary minus, the minus that C# finds in negative numbers, e.g. -1. Unary means applying to only one item. |
| \* | multiplication, note the use of the \* rather than the more mathematically correct but confusing x. |
| / | division, because of the difficulty of drawing one number above another on a screen we use this character instead |
| + | addition. |
| - | subtraction. Note that we use exactly the same character as for unary minus. |
| % | remainder operator. If you do integer math. For example, 8 % 3 will return a 2. |

### Changing the “Type” of Data

Whenever you move a value from one type to another, the C# compiler reviews what you are doing. It checks if the operation will cause information to be lost from the program. If I owed you 375 cents and I decided to change the “type” to dollars, and paid you 3 dollars instead, you might not be happy with the missing 75 cents. The compiler considers every operation in terms of "widening and narrowing" of values. The general principle which C# uses is that if you are "narrowing" a value it will always ask you to explicitly tell it that this is what you want to do. If you are widening there is no problem.

To understand what we mean by these terms we could consider suitcases. If I am packing for a trip I will take a case. If I decide to switch to a smaller case I will have to take everything out of the large case and put it into the smaller one. But it might not have room, so I have to leave behind one of my shirts. This is "narrowing".

However, if I change to a bigger case there is no problem. The bigger case will take everything that was in the smaller case and have room for more. This is "widening".

In C# terms the "size" of a type is the range of values (the biggest and smallest) and the precision (the number of decimal places). This means that if I write:

int i = 1;

float x = i;

This works fine because the floating point type can hold all the values supported by the integer type without losing any accuracy. However the opposite case:

float x = 1;

int i = x ;

would cause the compiler to complain (even though at the moment the variable x only holds an integer value). The compiler is concerned that you may be discarding information by such an assignment and treats it as an error. Note that this applies within floating point values as well, for example:

double d = 1.5;

float f = d

would cause an error as well, since the compiler knows that a double is wider (can store more significant digits) than a float.

#### **Casting**

We can force C# to regard a value as being of a certain type by the use of *casting*. A cast takes the form of an additional instruction to the compiler to force it to regard a value in a particular way. You cast a value by putting the type you want used in parenthesis just in front of it. For example:

double d = 1.5;

float f = (float)d;

In the above code the message to the compiler is "I don't care that this assignment could cause the loss of information. I, as the writer of the program, will take the responsibility of making sure that the program works correctly". You can regard casting as the compiler's way of washing its hands of the problem. If a program fails because data is lost it is not because the compiler did something incorrectly.

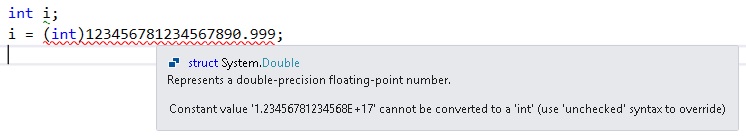
As we saw above, each type of variable has a particular range of possible values, and the range of floating point values is much greater than that for integers. This means that if you do things like this:

int i;

i = (int) 123456781234567890.999;

the cast would put a value in i that is very inaccurate.

VS will give this error message:



If you REALLY want to do that, you can tell VS to switch to an unchecked syntax, but it would be very rare case that this would be correct thing to do. In general, it is up to you when you write your program to make sure that you never exceed the range of the data types you are using. The compiler and the run time system might not notice and the code will generate bad data. Here is another example of a dangerous cast operation:

int i;

i = (int) 1.999;

The code above takes 1.999 (which would be compiled as a value of type double) and casts it to int. This process discards the fractional part, which means that the variable **i** will end up with the value 1 in it, even though the original number was much closer to 2. You should remember that this truncation takes place whenever you cast from a value with a fractional part (float, double, decimal) into one without (int, long).

#### **Casting and Literal Values**

We have seen that you can put "literal" values into your program. These are just values you want to use in your calculations. For example, in our window program we had to multiply the wood length in meters by 3.25 to convert the length value from meters to feet (there are about 3.25 feet in a meter).

C# keeps examines the things that it is combining, and this includes how it allows literal values to be used. This is so that statements like:

int i;

i = 3.4 / "hello";

will be flagged as a mistake. However, consider this:

float x;

x = 3.4;

This code looks perfectly legal. However, it is not. This is because the literal value 3.4 is a double precision value when expressed as a literal, and the variable x has been declared as a floating point. If I want to put a floating point literal value into a floating point variable you can use casting:

float x;

x = (float)3.4;

This casts the double precision literal into a floating point value, so that the assignment works.

To make life easier the creators of C# have added a different way we can express a floating point literal value in a program. If you put an **f** after the value this is regarded as a floating point value instead of defaulting to a type double. This means that:

float x;

x = 3.4f;

would compile correctly.

#### **Types of Data in Expressions**

When C# uses an operator, it makes a decision as to the type of the result that is to be produced. Essentially, if the two operands are integer it says that the result should be integer. If the two are floating point it says that the result should be floating point. But what if they don’t match? This can lead to problems, consider the following:

1/2

1/2.0

You might think that these would give the same result. Not so. The compiler thinks that the first expression, which involves only integers, should give an integer result. It therefore would calculate this to be the integer value 0 (the fractional part is always truncated with integer math). The second expression, because it involves a floating point value would however be evaluated to give a double precision floating point result, the more accurate answer of 0.5 .

If you are intentionally depending on the fact that the compiler will allow you to “widen”, it’s a good idea to use a cast to make it clear to the next person that reads your code that you are intentionally doing this. Looking at our example from above, while this is legal:

int i = 1;

float x = i;

it is better to make it clear:

int i = 1;

float x = (float)i;

### Constant variables

When you define a variable and give it a value, at any time later in the program you can change the value. As you will see, it is often very convenient to create a variable that holds a value that you want to restrict so that it can never be changed by your code. We can do this by marking a variable as a **constant**, i.e. it can never be changed.

const double PI = 3.141592654;

This means that you can do things like:

const double PI = 3.141592654;

double circ = rad \* 2 \* PI;

This has several advantages. All throughout your program when you want to do a calculation that needs the value of pi, you will not have to remember what pi really is. You look it up once, save it in your const variable at the beginning of your code, and use the name PI instead of a number throughout. By marking it as const, you will let the compiler notify you if you ever make a mistake and do something that would try and change the value of PI. Also, anyone else reading your code will have a better chance of correctly understanding it, as you have replaced 3.141592654 which might look to them like a random number somewhere in your code, with a name that is clear.

It is a good practice to create variables at the beginning of your program that hold special numbers that you intend to use all through the code. By giving these special numbers a name, and then using the name all though your code, if you ever want to change that special number, you only have to change one line of code, and not go hunting all through the code looking for every time you used that number. While it is not likely that you will ever want to change PI, you may very well change the start time from 8 am to 9 am, or the standard tip from 15 percent to 20 percent, or the price of a movie form 8.99 to 9.99.

We will do this in our window program, setting values at the top. If we ever decide we can handle larger windows, we won’t have to look all through the code for some use of the number 5.0 or 3.0. We can just change the value of those const variables at the beginning.

const double MAX\_WIDTH = 5.0;

const double MIN\_WIDTH = 0.5;

const double MAX\_HEIGHT = 3.0;

const double MIN\_HEIGHT = 0.75;

There is a convention that you always set constant variables names to all CAPITAL LETTERS. This is so that when others read your program they can see at a glance that a variable is a const and they will know not to try and change it.

## Check your understanding

#### **Questions 1:**

We save information in memory locations, to save it and later get it back, we have to give these locations what?

#### **Questions 2:**

Why do you think a variable is called a variable?

#### **Questions 3:**

What does putting const in from of a variable declaration accomplish?

#### **Questions 4:**

You can assign a value to a variable by either setting it equal to another variable, or setting it to the results of a method call, or you can?

#### **Questions 5:**

What is the difference between these two variables?

int dogWeight1;

uint dogWeight2;

#### **Questions 6:**

What is the difference between these two variables?

int dogWeight1;

float dogWeight2;

#### **Questions 7:**

What is the difference between these two variables?

char dog1;

string dog2;

#### **Questions 8:**

What would be the result of executing this line of code?

string userMessage = "Good \a afternoon";

#### **Questions 9:**

What is wrong with this line of code?

bool networkOK = "true";

#### **Questions 10:**

What is the difference in what the **+** sign does in these two code segments

int x = 3;

int y = 4;

int z = x + y;

string x = "3";

string y = "4";

string z = x + y;

#### **Questions 11:**

What will the value of x be?

float x = 3.0f + 4.0f / 5.0f + 7.0f \* 2.0f;

What will the value of x be?

float x = (3.0f + 4.0f) / (5.0f + (7.0f \* 2.0f));

Which one is “correct”? Are you sure?

#### **Questions 11:**

What is wrong with these lines of code?

const double MAX\_WIDTH = 5.0;

MAX\_WIDTH = 7.2;

#### **Questions 11:**

What is the difference between these 2 sets of 2 lines of code?

int i = 1;

float x = i;

int i = 1;

float x = (float)i;

# Chapter 5 – Making Decisions

Our first window program is very simple, it runs straight through from the first statement to the last, and then stops. More often you will come across situations where your program must change what it does according to the data which is given to it. Basically there are four variations of program flow:

1. Straight line, as we saw in the first programs.

2. Chosen (do I go left or right?) depending on a given condition, the subject of this chapter.

3. Repeated according to a given condition (stir batter until it thickens), the subject of the next chapter.

4. Event driven. This is not really a fourth variations, but more of a style of programming. An event driven program is made up of pieces (using some combination of the 3 variations above), and different pieces execute based on events. Events are things like a mouse click, new mail arrives, a time of the day, and more.

Practically every program ever written is composed of those variations. You can use this to good effect when designing an overall view of how your program is going to work.

### Conditional Execution - if

The program above to work out the wood and glass for our window store is nice; in fact our customer will probably be quite pleased with it. However, it is not perfect. The problem is not with the program, but with the usability.

If you give the program a window width of **-1** it goes ahead and works out a nonsensical result. Our program does not have any checking for invalid widths and heights. The user might have grounds for complaint if the program fails to recognize that he has given an unusable value. What we want to do is notice incorrect inputs and tell the user that he has done something incorrect. In our program specification, which we give the customer, we have said something like (this is our metadata):

The program will reject window dimensions outside the following ranges:

width less than 0.5 meters

width greater than 5.0 meters

height less than 0.75 meters

height greater than 3.0 meters

In order to allow us to do this, the program must notice invalid input values and reject them. To do this we can use the **if** construction:

if (**condition**)

**statement** or **block** we do if the condition is true

The **condition** determines what happens in the program. So what do we mean by a **condition**? Whatever valid C# code you enter inside of the parenthesis will be evaluated, if the result is a Boolean true, the following **statement** or **block** will be executed. If it is false, it will not be. Let’s look at a few examples.

if (true) // true is always true

{

Console.WriteLine("This message WILL be written to the console");

}

if (3 == 3) // is 3 equal to 3? Yes! (note the use of == not =)

{

Console.WriteLine("This message WILL be written to the console");

}

You will probably make a mistake of using only one **=** sign many times. One = sign is what we use to separate the left and right sides of a program statement. The right side is evaluated and the answer is copied into the variable on the left side of the **=** sign. That is how we use one **=** sign. But when testing **conditions**, we use two, **==,**  to ask the question, is the left side equal to the right side?”

Now let’s look at one where the condition is not true:

if (3 == 4) // is 3 equal to 4? No!

{

Console.WriteLine("This message will NOT be written to the console");

Console.WriteLine("And neither will this message be written to the console");

}

Since 3 is not equal to 4, neither of the two lines inside the code block, as marked with an opening **{** and closing **}** will get executed. We can include one statement or 400 inside of the code block. They will all either be executed, or none of them will be, based on if the condition evaluates to true or false.

if (3 == 4) // This one uses ONE statement, not a block

Console.WriteLine("This message will NOT be written to the console");

Console.WriteLine("But this message WILL be written to the console");

**TRICKY!!** As I stated in the definition of our if conditional, what follows the condition test is either **one** statement, or **one** block. In the example right above, since I did not put the code in inside of a set of parenthesis **{** one or more statements **}**, only the 1st Console.Writeline statement is subject to the true or false test. The second Console.Writeline statement will always be executed, no matter if the condition evaluates to true or false.

I find not using the parenthesis **{** one or more statements **}** form leads to many accidental bugs, and I do not let my students code that way. I insist that, even of the if conditional applies to only one line of code, they must use the parenthesis **{** one statement **}**.

Before we explore the if statement more, we need to be more clear about how you can construct the condition part of an if statement.

### Conditions and Relational Operators

To make conditions work for us we need a more complete set of additional relational operators which we can use in logical expressions. Relational operators work on operands, just like numeric ones. However any expression involving them can only produce one of two values, true or false. Here is the most common relational operator:

**== (equals)**  If whatever is on the left hand side and the right hand computer to the same result, then the expression has the value true. If they are not equal the value is false.

if (4 == 5)

would evaluate to false. Note that it is not particularly meaningful to compare floating point variables to see if they hold exactly the same values. Because of the fact that they are held to limited precision you might find that conditions fail when they should not. For example the following equation:

x = 3.0 \* (1.0 / 3.0) );

may well result in x computing to be 0.99999999, which would mean that:

if (x == 1.0)

would evaluate to be **false** - even though mathematically the test should evaluate to **true**.

Here are other common relational operators:

**!= (not equal)** The reverse of equals. If the operands are not equal the expression has the value true, if they are equal it has the value false. Again, this test is not advisable for use with floating point numbers.

**< (less than)** If the operand on the left is less than the one on the right the value of the expression is true. If the left hand operand is larger than or equal to the right hand one the expression gives false.

**> (greater than)** If the operand on the left is greater than the one on the right the result is true. If the operand on the left is less than or equal to the one on the right the result is false.

**<= (less than or equal to)** If the operand on the left is less than or equal to the one on the right you get true, otherwise you get false. It is quite valid to compare floating point numbers in this way.

**>= (greater than or equal to)** If the operand on the left is greater than or equal to the one on the right you get true, otherwise it is false. Again, it is quite valid to compare floating point numbers in this way.

**! (not)** This can be used to invert a particular value or expression, for example you can say !true, which is false, or you could say: (!x==y) which means the same as (x!=y). You use not when you want to invert the sense of an expression. You will see later as we build more complicated condition expressions how this one will be used often.

Please look through these examples and decide which of the nine Console.WriteLine statements will actually get executed.

int big = 300;

int small = 10;

int sameAsSmall = 10;

int sameAsBig = 300;

// #1

if (small != big)

{

Console.WriteLine("These numbers are not the same.");

}

// #2

if (small < big)

{

Console.WriteLine("small is less than big.");

}

// #3

if (big > small)

{

Console.WriteLine("big is larger than small.");

}

// #4

if (small <= big)

{

Console.WriteLine("small is less than, or the same as big");

}

// #5

if (small >= big)

{

Console.WriteLine("small is greater than, or the same as big");

}

// #6

if (small <= sameAsSmall)

{

Console.WriteLine("small is less than, or the same as sameAsSmall");

}

// #7

if (big >= small)

{

Console.WriteLine("big is greater than, or the same as small");

}

// #8

if (big >= sameAsBig)

{

Console.WriteLine("big is greater than, or the same as sameAsBig");

}

// #9

if ( ! (big == small) )

{

Console.WriteLine("big is not not eaqual to small");

}

(All *but one* of them will write to the console. Paste the code into Visual Studio and give it a try.)

### Combining Logical Operators

Sometimes we want to make more complicated choices, for example to test for a window width being valid we have to test that it is greater than the minimum **and** less than the maximum. Often you have a choice for how to achieve this.

#### **Nested if statements**

You can place an if statement inside of an if statement, like this:

// test for window width that it is greater than the minimum AND less than the maximum

if (width >= 0.5)

{

if (width <= 5.0)

{

// This code will only be executed if BOTH conditions are true

// calculate ( width + height ) \* 2 \* 3.25

woodLength = 2 \* (width + height) \* 3.25;

// calculate width times height

glassArea = (width \* 3.25) \* (height \* 3.25);

}

}

As we progress in the course we will very often nest an if inside of an if. We will even nest an if inside of an if that is nested inside of an if! (In which case all three conditions will have to be true.)

#### **Combine logical expressions**

But we have another way of doing this, using a combination of logical expressions. C# provides additional operators to combine logical values:

**&& (and)**  If the operands on **both** sides of the && are true, then the result of the && is true. If **either** one of them is false the result is false, for example:

if (width >= 0.5) && (width <= 5.0)

{

Console.WriteLine("Your window size is acceptable");

}

this would be true if the width was valid according to our above description. I’ve put brackets around each of the conditions in this example, so that it is easier to see what is going on. However, the compiler is able to work out that the && needs to be applied between the result of the two logical expressions, so they are not actually required. But we write code to be easily read, so we will always use them.

Caution: A common mistake is to write these combined statements the same way we do in English. For example, you would not normally say “Are you going to the movies with Judy AND are you going to the movies with Sam”? In English we assume the first part is common to both tests, so we would say “Are you going to the movie with Judy AND Sam?”.

So using are example above, here is the WRONG way of stating that:

if (width >= 0.5 && <= 5.0)

When you write combined expressions, **each** expression must be a complete expression, like this;

if (width >= 0.5 && width <= 5.0)

Now let’s look at the other common operator for combining test expressions, the “or”.

**|| (or)** If **either** of the operands on one side or the other of the || are true the result of the expression is true. The expression is only false if ***both*** operands are false. Using our examples above, first as we would say it in English: “Are you going to the movies with Judy OR with Sam”? If you intend to go with Judy, or with Sam, or with both Judy and Sam, the answer would be “true”. But as above, in C# we cannot cut short the expression, so we would have to state it as ““Are you going to the movies with Judy OR are you going to the movies with Sam”? That sentence seems like you would say yes only if you were going with one or the other. But in computer logic, including C#, it would be interpreted as yes if you are going with either one, or both.

Now looking at our window code, this next statement will be true if the window size is unacceptable:

if ( (width < 0.5) || (width > 5.0) )

{

Console.WriteLine("Your window size is not acceptable");

}

If you find that you are making huge and complicated conditions to get a statement you need:

if ( ( (this > 8) || (that < 19) ) && (otherThing == 15 ) && (day != "Friday" )

{

// do something only if these conditions are met

}

You might find that you can make things clearer by using multiple if statements, and indenting the code in an appropriate way. This will make the code easier to debug as well, because you can watch the program step through your conditions one at a time, instead of going off in a particular direction based on a huge condition you will then have to try and decipher. Here is that same exact logic expressed with if statements:

if ((this > 8) || (that < 19))

{

if (otherThing == 15)

{

if (day != "Friday")

{

// do something only if these conditions are met

}

}

}

### Using the else and if else structures for better logic flow

As we used the if structure above, you might have thought to yourself, “but if the if condition is false, and I can’t do the thing I wanted, how can I express what to do instead?” Well, just like in real life, “I think I’ll have waffles for breakfast if the strawberries are fresh, **else** I’ll get the pancakes.”

if (strawberries are fresh) // test a condition

{

Waiter, I’ll have the waffles with strawberries

}

else // there is no condition with the else,

// you ALWAYS do this when the if condition was false.

{

Waiter, I’ll have the pancakes

}

Furth more, if there are several choices and you need to pick the best choice, you can use the if else

if (strawberries are fresh) // test 1st condition

{

Waiter, I’ll have the waffles with strawberries

}

else if (blueberries are fresh) // test 2nd condition

{

Waiter, I’ll have the waffles with blueberries

}

else if (pancakes are buttermilk pancakes) // test 3rd condition

{

Waiter, I’ll have the buttermilk pancakes

}

else // again, there is no condition with the else,

// you ALWAYS do this if all the prior conditions were false.

{

Waiter, that’s disappointing, I’ll just have toast

}

In this example, if the strawberries were **not** fresh, but the blueberries **were** fresh, then the first if would have been false, so you would skip down to the 2nd else if test, which in this case returns true, so you order the waffles with blueberries and the next else if and else statements are never executed. No matter what the state of strawberries, blueberries, and buttermilk, you will only execute one of the four Waiter statements.

Let’s go back to our window program and let the customer pick a wood finish using our new else if and else structures.

Console.WriteLine("Please enter black, white, or natural for the wood paint.");

string woodFinish = Console.ReadLine();

if (woodFinish == "black")

{

Console.WriteLine("Ok, we will paint the window frame black.");

}

else if (woodFinish == "white")

{

Console.WriteLine("Ok, we will paint the window frame white.");

}

else

{

Console.WriteLine("Ok, we will leave the window frame unpainted");

}

In our example, there is no need to test “if they want natural”, since we are only offering 3 choices, and by the time we skip over the if and then the else if to get to the final else, the only choice left is natural.

### Applying our new knowledge to our window program

Here is our program updated using const variables and our new if structure. Paste it into VS, run it, and examine it until you are sure you understand every single line.

using System;

/\*

\* This program asks the user for the size of a window.

\* It then calculates the total length of wood needed for the frame

\* and the total area of the glass.

\*/

class Progam

{

static void Main()

{

double width;

double height;

double woodLength;

double glassArea;

const double MAX\_WIDTH = 5.0;

const double MIN\_WIDTH = 0.5;

const double MAX\_HEIGHT = 3.0;

const double MIN\_HEIGHT = 0.75;

string widthString;

string heightString;

// read in the width

Console.Write("Give the width of the window : ");

widthString = Console.ReadLine();

width = double.Parse(widthString);

if (width < MIN\_WIDTH)

{

Console.WriteLine("Width is too small.\n\n ");

Console.WriteLine("Using minimum");

width = MIN\_WIDTH;

}

if (width > MAX\_WIDTH)

{

Console.WriteLine("Width is too large.\n\n");

Console.WriteLine("Using maximum");

width = MAX\_WIDTH;

}

// read in the height

Console.Write("Give the height of the window : ");

heightString = Console.ReadLine();

height = double.Parse(heightString);

if (height < MIN\_HEIGHT)

{

Console.WriteLine("Height is too small.\n\n");

Console.WriteLine("Using minimum");

height = MIN\_HEIGHT;

}

if (height > MAX\_HEIGHT)

{

Console.WriteLine("Height is too large.\n\n");

Console.WriteLine("Using maximum");

height = MAX\_HEIGHT;

}

// calculate ( width + height ) \* 2 \* 3.25

woodLength = 2 \* (width + height) \* 3.25;

// calculate width times height

glassArea = (width \* 3.25) \* (height \* 3.25);

// and print it

Console.WriteLine("The length of the wood is " + woodLength + " feet");

Console.WriteLine("The area of the glass is " + glassArea + " square feet");

// this line just prevents the window from closing until you type the Enter key

Console.ReadLine();

}

}

### Range Checking with if statements

Beware of a common coding mistake made when selecting an answer based on a range. Imagine you have a program to help some decide what size T-shirt they should buy. Examine this code and see if you can spot the problem.

string size = "";

Console.Write("How much do you weigh?");

int weight = Convert.ToInt32(Console.ReadLine());

if (weight < 50)

{

size = "small";

}

if (weight < 100)

{

size = "medium";

}

if (weight < 150)

{

size = "large";

}

Console.WriteLine("Please select a {0} T-shirt", size);

Imagine you weighed 45 pounds, what size would the program tell you to get?

45 satisfies all three if statements, so you will end up with the last one, a large. It is important to use if else structures to avoid this, that way when a condition is satisfied, it does not look at the other choices. This code will work correctly.

string size = "";

Console.Write("How much do you weigh?");

int weight = Convert.ToInt32(Console.ReadLine());

if (weight < 50)

{

size = "small";

}

else if (weight < 100)

{

size = "medium";

}

else if (weight < 150)

{

size = "large";

}

Console.WriteLine("Please select a {0} T-shirt", size);

### Using the switch statement to make decisions

Many times you will have nested if/if else/else structures that are making a decision based on different variables, such as the temperature, is it raining, is the barometer rising or falling, etc. But sometimes you are choosing among several options based on *just one variable*, as we did above with woodFinish, then a switch structure works very well in place if a series of if / if else / else statements. The above code could achieve the same results using this code:

Console.WriteLine("Please enter black, white, antiqueWhite or natural for the wood paint.");

string woodFinish = Console.ReadLine();

switch (**woodFinish**)

{

case **"black"**:

Console.WriteLine("Ok, we will paint the window frame black.");

break;

case **"white"**:

Console.WriteLine("Ok, we will paint the window frame white.");

break;

case **"antiqueWhite"**:

Console.WriteLine("Ok, we will paint the window frame antiqueWhite.");

break;

case **"natural"**:

Console.WriteLine("Ok, we will paint the window frame natural.");

break;

default:

Console.WriteLine("Sorry, that was not a valid choice.");

break;;

}

Notice I added a 4th choice to make the use of the switch even more compelling. The switch has a few advantages when it can be used. Probably most important, is how clear the code is about what is going on. We are choosing one of 4 code paths based on one single variable. It allows for (but does not require) the default case, which is the code it will run if the item we are switching on (woodFinish) does not match any of the case choices we coded.

The format for a switch starts with the key word switch followed by an expression inside of a set of parenthesis, that is followed by a pair of { } to mark a code block. Inside of those braces you add 1 or more case statements.

The expression can be any C# code as long as when that code is done executing what is left is a single value of some data type. In this example, my expression is simply a string variable, woodFinish. It could have been something much or just a bit more complicated, such as x + y where x and y are two int variables.

Each case section (possible code path) you add is comprised of the key word case and then a value you want to compare to the expression and then that line must end with a colon **:** (NOT a semicolon **;** ). Note carefully, that the item you want to compare to the expression result must be of the same data type. In my example, since my expression resolves to be a string, each case statement must be followed by a string, so that the code can compare a string to a string. If instead I had a switch with an int expression, then the compare-to item after each case statement would also have to be an int.

switch (**x + y**)

{

case **17**:

Console.WriteLine("Ok, we will send you 17 of them.");

break;

case **3**:

Console.WriteLine("Ok, we will send you 3 of them.");

break;

}

After each case line comes zero to as many as you need lines of C# code. My example only has one line of code, but it can certainly be 3 lines, or 13, or 300! At the end your lines of code in each case section, you must add the keyword break. break always causes the code to leave whatever structure you are in. In this case, it leaves the switch statement and moves on to whatever code follows.

You can also add the special case, default, which is a match to the expression no matter what the expression is, as long as it does not match any other case line. Typically the default is at the bottom.

#### **One unique case feature**

I said that any case line must be followed by zero to as many as you need lines of C# code and then the keyword break. There is an exception, imagine that the fee for a bus ride is $5 Monday thru Friday, $4 on Saturday, and $3 on Sunday. If you want **multiple case statements to all execute the same code**, you can structure the switch like this:

string dayOfWeek = Console.ReadLine();

int fare = 0;

switch (dayOfWeek)

{

case "Monday":

case "Tuesday":

case "Wednesday":

case "Thursday":

case "Friday":

fare = 5;

break;

case "Saturday":

fare = 4;

break;

case "Sunday":

fare = 3;

break;

default:

Console.WriteLine("Sorry, that was not a valid choice.");

break;

}

If a case section has ***any*** line of C# code, then it must end with a break, but if it has **no** lines of code **and** you want the same behavior for this case as the one that follows, then the case section can be otherwise empty and it will do what C# code is in the next case statement under it..

## Check your understanding

#### **Questions 1:**

Write and test a simple dice game. Using Console.WriteLine statements, explain the rules to the user. Have them type the enter key to start the game. Declare two int variables, dice1 and dice2 then using code like this, set their values:

Random myRandomObject = new Random(); // using a .NET library class Random

int dice1 = myRandomObject.Next(1, 7); // will emulate dice values

int dice2 = myRandomObject.Next(1, 7);

Write a message to the user showing what the value of their two dice are, and then …

##### Version 1

Using if statements, tell the user they won if the total of the dice is 7 or 11.

##### Version 2

Using if, if else, and else structures, tell the user they won if the total of the dice is 7 or 11, otherwise tell them they lost.

##### Version 3

Using if, if else, and else structures, tell the user they won if the total of the dice is 7 or 11, or if the 2 dice have the same value, otherwise tell them they lost.

##### Version 4

Using if, if else, and else structures, but also using a combination of logical expressions ( such as

if ((condition 1) || (condition 2)

tell the user they won if the total of the dice is 7 or 11, or if the 2 dice have the same value, otherwise tell them they lost.

#### **Questions 2:**

Write and test an automobile pollution testing program where the user is asked to enter 3 values. Save the three float answers as CM (Carbon monoxide), HY (Hydrocarbons), and NO (Nitrogen oxides). Next ask them how many miles the car has on the odometer and save that value as an int. Using if, if else, and else structures, tell the user if the car passes the pollution test or fails. If it fails, tell them which of tests it fails (1, 2, or 3 tests). The conditions to pass are:

For cars with less than 50,000 miles,

* Carbon monoxide must be less than 3.4
* Hydrocarbons must be less than 0.31
* Nitrogen oxides must be less than 0.4

For cars with 50,000 or more miles

* Carbon monoxide must be less than 4.2
* Hydrocarbons must be less than 0.39
* Nitrogen oxides must be less than 0.5

#### **Questions 3:**

Write and test a program that will compute what a salesperson commission will be. Start by asking the user to enter the dollar amount the salesperson sold, save that as a decimal variable. Now compute the commission. The commission rates are:

* salesperson sold more than $1000, gets a 8% commission
* salesperson sold more than $500, gets a 6% commission
* salesperson sold $500 or less, gets a 4% commission

After you calculate the commission, reduce it by 10% for income tax, and then write out the amount they should be paid. (Be careful with this one, check your code with several dollar amounts to make sure it always gets the correct answer.)

#### **Questions 4:**

Write a program that will calculate the total cost of a pizza. The code should ask the user if they want a S, M, or L pizza (small, medium, or large). Based on their answer, they should add $5.00, $7.00, or $9.00 to the total bill. Next ask the user if they want 0, 1, or 2 extra toppings. If they choose 1, add $1.00 to the bill. If they choose 2, add $1.50. Now add 9% tax to the bill, and write out a message telling them how much the total bill is. You can assume perfect data entry, no user mistakes. Keep track of the cost with a type decimal. Use if, if else, and else structures.

#### **Questions 5:**

Re-do your code from Question 4 to make use of a switch statement to determine the cost for a pizza based on size. Also re-do the logic for toppings to also use a switch, but this time the pricing should be if they choose 1, add $1.00 to the bill. If they choose 2, add $1.75, and if they choose 3, add $2.50.

#### **Questions 6:**

Write a new program. Ask the user to enter a number between 1 and 12 which corresponds to the month they were born in. Using a switch statement, tell the user if they were born in Spring, Summer, Fall, or Winter.

For our program we will just assume that

* Winter is months 12, 1, 2
* Spring is months 3, 4, 5
* Summer is months 6, 7, 8
* Fall is months 9, 10, 11

#### **Questions 7:**

Write and test a simple dice game very similar to the one in Question 1. But in this game, winning is getting a total of 2, 12, 7, or 11. Use a switch statement. (Unlike the game in Question 1, getting two dice the same is not an automatic win.)

#### **Questions 8:**

Write a new program to calculate bus fares. Ask the user to enter the day of the week, and then their age. Here are the rules for the fare.

* On weekdays, the normal fare is $6.00, but for people under 16 or over 65 years old, the fare is $5.00.
* On Saturdays, the normal fare is $5.00, but for people under 16 or over 65 years old, the fare is $4.00
* On Sundays, the normal fare is $4.00, but for people under 16 or over 65 years old, the fare is $3.00

Using a switch structure, decide which day of the week it is and in the appropriate case statements, then decide the fare based on age using if / if else / else, or if statements with compound logic. Write out the fare you calculate.

# Chapter 6 - Loops

Conditional statements allow you to do something if a given condition is true. However often you want to repeat something over and over again until something changes, or maybe a specific number of times.

C# has three ways of doing this, depending on precisely what you are trying to do. Note that we have three ways not because we need three but because they make life easier when you write the program. As with many aspects of programming, there are many ways to achieve a correct result. The more structures and patterns you learn, the faster you can see how to solve a problem and generally the more clear your code will be.

Let’s look again at our window program. If our salesman using our program inputs an unacceptable height, the program simply limits the value and they will have to re-run the program, re-entering all the data again and hoping they don’t make a mistake. This is not a “friendly” program. It would be much better that when we ask for a number, we keep asking over and over (in code, we call this looping) until they enter a valid number, and then continue with the rest of the program.

Let’s introduce the 3 types of loops, and then use each one of them to apply to this problem.

### while loop

The **while loop** does something, *while* some variable has (or does not have) a particular value. So it has the basic form of:

while (condition is true)

{

Execute a block of code, over and over and over.

}

Notice that if the block of code does not do something to change the condition, the code will be stuck in the loop forever. This is the famous infinite loop, an expression that even non-programmers use in everyday conversations.

Let’s use a **while loop** to achieve the multiplication of 14 times 5. (We could just say 14 \* 5, but this is an example to learn looping.) Our variable count is what is often called the control variable, as it controls when we get out of the loop. We initialize the control variable, we test it each time before doing the loop again, and each time though the loop we increment it (we count). In this example we will go through the loop 5 times and then exit.

int multiplicand = 14;

int multiplier = 5;

int answer = 0;

int count = 0;

while (count != multiplier)

{

count = count + 1; // keep track so we will exit the loop

answer = answer + multiplicand;

}

Console.WriteLine("Multiplying 14 by 5 is equal to: " + answer );

Notice how I wrote out the string message and then I “added” the answer to the string. This is using two techniques that we will cover in more detail later. The first is, if you put a plus sign between two strings, C# will glue them together into one string; this is called concatenation. The second is, answer is actually an integer, not a string, so this should be an illegal operation. In almost all cases, it will be, but this nice chunk of code, the WriteLine method of the class Console, is very forgiving and when you tell it to write almost any type of variable out to the console, it will automatically convert them into a string. More on these two ideas later.

While loops come in two forms, **definite** and **indefinite**. A definite while loop is one in which a counter is set up before entering the loop, and then that counter is used to determine how many times the loop will execute. IN the example we just looked at, we entered the loop with multiplier set to 5, so that while loop will always execute exactly 5 times. Indefinite loops are ones which execute until something changes the loop control variable. In many cases, this will be done by asking the user a question, such as “do you want to quit, Y or N”? So as you enter the loop, it is not pre-set how many times the loop will execute. One user may choose to execute the code in the loop 3 times, while another may choose 15 times.

Here is code that will do integer division using a **while loop**. It should ask the user for the dividend and the divisor but for simplicity, I hard coded them to 16 and 5. (Remember, integer division truncates any remainder, so 16/5 = 3 and we throw the left over 1 away.)

int dividend = 16;

int divisor = 5;

int answer = 0;

bool notDoneYet = true; // initialize our test condition

while (notDoneYet)

{

dividend = dividend - divisor; //keep subtracting

if (dividend >= 0) // until the dividend gets to 0 or less

{

answer = answer + 1; // keep track of how many times we subtracted

}

else

{

notDoneYet = false; // when the dividend is 0 or less,

// set the condition to be false so we leave the loop

}

}

#### Console.WriteLine("The answer is: " + answer);

If that example asked the user for the dividend and the divisor, it would still be a definite loop, as when you get to the while section, the number of times it will loop is fixed. Imagine you had a program that calculated the value of π (pi) to as many digits of accuracy as the user wanted. The program might write out **3** and then ask the user if they wanted more accuracy, and if they said yes, it would write out **3.1**, and ask again, and on and on and on, maybe getting all the way to **3.141592653589793238462643383279502884197169399375** before the user finally says that’s good enough. That code would be an indefinite while loop, as the decision to exit the loop is made based on user input, and there is no way of knowing beforehand how many times it will loop.

Whenever you want to create a while loop (or either of the next two loops) make sure you always:

* Initialize some loop **control variable**
* Test that **control variable** in the while expression
* Alter the value of the **control variable** someplace inside the loop so you can get out of the loop

While loops (and the other two loop forms) do not have to always count up by one. You can count up, or down, and by 1, 2, or any other increment.

### do while loop

The **while loop** starts by testing a condition, and then doing something in a code block. The **do while loop** is very similar, and is a bit like a **while loop** flipped upside down. With a **do while loop**, we do the code block ***first***, and then at the end of the code block, we test if we should go back and do the code block again, or if we are done. Its basic form is:

do

{

block of code, over and over and over.

}

while (condition is true);

So here is the same multiplication code using the **do while loop**:

int multiplicand = 14;

int multiplier = 5;

int answer = 0;

int count = 0;

do

{

count = count + 1; // keep track so we will exit the loop

answer = answer + multiplicand;

}

while (count != multiplier);

Console.WriteLine("Multiplying 14 by 5 is equal to: " + answer);

And here is the same division code using a **do while loop**:

int dividend = 16;

int divisor = 5;

int answer = 0;

bool notDoneYet = true; // initialize our test condition

do

{

dividend = dividend - divisor; //keep subtracting

if (dividend >= 0) // until the dividend gets to 0 or less

{

answer = answer + 1; // keep track of how many times we subtracted

}

else

{

notDoneYet = false; // when the dividend is 0 or less,

// set the condition to be false so we leave the loop

}

}

while (notDoneYet);

Console.WriteLine("The answer is: " + answer);

Why do we have both forms? Well, we could get by with just either one of them. They have one big difference, the **do while loop** will always go through the code block at least once. And sometimes the logic of your program is easier to implement if you are sure that code block will be executed at least once. Whereas, if the condition happens not to be true, a **while loop** might never go into the code block even once.

This difference shows up clearly when you use the **do while loop** when your loop condition is based on user input. So in the loop you ask the user a question, and based on their answer, you will continue to loop or leave the loop. But let’s start by looking at an example using a **while loop**.

int userNumber;

string userString;

Console.WriteLine("Enter a number, I will calculate its square.");

Console.WriteLine("Enter the number 0 if you want to quit");

userString = Console.ReadLine();

userNumber = Convert.ToInt32(userString);

while (userNumber != 0)

{

Console.WriteLine("The square of " + userNumber);

Console.WriteLine("is: " + (userNumber \* userNumber) );

Console.WriteLine("Enter a number, I will calculate its square.");

Console.WriteLine("Enter the number 0 if you want to quit");

userString = Console.ReadLine();

userNumber = Convert.ToInt32(userString);

}

This program uses another concept we will cover later. When the user enters their choice, your code is handed a string. But we can’t do math on a string, so we have to create an integer variable out of the string. There is a nice library class named Convert which has many methods to let you convert from one type of variable to another. In this case, we are using the convert from a string to an int method. More on this later.

Notice we are deciding **to enter** and also to stay in the loop based on what the user types. If the first time we ask the user for a number, they enter a zero, the code will not go into the loop, it will skip right over it. Also notice, that once we do fall into the loop, the only way of getting out is to change the control (the user’s number) to a zero. Any **while loop** must have a way of changing the control variable. Therefore, inside the loop, again we have to ask the user for a new number, read the number, and then again convert the number to an int. So the lines:

Console.WriteLine("Enter a number, I will calculate its square.");

Console.WriteLine("Enter the number 0 if you want to quit");

userString = Console.ReadLine();

userNumber = Convert.ToInt32(userString);

are repeated, once before going into the loop, and then again inside the loop at the bottom, so you are ready to go back up to the while condition and decide to stay in the loop or not. This is not terrible, but this same program implemented with a **do while loop** works a bit nicer.

int userNumber;

string userString;

do

{

Console.WriteLine("Enter a number, I will calculate its square.");

Console.WriteLine("Enter the number 0 if you want to quit");

userString = Console.ReadLine();

userNumber = Convert.ToInt32(userString);

if (userNumber != 0)

{

Console.WriteLine("The square of " + userNumber);

Console.WriteLine("is: " + (userNumber \* userNumber));

}

}

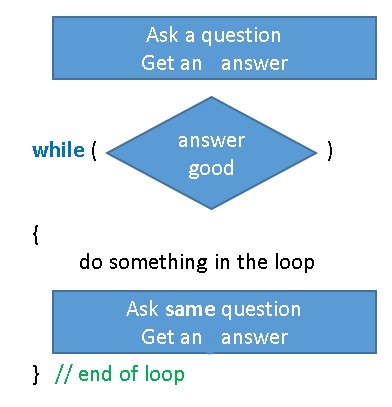
while (userNumber != 0);

Notice we do not have to repeat those four lines.

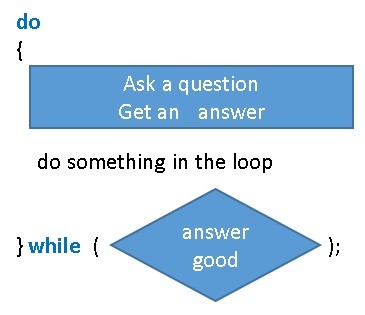
So again, the difference between the **while loop** and the **do while loop** is, ***when*** do you test if you are done looping, before the repeatable code block or after it, and therefore, a **while loop** might not ever go into the loop, whereas a **do while loop** will always go through the loop at least once.

#### **General form of while and do while loops**

**while loop asks question before loop and then asks same question at the bottom of loop**

****

**do loop asks question only once, and often as the first thing inside the loop**

****

### for loop

Now for the third choice for loops. The **for** **loop** is not really a new type of loop. Remember that a **while loop** can work one of two ways, indefinite or definite. It can loop until some condition changes, often based on user input, or it can loop a fixed number of times. If you look at our multiplication definite **while loop** example above, it will always loop an exact number of times, namely, as many times as is the size of the multiplier value.

A **for loop** is really a shorthand way of coding the *fixed-number00f-times,* definite **while loop**. It allows you to write in one line what a normal **while loop** takes three or four lines to do. The general structure is:

for (setup ; test if done ; update control variable)

{

block of code, done a specific number of times

}

Or as often used,

for (int i = 0; i < length; i++)

{

block of code, done a specific number of times

}

That same **for loop** code expressed as a **while loop** would be

int i = 0;

int length = 5;

while (i < length)

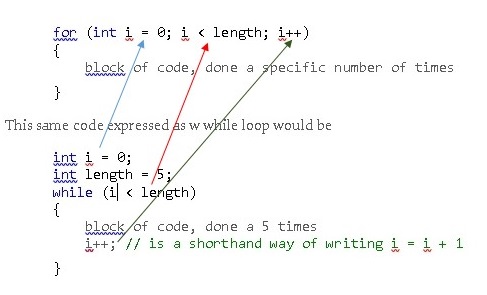
{

block of code, done a 5 times

i++; // note, this is a shorthand way of writing i = i + 1

}

Notice in this diagram how the 3 lines of the **while loop** have been combined into one line of the **for loop**.



I think it is wise to code with the **while loop** until you are quite comfortable with it, and then learn this nice **for loop** shortcut. One thing that is confusing about the **for loop**, even though the change to the control variable (the i++ code) is written on the same line as the **for loop**, it is always executed as the very last line of the block of code. All of these three loops types show up in most other computer languages, and as it turns out, the **for loop** is used much more often than the other ones. After we learn arrays and Lists, which are very common ways of storing data, you will see how convenient a **for loop** is for processing each item in the array or loop. More on that later. The most common variable used in a for loop to keep track of the number of times you are going through the loop (the control variable) is the letter i. This practice goes way back in the years, at least to the 1960’s days of FORTRAN and other early languages. Memory was expensive so variable names were kept very short. You might as well get used to seeing i, j, or other single letters used as the control variable, since it is done so often. But I suggest you use a more descriptive term for your control variables.

You are also free to use any comparison as needed, such as >=, ==, etc. And the iteration count can start at 0, 1, or 123, in other words, any number you like. Also, the i++ doesn’t have to be just counting by 1, you can make your **for loop** do only even numbers by saying instead, i=i+2. You can even start at a big number, and count backwards, using i--.

Let’s look our same multiplication problem in the structure of a **for loop**.

int multiplicand = 14;

int multiplier = 5;

int answer = 0;

for (int count = 0; count < multiplier; count++)

{

answer = answer + multiplicand;

// remember, the incrementing of the control variable actually happens here!

}

Console.WriteLine("Multiplying 14 by 5 is equal to: " + answer);

Compare this to the **while loop** solution above and make sure you can see how the various pieces of the **while loop** map over to this **for loop**.

While it would be possible to also do the division problem using the **for loop**, the **for loop** does not fit that problem very well, since the division problem is not really a *fixed-number00f-times* **while loop.** Instead, let’s look at another program that fits the nature of a **for loop** better. This program will raise a number to the power of another number. Such as 3 to the 4th power is 3\*3\*3\*3 = 81.

Console.WriteLine("Program raises an integer to the power of another integer.");

Console.WriteLine("Enter the base number you want to raise to a power.");

string baseString = Console.ReadLine();

int baseNumber = Convert.ToInt32(baseString);

Console.WriteLine("Enter the pwer number you want to raise the base to.");

string powerString = Console.ReadLine();

int powerNumber = Convert.ToInt32(powerString);

Int64 answer = 1;

for (int i = 1; i <= powerNumber; i++)

{

answer = answer \* baseNumber;

}

Console.WriteLine("The answer is: " + answer);

Notice that I start the loop counter,i, off at 1, not 0 and the code will loop until i is equal to the powernumber.

#### **Don't be too clever**

Some people like to show how clever they are by doing tricky things with the setup, condition and update parts of the **for loop** statements, which can do things other than simple assignment, test, and increment. Some programmers think they are very clever if they can do all the work "inside" the **for** part at the top and have an empty statement after it.

There is rarely need for such convoluted code. When you are writing programs the two things which you should be worrying about are "How do I prove this works?" and "How easy is this code to understand?" Complicated code does not help you or the poor college graduate that gets hired to maintain your code later to do either of these things.

#### **Examine 3 loops that almost do the exact same thing**

This one program has each of the three styles of loop. Each of them do the 3 steps

1. Initialize an index, a counter, a "Control Variable"
2. Test "are we done yet", and get out when we are
3. Somehow, change the Control Variable so you will get out of the loop

All 3 loops have these 3 steps labeled with the numbers 1, 2, and 3 as a comment.

static void Main(string[] args)

{

// all three loops will use these two variables

int counter;

const int LIMIT = 5; // if you make LIMIT 0, how will the 3 loops behave?

// While loop -------------------------------------------------------

Console.WriteLine("while loop");

counter = 1; //PART 1

while (counter <= LIMIT) //PART 2

{

Console.WriteLine(counter);

counter = counter + 1; //PART 3

}

Console.WriteLine();

Console.WriteLine();

// for loop --- A shorthand while statement with a definate loop. ----------------

Console.WriteLine("for loop");

for (counter = 1; counter <= LIMIT; counter++) //Part 1, 2, 3

{

Console.WriteLine(counter);

// counter++ really gets executed here

}

Console.WriteLine();

Console.WriteLine();

// do while loop --- Loop always executes at least once. ------------------------

Console.WriteLine("do while loop");

counter = 1; //PART 1

do

{

Console.WriteLine(counter);

counter = counter + 1; //PART 2

} while (counter <= LIMIT); //PART 3

Console.ReadLine();

}

In my sample code above, all three loops count the way people do, from 1 to 5, however most things in computer languages, such as arrays and loops, start from zero, not one. So this line

for (counter = 1; counter <= LIMIT; ++counter) // note the <=

is much more commonly written as:

for (counter = 0; counter < LIMIT; ++counter) // note the < and not <=

And it will do it 5 times, but starting from zero and going to 4.

If you use the "for tab tab" short cut to build your for loop, it will build the loop starting at 0.

### Nested Loops

If you are looping through something that is one dimensional, a loop works nicely. If you want to loop through something that has two dimensions, you will want to put a loop inside of a loop. This is called a nested loop. One loop is nested inside of another one. Generally the inside loop does some nice thing, but you want that repeated operation to itself be repeated over and over again, so you put the inside loop inside of an outside loop. Look at this code. Paste it into a Main method in a new VS project and run it.

static void Main(string[] args)

{

// a very small nested loop program.

for (int outerCounter = 0; outerCounter < 5; outerCounter++)

{

Console.WriteLine("in the outer loop, about to do the inner loop: {0}", outerCounter);

for (int innerCounter = 0; innerCounter < 5; innerCounter++)

{

Console.Write(" inner loop");

}

Console.WriteLine();

}

Console.ReadLine();

}

It is not a very useful program, but hopefully it helps you visualize a nested loop program in action.

Now paste this code into the Main method in a program

static void Main(string[] args)

{

int currentProduct;

for (int outsideCounter = 1; outsideCounter < 11; outsideCounter++)

{

for (int insideCounter = 1; insideCounter < 11; insideCounter++)

{

currentProduct = (outsideCounter \* insideCounter);

Console.Write("{0,4}", currentProduct);

// the {0,4} uses a placeholder and fill characters to make it line up nicely

}

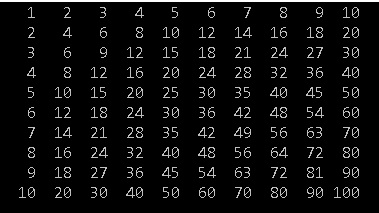
Console.WriteLine();

}

Console.ReadLine();

}

With this code, we are writing out the multiplication tables from 1 x 10 to 10 x 10. The inside loop is writing 10 numbers across the line. For the first line, the outside counter’s value is 1, so we are getting 1 x 1 through 1 x 10. When it is done writing those 10 values, we fall out of the inner loop, do a Console.WriteLine() to move down on the screen to the next line, and then the outside loop bumps up by one, and then it does the inner loop again, so we get 2 x 1 up to 2 x 10 on that line. The outside loop bumps up one over and over, each time calling the inside loop, which runs 10 times, giving us the 10 rows of 10 numbers.



When we get to the chapter on arrays, we will learn about 2 dimensional arrays. We will make extensive use of nested loops as we work with these.

#### **Breaking Out of Loops**

I showed the use of a break statement in the discussion of the switch decision structure. Sometimes you may want to escape from a loop whilst you are in the middle of it, i.e. your program may decide that there is no need or point to go on and wishes to leap out of the loop and continue the program from the statement after the loop’s code block.

You can do this with the break statement. As was the case in the switch, the break command causes the code path to leave the loop immediately. Your program would usually make some form of decision to quit in this way. You can break out of any of the three kinds of loop. In every case the program continues running at the statement after the last statement of the loop’s block. A good example of this would be if you are in a loop searching through a long list of names, when you find the name you want, there is no need to keep searching the list, so you can code in effect, “if this is the name I am looking for, get out of here”. We will see coding examples of this later.

#### **Operator Shorthands**

Since the **for** **loop** almost always uses an operation shorthand (i++), I will elaborate on these shorthands. So far we have looked at operators which appear in expressions and work on two operands, e.g.

window\_count = window\_count + 1;

In this case the operator is + and it is operating on the variable window\_count and the value 1. The purpose of the above statement is to add 1 to the variable.

C# provides a set of shortcuts for some simple statements. In this case,

window\_count++;

will do the same thing. We can express ourselves more succinctly and the compiler can generate more efficient code because it now knows that what we are doing is adding one to a particular variable. The ++ is called a unary operator, because it works on just one operand. It causes the value in that operand to be increased by one. There is a corresponding **--**operator which can be used to decrease (decrement) variables by one.

Note carefully, you can do either of window\_count++ or ++window\_count. Used in isolation,

window\_count++;

or

++window\_count;

there is no important difference. However used as part of a more complex statement, there is a big difference. Examine this code.

int window\_count = 12;

int door\_count = 4;

Console.WriteLine(window\_count++);

Console.WriteLine(++door\_count);

What will be written to the console? The correct answer is

12

5

After those 4 line of code, window\_count will be 13 and door\_count will be 5 but the question is, WHEN does the variable get incremented? The difference in placing the ++ before or after changes the timing of when the increment happens. Within a single statement line that requires more computing if the ++ is before the name of the variable, the variable gets incremented and then the rest of that line of code gets computed. If the ++ is after the variable, then the entire line of code is executed and only then does the variable get incremented.

The other shorthand which we can use is when we add a particular value to a variable. We could write:

house\_cost = house\_cost + window\_cost;

This is good, and very clear. But there are some additional operators which allow us to shorten this to:

house\_cost += window\_cost;

The += operator combines addition and the assignment, so that the value in house\_cost is increased by window\_cost.

Some other shorthand operators are:

|  |  |
| --- | --- |
| **a += b** | the value in a is replaced by a + b |
| **a -= b** | the value in a is replaced by a – b |
| **a /= b** | the value in a is replaced by a / b |
| **a \*= b** | the value in a is replaced by a \* b |

At a first programming course level, I am not a fan of these shortcuts. I suggest you write the lines of code out without them, and maybe as you enter the 2nd or 3rd programming course, you can adopt these shorthands.

### Variables and Scope

We have been using braces { } to mark code segments for our Main method, our loops, and are if statements. It is useful to think of those braces as putting a moat around that section of code, what we call code blocks. Any variables you define within that moat, can be accessed **inside** of the moat, and they can be used inside of any more moats that are **inside** of that moat, but they cannot be accessed **outside** of the moat where they were defined.

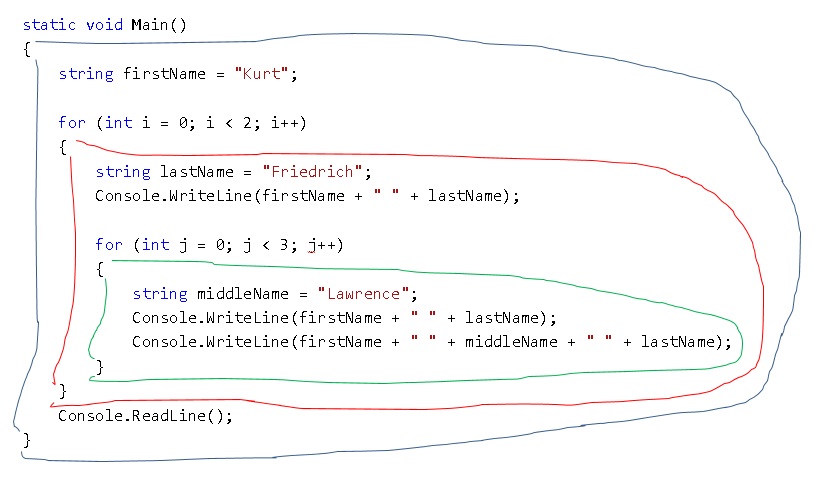


This concept is called scope. What is the scope of a variable? After you define a variable, where can you use that name, and where can’t you? That is what we are talking about. When I come home from work, I often ask “how is Percy?” Anyone in my family will answer me, because the name Percy is “in scope’ in my house.



But if I walk into a grocery store and ask “have you seen Percy here?” they will say “who is Percy?” Percy is not in scope outside of my house.

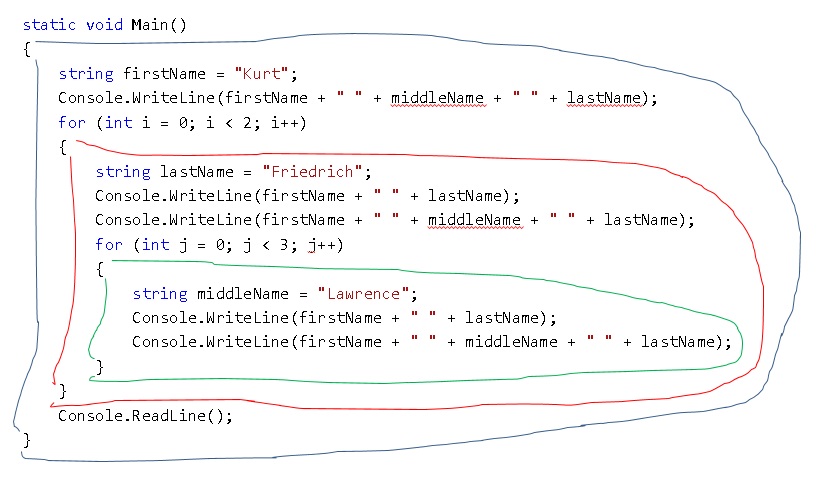
Look at the next code example. I have extended the implied boundaries of the 3 sets of braces { }. This really is what they imply, they are putting a boundary around a code block. So we have a green moat inside of a red moat inside of a blue moat.



This code will work correctly. The inner-most green moat created by the 2nd for loop can write out the variables firstName, middleName, and lastName; they are all in scope. It can also use the variables i and j.

Inside the red moat, created by the 1st for loop, we can write out the variables firstName and lastName; they are all in scope. But we cannot use the variables middleName and j, because they are defined on the other side of an **inner** moat, they are protected from the outside world.

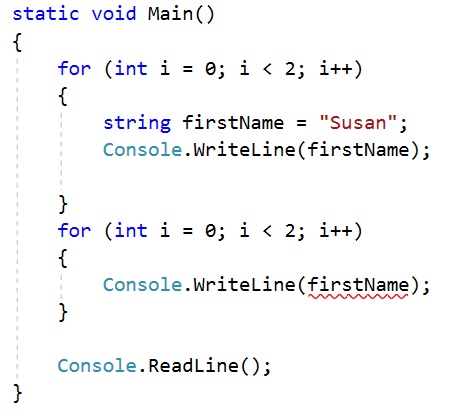
Now look at this modified version.



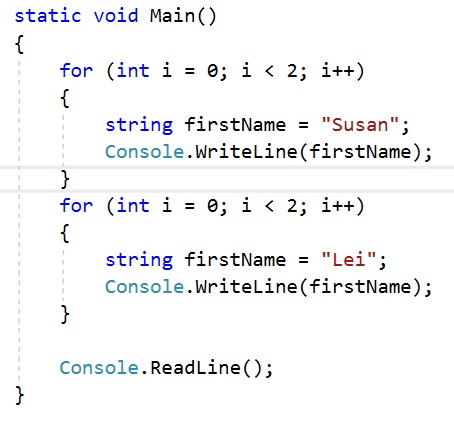
You can see the compiler is not happy (items underlined in red by the compiler). Inside the blue moat, I am trying to use middleName, and lastName, which are protected inside of the red and green moats. Likewise inside the red moat, the compiler will not let me use middleName, but of course I can use lastName because that is defined inside of the red moat.

When you get compiler errors that say a variable name is not known, odds are good you have coded a scope bug.

Below is an example of such a bug. Notice how the variable firstName is underlined in red by the compiler inside the second loop. Each of these two **for loops** have their own code block; moat. firstName is defined inside of the first moat. The second loop is not inside of the first, so it is its own little nation, and it can’t see the variable firstName defined inside another moat.



This version of that code fixes it, but it is fairly dangerous to have two variables with the same name, while not in the same code block, so not in the same scope, but both in the same class. The compiler is happy, but this can lead to coder confusion, especially two years later when someone is working on your code.



### Practice with a Dice Game

Let’s have some fun, and introduce a way of getting random numbers. Many simple games can be coded with what we have covered so far, especially if we can use random numbers. We are going to write a dice program. It will simulate rolling two six-sided dice. If the total of the two is 7 or 11, you win that turn. Otherwise you lose. If you get to 3 wins before you have 3 loses, you win the game.

We will need to keep track of how many games are won, and are lost, and the value of the two dice. So start with four variable declarations.

int gamesWon = 0;

int gamesLost = 0;

int diceOne = 0;

int diceTwo = 0;

Now comes some magic. Just like we have used a .NET library class names Console and some of its methods to read and write to our user, we are going to use another class in that library, it is called Random. It might be a good time to go back and re-read the section on [Class, Object, Method, Property](#_Class,_Object,_Method,) in chapter 1. Whereas the Console class is a static class, so we can call its methods just by using Console.MethodName, the Random class is not marked static. If you want to use the code in that class, you must make a copy of that code, in what is called an object. Then we will use its methods by calling ObjectName.MethodName. So this line of code will make a copy of that code in the Random class so our personal use.

Random myRandom = new Random();

I named my object, my chunk of code that is a copy of the code in the Random class, myRandom. Once we get into the loop of the game, we will use my new object’s method.

Since I want to play the game over and over until I win or lose 3 times, it’s pretty clear I need a loop. I want to stay in the loop until one of those counters gets to 3. A while loop will work nicely.

while (gamesWon < 3 && gamesLost < 3)

{

}

I will be stuck in this loop until one or the other of those variables gets incremented from 0 up to 3.

Now I will code the inside of the loop, the code that we will do over and over. First I will let the user know where they stand and wait for them to continue the game by typing the Enter key.

Console.WriteLine();

Console.WriteLine("You have " + gamesWon + " wins and " + gamesLost + " loses");

Console.WriteLine("Press the Enter key to roll the dice.");

Console.ReadLine();

Now for the game logic. I want to simulate rolling 2 dice. I will do that by asking my random number generator code to give me a random number between 1 and 6 for each dice.

diceOne = myRandom.Next(1, 7);

diceTwo = myRandom.Next(1, 7);

It’s not important that you understand exactly how this works, but I call a method named Next in my object myRandom and I pass in two values to tell it I want a number no lower than 1 and less than 7. That method does its job, we don’t care how, and returns a number to me which I save in the dice variables. That’s all there is to using the .NET library’s Random code. You make a copy as a new object and name it; in my case, I called it myRandom. Then you can call the Next method which you now have a copy of, and give it a lower bound and one more than the upper bound.

Now we tell the user what the dice values roller were;

Console.WriteLine("You rolled a " + diceOne + " and a " + diceTwo);

And now we decide if they won or lost. Based on which, we increment the counter and tell the user.

if (diceOne + diceTwo == 7 || diceOne + diceTwo == 11)

{

gamesWon = gamesWon + 1;

Console.WriteLine("You won that roll!");

}

else

{

gamesLost = gamesLost + 1;

Console.WriteLine("Sorry, you lost that roll");

}

Now if either of those increment instructions pushed them up to a 3, we will fall out of the while loop. Otherwise, we will go do all that code again. If we did fall out, we need to decide if we fell out because the won, or they lost, and tell them.

Console.WriteLine();

if (gamesWon == 3)

{

Console.WriteLine("Congratulations! You won the game!");

}

else

{

Console.WriteLine("Sorry, you lost the game.");

}

Console.ReadLine();

Here is a complete copy of the game:

static void Main()

{

int gamesWon = 0;

int gamesLost = 0;

int diceOne = 0;

int diceTwo = 0;

Random myRandom = new Random();

while (gamesWon < 3 && gamesLost < 3)

{

Console.WriteLine();

Console.WriteLine("You have " + gamesWon + " wins and " + gamesLost + " loses");

Console.WriteLine("Press the Enter key to roll the dice.");

Console.ReadLine();

diceOne = myRandom.Next(1, 7);

diceTwo = myRandom.Next(1, 7);

Console.WriteLine("You rolled a " + diceOne + " and a " + diceTwo);

if (diceOne + diceTwo == 7 || diceOne + diceTwo == 11)

{

gamesWon = gamesWon + 1;

Console.WriteLine("You won that roll!");

}

else

{

gamesLost = gamesLost + 1;

Console.WriteLine("Sorry, you lost that roll");

}

}

Console.WriteLine();

if (gamesWon == 3)

{

Console.WriteLine("Congratulations! You won the game!");

}

else

{

Console.WriteLine("Sorry, you lost the game.");

}

Console.ReadLine();

}

## Check your understanding

#### **Questions 1:**

Write a program that will calculate how much money you will have in the bank after 5 years. Assume you start with $1000, and at the end of each year the bank adds 5% of that day’s balance to your account. Use a “Definite” While loop.

#### **Questions 2:**

Write a similar interest program that will calculate how much money you will have in the bank, however this version will write out your balance at the end of each year. It will then ask if you want to go to the next year or quit. So this program will write out from zero to as many years as the user likes. Assume you start with $1000, and at the end of each year the bank adds 5% of that day’s balance to your account. Use an “Indefinite” While loop.

#### **Questions 3:**

Write a number guessing game using an “Indefinite” While loop. Start by having the computer pick a random number between 1 and 100. Have the user guess the number, if they get it correct, tell them so and tell them how many guesses it took. If they did not get it on the first guess, enter the While loop and have them continue to guess until they get it correct. When the get it correct, you should leave the loop and then execute the existing code you wrote that tells them they were correct and how many turns it took.

#### **Questions 4:**

Think about how to play your Question 3 game to consistently get it in the fewest turns. If you can’t think of a good strategy, investigate the concept of Binary Search on the web.

#### **Questions 5:**

Re-do Question 2 using a Do While loop.

#### **Questions 6:**

Re-do Question 3 using a Do While loop.

#### **Questions 7:**

Re-do Question 1 using a for loop.

#### **Questions 8:**

Using a definite while loop, write a program that counts and writes to the console, the value 10 downward to 1, sort of like a countdown timer for a rocket launch to the ISS. After that works correctly, in the same program, write more code to do the exact same thing, but this time using a for loop structure.

#### **Questions 9:**

Examine this program and see if you can explain how “dog” simultaneously holds three different values, and why when it runs, it outputs:

Dog's name is: Percy

Dog's name is: Percy

Other dog's name is: Rover

Other dog's name is: Fido

The program:

namespace Scope

{

class Program

{

static void Main(string[] args)

{

string dog = "Percy";

for (int i = 0; i < 2; i++)

{

string intro = "Dog's name is: ";

Console.WriteLine(intro + dog);

}

//Console.WriteLine(intro + dog); // why won't this work?

string intro2 = "Other dog's name is: ";

Console.WriteLine(intro2 + Dog2.dog);

Console.WriteLine(intro2 + SomeOther.Dog2.dog);

Console.ReadLine();

}

}

static class Dog2

{

public static string dog = "Rover";

}

}

namespace SomeOther

{

static class Dog2

{

public static string dog = "Fido";

}

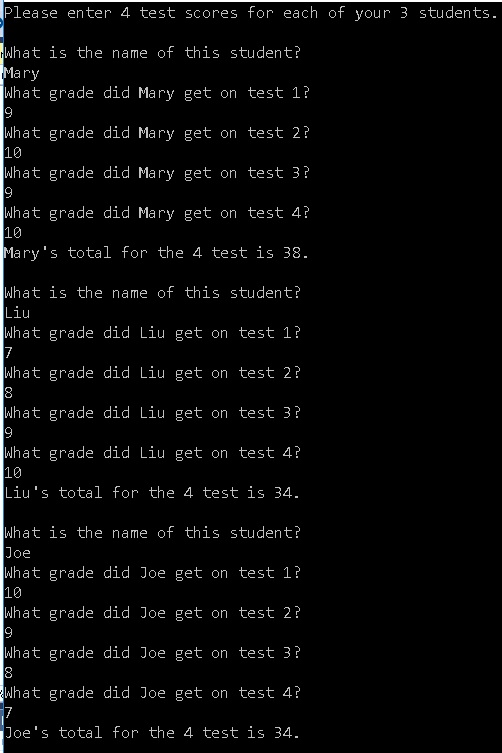
}

#### **Questions 10:**

Using Write a program using a for loop that asks the user to input five integers. Then calculate the average of the five numbers as a double variable and write that answer out to the user. After that code works, wrap all that code inside of another for loop that runs three times, so the user will be asked to input 5 numbers three times, and the program will write out each of the three averages.

#### **Questions 11:**

Write a program using nested loops. It should ask the user to enter grade data for 3 students for each of 4 tests. Each of the 3 times through the outer loop, it should start by asking the user the name for the current student. Also inside of that outer loop, an inner loop should ask for each of 4 grades for that student, after it leaves the inner loop, at the bottom of the outer loop, it should write out the students name and total score for the 4 tests. Here is a sample output.



# Chapter 7 – Writing to the Console and Basic String Usage

We now know almost enough C# to write some very interesting programs, but to communicate with the program’s user, we need more capabilities. We need to understand the basics of dealing with strings and how to write information to, and read from, the console. Some of the previous programs did a bit of these things, but they were not explained in sufficient detail.

You have presumably already acquired skill in manipulating numbers in an algebra course. However with computers, we not only manipulate numbers, but also letters and words as well. An individual letter is called a char (character) and a variable that can hold from zero to many characters is a string variable. Everything in C# is an object, and all objects have methods. Both the char and the string objects come with a very complete set of methods and properties to make it easy to manipulate them. It might be a good idea to go back and review this section now: [Class, Object, Method, Property](#_Class,_Object,_Method,)

### Concatenation

As I mentioned in the chapter on loops, you can glue two or more strings together to create one longer string. The + operator is used for this, and this operation is called concatenation. Examine this code and decide what it will write to the console:

String firstName = "John";

string lastName = "Doe";

string combination = firstName + " " + lastName;

Console.WriteLine("Mr. " + combination);

There is an important subtlety here, slow down and read this carefully. The third line will set the combination variable to be "John Doe"; Notice how I had to concatenate the space in between the two names if that is the desired outcome. When I wrote to the console, I also told the Console WriteLine method to concatenate the literal "Mr. " to my combination variable. **But note**, this concatenation in the 4th line is just an instruction to the Console method, it is not an assignment statement like the third line is. So after these four lines are executed, the console will display

Mr. John Doe

but the value of combination will still be just "John Doe" and not " Mr. John Doe".

### Comparing two strings

If you want to make a decision based on if two strings are the same word or phrase, just like with numbers, we use the same relational operators == or !=

if (firstName == lastName)

{

Console.WriteLine("This person has the same first name and last name!");

}

There are actually three other ways to compare strings using a string object’s methods, Equals(), Compare(), CompareTo() but they are more complicated and have slightly different behaviors. For now, we will use only the == or != .

### Length property

If you need to know how many letters are contained in a string, each string object has a property that will give you that information. See this code.

Console.WriteLine(firstName.Length);

If the firstName variable had been set to be “John”, then the console would write out the number 4. It would make sense to instead use this code.

Console.WriteLine( (firstName.Length).ToString() );

When there are parenthesis, the compilers processes the code from the inside of parenthesis to the outside. So the compiler would first evaluate the property value of firstName.Length, which will be an integer. If we want to treat that value as a string, every integer object has a ToString() method which will convert the integer value 4 to a string value of 4. And then the compiler would be left with

Console.WriteLine("4");

Where that 4 is a string, not an integer. It turns out the Console’s WriteLine method is a very “friendly” method, and very much UNLIKE the rest of the C# language, you can ask it (pass in a variable) to write out just about any kind of data type, and instead of complaining that you can only write strings to the console, it converts whatever you give it to a string automatically. That is why the previous code worked.

Console.WriteLine(firstName.Length);

You are actually giving an integer to the WriteLine method, and it coverts it to a string for you. While this is generally convenient, it is inconsistent with most of the C# language, which always requires you to be consistent with your data types. For example, this code will fail with a compiler error.

string wordLength = firstName.Length;

Here you are telling the compiler that the variable wordLength is a string, and then you are trying to store an integer into it. If you really wanted to do this, then in this, and in almost all other cases, you must overtly convert the one data type to the other. This code is correct.

string wordLength = (firstName.Length).ToString();

### string Substring method

We have shown how use can use concatenation to glue two or more strings into one string, but what if you want to break a string apart into pieces? The string object has a method call SubString that will do this for you. Start with a string variable.

string testString = "waterworld";

Using the SubString method looks like this.

testString.Substring(start at, how many));

Where you supply a value for how many letters from the beginning you want to start at and then how many letters you want. When you call a method, the values you pass into the method are called arguments. So this Substring method is being passed in (given), two arguments, a start at value and a how many value.

Notice the position of each letter in the testString variable. Programming languages almost always start counting things starting at zero instead of one. (If you are like me, you will code a bug in about every third program because you will forget to start at zero some times.)

0 1 2 3 4 5 6 7 8 9

w a t e r w o r l d

Examine these method calls carefully and decide what string would be written to the console.

Console.WriteLine(testString.Substring(0, 1));

// start at letter 0 and give me one letter

Console.WriteLine(testString.Substring(0, 2));

// start at letter 0 and give me 2 letters

Console.WriteLine(testString.Substring(1, 2));

// start at letter 1 and give me 2 letters

Console.WriteLine(testString.Substring(2, 3));

// start at letter 2 and give me 3 letters

Console.WriteLine(testString.Substring(0, testString.Length));

// start at letter 0 and give me 10 letters

Console.WriteLine(testString.Substring(testString.Length - 4, 2));

// start at letter (10-4) and give me 2 letters

### string Contains, EndsWith, and IndexOf methods

These methods do what their name suggests. Using the same string variable,

string testString = "waterworld";

And the same numbering scheme,

0 1 2 3 4 5 6 7 8 9

w a t e r w o r l d

Look at these examples.

Console.WriteLine(testString.Contains("ate"));

// The letters “ate”, in that order, are in the word, so it will write out 🡺 true

Console.WriteLine(testString.EndsWith("ld"));

// The word does end with the letters “ld”, so it will write out 🡺 true

Console.WriteLine(testString.IndexOf(‘r'));

// The letter “r” is found first at location 4, so it will write out 🡺 4

Console.WriteLine(testString.IndexOf("ter"));

// The letters “ter” are found first at location 2, so it will write out 🡺 2

Console.WriteLine(testString.IndexOf('w'));

// The letter “w” is found first at location 0, so it will write out 🡺 0

Console.WriteLine(testString.IndexOf('w', 4));

// The letter “w” is found first at location 5, AFTER starting to look

// at location 4 (this skips over the first “w”) so it will write out 🡺 5

IndexOf() returns a -1 if the char or string is not found. Make sure you always check for this. -1 is commonly used is C# as an indicator when you ask the code to find something and it fails to find it.

Console.WriteLine(testString.IndexOf('b'));

// The letter “b” is NOT found anywhere in the string, so it will write out 🡺 -1

### Better ways of writing output to the Console

By using the above string methods along with concatenation, you can create proper print strings to output to the console to communicate with the user. There are more features of C# that enable more flexible outputs to the console.

#### **Using Placeholders in Print Strings**

Instead of creating a series of strings concatenated with variables to create the text you want to print out, there is a **placeholder** feature which is often neater and easier. A placeholder gets inserted into a string and marks a place where the value of some variable is to be inserted when printed. Consider:

int index = 150 ;

double first = 1234.56789 ;

Console.WriteLine ( "Index: {0} First: {1}", index, first ) ;

This would print out:

Index: 150 First: 1234.56789

The {n} part of the string says “parameter number n, counting from 0”. Consider another example:

string firstName = "John";

string lastName = "Doe";

Console.WriteLine("First name is {0} and last name is {1}.", firstName, lastName);

This would print out:

First name is John and last name is Doe.

Without placeholders, the code would be:

Console.WriteLine("First name is " + firstName + " and last name is " + lastName + ".");

Placeholders are a bit nicer, but in addition, they give you better control of the output.

#### **Adjusting real number precision**

Placeholders can have formatting information added to them:

int index = 150 ;

double first = 1234.56789 ;

Console.WriteLine ( "Index: {0:0} First: {1:0.00}", index, first ) ;

This would print out:

Index: 150 First: 1234.57

The 0 characters is just a marker for one or more digits that are on the left side of the decimal point. The number of zeros you place on the right side indicate how many digits of accuracy you want to appear in the console.

#### **Printing in columns**

Finally I can add a width value to the print layout information. This is very useful if you want to print material in columns:

int day1 = 4;

double reading1 = 1234.56789;

int day2 = 16;

double reading2 = 264.386;

int day3 = 152;

double reading3 = 42264.3862596;

Console.WriteLine("day1: {0,4:0} reading1: {1,10:0.00}", day1, reading1);

Console.WriteLine("day2: {0,4:0} reading2: {1,10:0.00}", day2, reading2);

Console.WriteLine("day3: {0,4:0} reading3: {1,10:0.00}", day3, reading3);

This would produce the output:

day1: 4 reading1: 1234.57

day2: 16 reading2: 264.39

day3: 152 reading3: 42264.39

The integer value is printed in a column 4 characters wide, and the double is printed in a 10 character wide column. At the moment the output is right justified, if I want the numbers left justified I make the width negative:

Console.WriteLine("day1: {0,-4:0} reading1: {1,-10:0.00}", day1, reading1);

Console.WriteLine("day2: {0,-4:0} reading2: {1,-10:0.00}", day2, reading2);

Console.WriteLine("day3: {0,-4:0} reading3: {1,-10:0.00}", day3, reading3);

This would produce the output:

day1: 4 reading1: 1234.57

day2: 16 reading2: 264.39

day3: 152 reading3: 42264.39

Note that this justification would work even if you were printing a string rather than a number, so if you want to print columns of words you can use this technique to do it.

### Dealing with bad user input

If you tell the user to enter a number, let’s say, their age, and they write “eighteen”, or “none of your business” and you take their input string and try and convert it to an int, your code will blow up (throw an exception). It is much better if you handle that situation more gracefully and let the user recover. There is a general way of dealing with issues called “try catch”, but we won’t cover that in this course. There is another more specific way that we will learn now. For most numeric data types, they have a method called TryParse that will tell you if a string can be converted to their data type successfully. Let’s look at the Int32 version.

There are several integer data types, including Int16, Int32, and Int64. They are all good for holding integers, they just use fewer or more bits of memory, and therefore the maximum numbers they can hold vary accordingly. We often code using a shortcut int. int is exactly the same as saying Int32, it is just a shorthand way of stating it. But when you want to use the methods that these integer objects have, you cannot use the shortcut, you must use the full name. So this is the TryParse method for our normal 32 bit integer.

Int32.TryParse(someString, out parsedValue);

This method needs to be given (passed in) two aruguments, someString and parsedValue, with the later argument being preceded by the key word out. out is more complicated than I want to go into now, just use it when doing a TryParse. *( out lets you pass in a variable that has not yet been given a value. You are telling the method “create a value, write it into my parsedValue variable, and when you are done, I’ll be able to use parsedValue to get the value you set into it.”)*

someString is an input string that you want TryParse to “parse” into an integer. Parse is a fancy word for evaluate and do something with, so you are saying, “please Try and Parse this string into an Int32.”

parsedValue is a variable that you must declare but give no value to, before using TryParse, and **IF** the conversion is successful, it will contain the integer version of the someString you gave it.

This method returns a Boolean true or false. True means it was successful in converting the string to an integer, and false says it could not. If the conversion fails, it means your user typed an inappropriate string, and because we are good programmers, we will let them try again and again until they get it right.

bool ok = false;

while (!ok)

{

Console.Write("Please enter an integer value: ");

string userInput = Console.ReadLine();

int parsedValue;

ok = Int32.TryParse(userInput, out parsedValue);

if (ok)

{

Console.WriteLine("Good job entering a valid value, {0}", parsedValue);

}

else

{

Console.WriteLine("nope, that input just won't work!");

}

}

This code uses a while loop that will run forever until the variable ok is changed from false to true. Each time though the loop we get a new string from the user, we call TryParse, and then it returns true or false which is set into our ok variable. If it was bad user input, ok will be set again to false, and the loop will continue. However if it was good input, ok will now be true, and we will leave the loop and know that our user input string is now converted into a good integer that is living in the variable parsedValue .

In a real program, you should always use this sort of logic for getting user input. However because it makes the programs bigger, when I am introducing new topics in the rest of the book I prefer to focus just on the new topic, so I will ashamedly not be using this, and I will blindly assume my users are perfect.

## Check your understanding

#### **Questions 1:**

Write a program that uses a loop to ask the user to enter a new username (one word), and then ask the user to confirm by entering it a second time. If the two names match, the loop should exit. If they do not, give the user a message saying they made a mistake and let them try to enter a username two times again. Stay in the loop until they get it correct. Then follow that loop with a 2nd loop that does the same basic thing, but this time to get their password. When they get that also correct, write out a message that says "Welcome to your new account."

#### **Questions 2:**

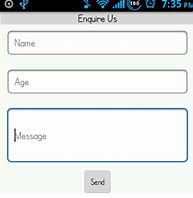
Write a program that asks the user for 3 first names. Then write the 3 names back out in order from fewest number of letters to most number of letter. Use placeholders with formatting to force each output name to take 12 spaces. This can be coded with just 3 **if** statements. For example, if the user entered David, Sue, Andrea, the program would write out:

Sue David Andrea

#### **Questions 3:**

Start a new project in Visual Studio and one at a time paste in the examples given in the chapter where string.Substring(start at, how many) is introduced. Try and predict what will be written out to the console, then run the program and if you were wrong, figure out what was wrong.

#### **Questions 4:**

Write a program that asks the user to enter their age (as an integer), and their height in inches as a double. Make use of the while loop and "TryParse" technique described in the book to verify and force them to enter valid values. Test using some bad values as well as good values.

#### **Questions 5:**

Write a program that accepts one word from the user, and then it writes out that word with the first and last letters swapped. So if they enter **Boston**, it would write out **nostoB**.

#### **Questions 6:**

Write a program that accepts a word, and replaces every vowel with the letter x. For example, if the user types in **football**, the program should write out **fxxbxll**. Assume the vowels are a, e, i, o, and u. Use the .ToLower() method to force the users input word to be all lower case, no matter what they type. (Do not take the easy way by using myString.Replace(oldChar, newChar).)

# Chapter 8 - Arrays

We now know how to create programs that can read values in, calculate results and print them. Our programs can also make decisions based on the values supplied by the user and also repeat actions a given number of times. However, our programs have all used only a small amount of data, and it was reasonable to give a unique name to every single piece of data. There are lots of structures in C# that make it much easier to deal with programs that manipulate large amounts of data. Arrays are one such structure that are used often. It is convenient and compared too many other ways of dealing with large amounts of data, it is quite fast.

Imagine your friend plays on a cricket team. (If you were born in the United States, you might have to look up this game which is played all around the world.) He would like you to write a program for him to help him analyze the performance of his 11 players. When a game of cricket is played each member of the team will score a particular number of points that are called runs. What the customer wants is quite simple, given a set of player scores from a recent game, he wants a list of those scores in ascending order.

The first thing you do is refine the requirements and add some metadata. You discuss sensible ranges (no player can score less than 0 or more than 1000 runs in a game). You draw out a rough version of what the program will accept, and what information it will print out. Finally, you agree on when the software must be finished and how you are going to demonstrate it to verify if meets the requirements. You write all this down and get it signed. With all this settled, it is time to write the actual program. The first thing to do is define how the data is to be stored:

int score1;

int score2;

int score3;

int score4;

int score5;

int score6;

int score7;

int score8;

int score9;

int score10;

int score11;

Now you can start putting the data into each variable. You can ask the user eleven times for a score, and save the results in each of these eleven variables. That’s at least 33 lines of code just to get the 11 variables loaded with the data. All we have to do next is sort them. This will be hard! It can be done, you could create a second set of eleven variables and use a carefully nested loop to copy from the first set to the second. Clearly there has to be a better way of doing this, after all, we know that computers are very good at sorting. Also, if the buyer comes back later and says “oh, we now have 12 people on the team”, it would require many changes to the code.

C# provides us with a structure to hold data called an *array*. Hopefully you used arrays when you took Algebra. An array allows us to declare a whole collection of a particular data type. We then use sub-names, called a *subscript or index,* to indicate which item in the collection we want to use. Each item in an array is called an **element**. Consider the following:

int[] scores = new int[11];

for (int i = 0; i < scores.Length; i = i + 1)

{

Console.Write("Enter score for player {0}: ", i);

scores[i] = Convert.ToInt32(Console.ReadLine());

}

Array.Sort(scores);

for (int i = 0; i < scores.Length; i = i + 1)

{

Console.WriteLine("Score {0}: {1}", i, scores[i]);

}

There are several new things in this code, let’s go through them.

In line 1 we define this new **array** collection. As with any variable, we start by declaring a data type, int[], and then giving it a name; scores.

You can define arrays that hold any one data type, such as double[] scores, string[] scores, etc. You are asking the system to create a collection in which every member will be that data type.

Still in line 1, we then finish the definition of the array by using a reserved word, new and then finish with

new int[11];

That tells the complier to build the array and make room for 11 items which we may store there. A “feature” of arrays is, before you can use them, you must have one of these code lines that defines its size, and you cannot change it later. This makes them a bit of a problem to deal with, but the advantage is how fast the system can read and write to them, as compared to some other collection types that are more flexible, but much slower.

At line 2, now that we have the array, we can use a for loop to ask the user over and over for the score, and save them into the array. for loops and arrays go together like milk and cookies. for loops have an index counter they use to count how many time to execute the loop. That same index works perfectly to index through the array from location 0 to location 10 in the array. Notice, like everything in C#, arrays start at location 0, not 1.

I could have written line 2 as

for (int i = 0; i < 11; i = i + 1)

but by using a property, scores.Length, that every array has, I can get the value of its length. Using the Length property of an array is a much better way of coding it:

for (int i = 0; i < scores.Length; i = i + 1)

Notice that my simple little program has two for loops based on that array. Imagine a much bigger program that has 23 loops. If I wanted to change my program from 11 players to 12, I would have to go all through the code, hoping I could find all 23 instances of a loop that was “hard coded” to an 11, and change them to a 12. But by coding using the Length property, if I change just line 1 from

int[] scores = new int[11];

to

int[] scores = new int[12];

All the code just works. It auto adapts to the size of the array. So when working with arrays and loops, never hard code the loop counter to a number, always use the Length property.

Line 4 takes advantage of the loop counter as it asks the user for each input with a user friendly count. Notice also that it is not using a Console.WriteLine, instead it uses a Console.Write . The former writes out a line of text and then moves down to the next line, whereas the Console.Write just writes out the text but leaves the cursor right at the end of the text. This works nicely when you ask the user a question and you then have them type in the answer on the same line. See the program output below, for example, the first line.

Enter score for player 0: 4

The code wrote out the instructions for the user up to the semicolon, and then the cursor does not move, so that user can see the correlation between the score (4) they are entering and the current count.

Line 5 takes gets the user’s string input, converts it to an integer, and then saves that value in the array. Where does it go in the array? It goes into one of the 11 slots we created, the one whose [subscript number} is currently pointed to be the value of our loop counter i.

Line 7 is using some C# magic, we will cover that a bit later in this chapter, for now, just know that it sorts the array for us.

Lines 8-12 write out the 11 values of the array after it was sorted. Notice how the loop again sequences though each item in the array using the loop index i. This program is not user friendly, as it exposes the array requirement of going from 0 to 10, while the user would much rather see 1 through 11. This can easily be fixed with a few small code changes, but I don’t want to make those 12 lines of code more complex, I want you to be able to focus on how the array was created, how its values got set, and how they were writing out to the console.

Look at this code running on the console:

Enter score for player 0: 4

Enter score for player 1: 5

Enter score for player 2: 2

Enter score for player 3: 0

Enter score for player 4: 7

Enter score for player 5: 7

Enter score for player 6: 3

Enter score for player 7: 0

Enter score for player 8: 2

Enter score for player 9: 5

Enter score for player 10: 3

Score 0: 0

Score 1: 0

Score 2: 2

Score 3: 2

Score 4: 3

Score 5: 3

Score 6: 4

Score 7: 5

Score 8: 5

Score 9: 7

Score 10: 7

#### **The range of an array**

It is important that you never try and access an index for an array to a location in the array that does not exist. In our array

int[] scores = new int[11];

If your code tries to use scores[12] or scores[-3] your program will “blow up” with what’s called an ugly system exception. The exception message will have the words “out of range” in it, telling you that you tried to access an array value that does not exist. Try it yourself and see what I am talking about.

You might think, “Oh, I would never do that”, but it is a common bug. For example, your code might ask the user “which score do you want to see”, and the foolish user my enter a 23 and boom! Your program dies.

#### **Initializing arrays**

There are several choices for how you create an array.

(1) You can be very explicit and use all the available fields. You give it a data type, a name, set the size, and prepopulate the array with values. Note the length and the number of value list items must be the same.

int[] scores = new int[5] {100, 76, 88, 100, 90};

(2) You can leave out the value in the [ ] and the system will use the count of items for that number.

int[] scores = new int[] {100, 76, 88, 100, 90};

(3) The lazy person way, you can leave out the new int[] and the system will assume you want an array of integers with those five values set.

int[] scores = {100, 76, 88, 100, 90};

Of these three ways, number one is the safest, as you are letting the compiler double check your intentions. It is too easy, for example, when entering a list of 20 values to accidentally skip one. If you use the second or third method, the compiler can’t tell if you wanted 20 but accidentally only listed 19. But if you use the first one, specifying [20] and then you only enter 19 values, the compiler will tell you before you try and run your broken code.

(4) You don’t know the values at the time of creating the array, you will fill them in later (as we did in the program just above).

int[] scores = new int[5];

If you use the choice (4), where you do not specify values as you create the array, the system gives a starting value to each element.

* Numeric arrays: all elements are set to 0
* bool arrays: all elements are set to false
* String arrays: all elements are set to null

A null is a special value when it shows up in any C# code. It says to the system, this variable has NEVER had its value set. If you try and read it, you will get a “null value exception”, not a good thing. Note that a null is not the same as a value of zero, and it is not the same as a string that has been given this value

string firstName = "";

In this case firstName is not null, it does have a value that was assigned. The value is an empty string, so if you print it to the console it will write nothing. However an empty string is very different than a null.

#### **Searching an Array Using a Loop**

Often after putting values into an array, you will need to answer the question, is this value in the array? It is easy to answer that question with a for loop.

string[] cities = new string[] { "Redmond", "Bellevue", "Kirkland", "Kent" };

Console.Write("What city are you looking for? ");

string userInput = Console.ReadLine();

bool found = false;

for (int i = 0; i < cities.Length; i = i + 1)

{

if (cities[i] == userInput)

{

found = true;

}

}

if (found)

{

Console.WriteLine("{0} is in the list.", userInput);

}

else

{

Console.WriteLine("{0} is not in the list.", userInput);

}

Line 1 creates and populates the array.

Line 4 creates a “pessimistic” variable found, assuming we will not find the city. As it turns out, this makes the code easier to write.

In lines 5 through 11, we examine, one by one, each name stored in the array, if we never get a match, we exit this loop and our pessimistic bool, found, is still false. On the other hand, if the city name entered did match one of the elements in the array, at line 9 we set found to true.

Lines 12 through 19 write out one true or false, based on the value of found.

### Parallel Arrays

You will see later in this chapter that you can create 2, 3, or n dimensional arrays, which allow you to store many rows and columns of data, much like a spreadsheet. However, there is a problem, all elements in an array must be the same data type. So if you wanted a two dimensional array that stored ten people’s first and last names, as they are both strings, this will work nicely. But if you wanted to store their name and their age, you cannot do that in a single array, as you would be trying to mix strings and integers in the same array.

In this situation, you can use a technique called parallel arrays. They work nicely, but, the compiler does not understand them. They are not part of the C# language. They are merely a programming technique that the coder implements. So you need to set them up carefully to avoid bugs.

The basic idea is that you have two (or more) arrays that are of the **same** length, but are of different data types.

As you add data, you must add data to both arrays at the same time, keeping each index the same. Imagine you are trying to keep track of final grades for 5 students. You want their name and grades to be correlated.

|  |  |  |
| --- | --- | --- |
| **Student name** | **“index”** | **grade** |
| Sneezy | 0 | 84 |
| Sleepy | 1 | 92 |
| Dopey | 2 | 100 |
| Doc | 3 | 44 |
| Bashful | 4 | 87 |

So you create two arrays, both of size 5, one is a string array and the other is an integer array. As you index your way thought the string array entering name values, you use the SAME index to write an integer grade into the integer array. Or in some cases, you know the values and you can create them already synchronized like this:

string[] nameArray = {"Sneezy", "Sleepy", "Dopey", "Doc", "Bashful" };

int[] gradeArray= { 84, 92, 100, 44, 87};

The computer just knows there are two arrays, it is up to the programmer to initialize, maintain, and read from the arrays in a coordinated way. Now see how this code does a search through the nameArray and when it finds the match, it uses the same index i to get the score out of the gradeArray.

Console.Write("Enter a dwarf's name: ");

string dwarfName = Console.ReadLine();

Console.WriteLine();

for (int i = 0; i < nameArray.Length; i++)

{

if (nameArray[i] == dwarfName)

{

Console.WriteLine("{0} scored a {1}.", dwarfName, gradeArray[i]);

break;

}

}

Here is sample output from that program:

Enter a dwarf's name: Doc

Doc scored a 44.

### Using the Sort(), Reverse(), and BinarySearch() Methods

Because arrays are used so extensively, the C# language has methods already available to manipulate them so that everyone doesn’t have to write their own methods. We will look at three of them. Hopefully you are getting use to the idea of methods. So if C# is going to have methods to help us manipulate arrays, it would be appropriate to put them all in the same class definition. Which they did. Now what would be a good name for a class that has many methods for dealing with arrays? How about Array? Well, the good news is, that is an easy class name to remember, when you need a method to do something with an array, always check the Array class. But maybe not so good, students often get the word array (lower case) which is a data structure confused with Array (upper case A), that class with all these nice methods. So stop now, and try and get this to stick in your head. There is a class with a name **A**rray that has methods that you use to manipulate an **a**rray.

#### **Sort**

If you go back and look at our first array program at the start of this chapter, you will remember I sorted the array with one line of code, but I never explained how that worked. Well now I will. This is the line:

Array.Sort(scores);

What we have here is a Sort method that lives in a class named Array, and this method requires one parameter to be passed in, namely, the name of the array you want it to sort (scores). So scores is an array, Sort is a method in the class Array. It’s pretty easy to use. This Sort method will sort any array made up of any one of the normal simple data types, such as int, sting, double, char, etc. Now arrays can also hold custom objects that you define, and in general, Sort cannot sort arrays of these types until you add some special code called an Interface. But that is a much more advanced topic. For now, we will stick with simple data types in our arrays, and these methods all work nicely.

One more thing about the Array class. It is a static class, which means there can only be one, you cannot make many object copies from it, and when you want to reference its methods, you use the name of the class, not the name of some object created from the class (since you cannot make an object from a static class). The reason why the C# people made it static is that the method does not rely on any pre-existing data. All it needs is its code and the data (array) you give it. So there is no reason to make copies, one copy will server everyone. No one needs a custom copy that uses any previously set special data.

#### **Reverse()**

If you understand how to use the Sort method, then understanding how to use the Reverse method is really easy. If you want a method to flip all the values in your array, for example,

int[] niceArray = new int[5] { 1, 2, 3, 4, 5 };

then you just pass the name of your array into the Reverse method, and it will do the job for you.

Array.Reverse(niceArray);

After doing that one line of code, your array values will be flipped to { 5, 4, 3, 2, 1 }

#### **BinarySearch()**

BinarySearch is very useful, but just a bit more complicated to use. You can use it to tell you if an item is in an array and which element it first matches. We did this ourselves with a for loop earlier in this chapter. This method of the Array class will do the job for you. But, it’s not perfect in all cases. To use it all these items must be true:

* Your array items MUST be arranged in ascending order. (You can often use the Sort method we just went over, but there are times when sorting the array is not correct. For example, if the array is part of a set of two parallel arrays, sorting one array breaks the correlation between the two arrays. In that case, you would have to fall back and use the for loop to do the search.)
* Your array should not hold duplicate values, or if it does, then you have to be happy with the BinarySearch method returning ANY ONE of them, and you have to not care which one it returns.
* You are searching for an exact match, such as ID == 3256, and not a range match such as if ID is greater than 3200 and less than 4000.

The form of this method is: Array.Sort(arrayName, value-to-search-for);

It returns an integer which is either -1 if the value-to-search-for was not found, or it returns the index array value where it found a match. Look at this code.

int[] idArray = new int[5] { 2543, 5213, 3937, 2834, 2723 };

Array.Sort(idArray);

int foundIndex = Array.BinarySearch(idArray, 5213);

if (foundIndex < 0)

{

Console.WriteLine("That ID was not found in the array.");

}

else

{

Console.WriteLine("That ID was found at index location: {0}", foundIndex);

}

The output when I run this is:

That ID was found at index location: 4

Notice that before using BinarySearch, I first use Sort to sort the array. Then I ask BinarySearch if the ID of 5213 is in the array. Next I check if it was not found, and then if it was found, I can use that index to, for example, tell someone that was a valid ID, or I can use the index to modify that ID be writing back into that array location. Make sure you always test for the negative return, otherwise subsequent code could crash badly when you try and use that invalid array index of -1.

Note, the reason to use BinarySort instead of your own for loop is not so much the code convenience, but the speed of execution. If you were searching for an employee ID in a company with about 65,000 employees, your loop will cycle 65,000 times. (Okay, on average 32,768 times if you optimize your loop and use a break statement when you find a match.) A binary sort algorithm will find it, worst case in 16 cycles! Much faster.

### Two Dimensional Arrays

As most people are familiar with spreadsheets, two dimensional arrays are generally easy to understand. All we have to do is learn the syntax for reading and writing to them. Imagine we have 3 salespeople and we want to record their sales for each of four quarters in a year. We could use a two dimensional array of doubles, which would be 3 rows by 4 columns.

double[,] Sales = new double[3, 4];

To specify each array location, we use the same [ ] but we include two numbers, which row and which column. Note that in my diagram below, only the yellow part is the array, the white headings are just there for clarity, they would not be there in the code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Quarter 1 | Quarter 2 | Quarter 3 | Quarter 4 |
| Joe | Sales[0,0] | Sales[0,1] | Sales[0,2] | Sales[0,3] |
| Mary | Sales[1,0] | Sales[1,1] | Sales[1,2] | Sales[1,3] |
| Li | Sales[2,0] | Sales[2,1] | Sales[2,2] | Sales[2,3] |

So if we want to set what Mary did in quarter 3, we address the Sales array

Sales[1, 2] = 45000;

And then if we want to write out what Mary sold in quarter 3;

Console.WriteLine("In Q3 Mary sold ${0}", Sales[1,2]);

I am treating the first index as the row index, and the second index as the column index. Note that this is completely arbitrary, as long as you are consistent, you can think of it as [row, column] or [column, row]. Just be very sure to be consistent in your code.

I hope you agree that reading and writing into a two dimensional array is fairly straightforward if you do it one element at a time. Imagine you want to fill in all 12 elements of our array by asking a user for each of the 12 values in order. You need to walk across four elements and do that three times for the rows. If the array were actually 50 rows and 4 columns, you would not want to write 200 sequences of code to fill in the value. Instead, you would want to use a for loop inside of a for loop, often called a nested loop.

double[,] Sales = new double[3, 4];

for (int row = 0; row < Sales.GetLength(0); row++)

{

for (int column = 0; column < Sales.GetLength(1); column++)

{

Console.WriteLine("qtr {0} data for person {1} ", column+1, row+1);

Sales[row, column] = Convert.ToDouble(Console.ReadLine());

}

}

The first loop executes the inside loop three times. Each time the inside loop is executed, it gets four pieces of data and save it in the array. Notice that for the count of how many times to execute each loop, I do not use the array’s .Length property, as that is not valid for a two (or more) dimensional array. Since there are two lengths, you must use a Sales.GetLength(x) method of the array, where x is a o or a 1, depending if you want the row count or column count.

Because arrays start with zero, and not one, for the questions that I wrote to the user to be more clear, I added one to the column and row. This only changes what is printed out, it does not change the value of column or row; there was no = sign, so nothing is changed. So I am mapping the real array indexes to the way humans think of them for display purposes.

Now let’s add code to write the contents of the array back out after the data is filled in. We will use nested loops again.

for (int row = 0; row < Sales.GetLength(0); row++)

{

for (int column = 0; column < Sales.GetLength(1); column++)

{

Console.Write( Sales[row, column]);

Console.Write('\t');

}

Console.WriteLine();

}

For the inside loop I used the Console.Write not the Console.WriteLine because I wanted to write the four numbers across the line. But to keep the numbers from being jammed together, I also write out a tab character between them. Then at the end of each inner loop, when I have written out four numbers, I do a Console.WriteLine so that I drop down a line to write out the next set of four numbers.

Here is the console view if I run both parts of this code, I used particular choices for my sales data to make it easier to see if my code is working correctly.

qtr 1 data for person 1

11

qtr 2 data for person 1

12

qtr 3 data for person 1

13

qtr 4 data for person 1

14

qtr 1 data for person 2

21

qtr 2 data for person 2

22

qtr 3 data for person 2

23

qtr 4 data for person 2

24

qtr 1 data for person 3

31

qtr 2 data for person 3

32

qtr 3 data for person 3

33

qtr 4 data for person 3

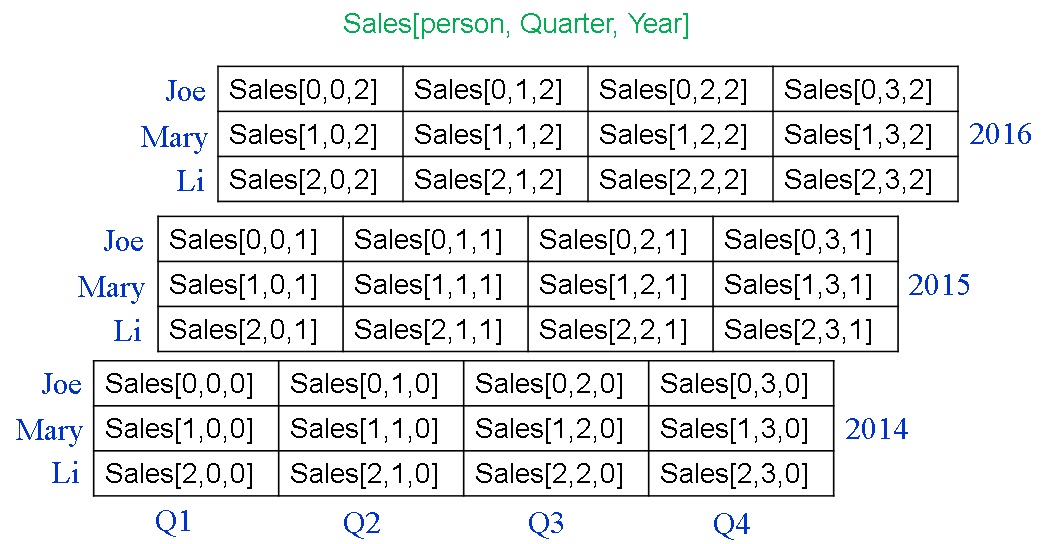
34

11 12 13 14

21 22 23 24

31 32 33 34

Arrays can be any dimension you like, such as [3,3,3} or even [16,23,8,34], but I certainly can’t make a good mental model of such a beast. Here is an example of a three dimensional array that can be understood fairly easily. The first index integer is a pointer to which sales person, the second index integer is a pointer to which quarter, and the third index integer is a pointer to which year.



I suggest that if you can’t make a good mental model of an array that you are considering, you should probably think of another way of holding the data.

### An Array of Arrays (Advanced Concept)

Yes, you can create an array that holds any C# object, including arrays! Imagine you want to record the air temperature every hour on the hour for one week. Here is one way you could do that. See if you can make sense of this:

// create an array of size 7 that holds seven arrays

int[][] WeekofHourlyTempsArray = new int[7][];

// now create an array of size 24 in each of the 7 array slots

WeekofHourlyTempsArray[0] = new int[24]; // an array for Sunday’s 24 temps

WeekofHourlyTempsArray[1] = new int[24];

WeekofHourlyTempsArray[2] = new int[24];

WeekofHourlyTempsArray[3] = new int[24];

WeekofHourlyTempsArray[4] = new int[24];

WeekofHourlyTempsArray[5] = new int[24];

WeekofHourlyTempsArray[6] = new int[24]; // an array for Saturday’s 24 temps

// if day 0 is Sunday, here is code to record the temp

// Tuesday at 1:00am until Tuesday at 5am

WeekofHourlyTempsArray[2][0] = 45; // day 2 => Tues, 0 reading => 1 am

WeekofHourlyTempsArray[2][1] = 44;

WeekofHourlyTempsArray[2][2] = 42;

WeekofHourlyTempsArray[2][3] = 41;

WeekofHourlyTempsArray[2][4] = 43; // day 2 => Tues, 4 reading => 5 am

// etc

Yes, a single 7 by 24 2 dimensional array might be easier to deal with if you control everything, but imagine weather stations are sending you daily reports of 24 hourly temperatures in an array. With an array of arrays, you can just store the arrays as given to you, rather than reading each value in the array sent to you and copying its value over into a n x n array.

### Arrays of the char data type

We covered many string manipulation techniques earlier, but I saved one until after our introduction of arrays. Sometimes the best way to manipulate a string is to turn each character of the string into an array of the char data type. Then you can make changes to one character at a time, using an index pointer to walk through what was a string, but is now an array of characters. Here are three nice C# capabilities to facilitate this.

#### **Turning a string into array of char’s**

string personName = Console.ReadLine();

personName.ToLower();

char[] nameArray = personName.ToArray();

First I used another string object method .ToLower(). This will change every letter in the string to be lower case. (There is also a .ToUpper method to change all characters to upper case.) If you are going to take actions based on the matching of a particular string, it is often a good idea to use one of these. For example, if you ask the user to enter the word “done” to exit a program, they might type Done, DONE, or done and expect all of them to work. You could test with three if statements for all three possibilities, but it’s generally easier to convert their input to all lowers or uppers and then you only have to test for a match of one.

The third line uses another string object method, .ToArray() which convieniently makes a character array out of your string. This has no impact on the string (in this case, personName). Note, this only works if at the top of your program you add another .NET library that is used by this method. Add this librry:

using System.Linq;

Now that I have an array of the caharacters, I can do whatever I like to them, such as swapping all vowels for the letter x, or reversing the entire array. Next you can put the string back together using a method of the string class.

string newString = new string(nameArray);

Here is a working code example

Console.WriteLine("Enter a 4 letter name.");

string personName = Console.ReadLine();

personName.ToLower();

char[] nameArray = personName.ToArray();

char temp = nameArray[0];

nameArray[0] = nameArray[3];

nameArray[3] = temp;

temp = nameArray[1];

nameArray[1] = nameArray[2];

nameArray[2] = temp;

string rebuiltWord = new string(nameArray);

Console.WriteLine("now the name is: {0}", rebuiltWord);

Try and work though the code in your mind and predict what the code does, then paste the code into VS and see if you were correct.

## Check your understanding

#### **Questions 1:**

Write a program that gives away random winnings. A bit like a “one armed bandit”. The player pays $5.00 to play. Then they get to “pull the lever” 5 times. For each turn, based on a new random number between 0 and 9, they win money as defined by this array:

int[] Prize = new int[] { 2, 0, 0, 3, 0, 0, 1, 0, 5, 0};

After each turn, tell them how much they won, and keep a running total. When the 5 turns are over, show them their total winnings.

#### **Questions 2:**

Write a program that uses a loop and asks a user to enter 5 numbers and saves them as decimals in a decimal array. Using a second loop, walk through the loop and add the values, divide the sum by the length of the array (using that property, don’t hardcode the 5) to calculate the average, and then tell the user the average of their 5 numbers. (Don’t do it all in one loop, which is more efficient, but not as good for learning arrays.)

#### **Questions 3:**

Write a “speed dial” application. It will store 10 phone numbers in a Int64 (same as long) array. The code should build a loop. In the loop the user has 3 choices,

* quit the program,
* add a new phone number at a particular digit (0 – 9),
* or enter a single digit (from 0 to 9) and the application will display the number stored there ( or the word empty if that one has not gotten a value). The phone number will not have spaces or –‘s, so 4351239876 would be an example of how it is entered and how it is displayed.

#### **Questions 4:**

Write a program that finds names stored in an array. First create a string array:

string[] nameArray = { "Sneezy", "Sleepy", "Dopey", "Doc", "Bashful" };

Now ask the user to enter a name. Using a loop, walk through the array and see if that name is in the array. If it is, tell the user it is in the array, and tell them the index where it is. If it is not, tell them that.

#### **Questions 5:**

Write a program to try out Array.Sort, Array.Reverse, and Array.BinarySearch

First create a string array:

string[] nameArray = { "Sneezy", "Sleepy", "Dopey", "Doc", "Bashful" };

Now write that to the console using the string.Join(separator, arrayName) method.

Console.WriteLine(string.Join(" \* ", nameArray)); // nice way to write out a short array

* Now sort the array, and write it out again.
* Then reverse the order, and write it out again.
* Sort it again so we can try BinarySearch.
* Now use BinarySearch to see if “Dopey” is in the array. If it is, write out the array position it is at. If it is not, write that out. Do the same thing again, searching for “Doxey”.

#### **Questions 6:**

Write a program “Guess the mystery six letter word” game. We could seed our game with 100 or more words, but we’ll start with just 10.

string[] wordArray = new string[10] { "helmet", "wizard", "realty", "shovel", "summer", "reason", "exotic", "tiring", "easily", "genius" };

At the start of the game, pick one of the 10 at random to be the mystery word. Now create a string array of size 20, which will start as empty, but where we will keep track of incorrect letter guesses so that the player can see letters they have already guessed. Create another string array of six where we will keep track of good guesses.

string[] lettersNotFound = new string[20];

string[] lettersFound = new string[6] {" \_ ", " \_ ", " \_ ", " \_ ", " \_ ", " \_ "};

Next create the main game loop, in which each time through, the player will get to guess one more letter, and also take a guess at the word. Each time through the loop:

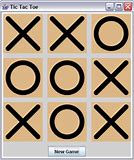
* Write out both the current state of the letters found, and the state of the incorrect letters guesses. Use the handy way of writing out an array as one line; Console.WriteLine(string.Join(" ", array));
* Ask them to guess one more letter.
  + If they pick a letter that is in the mystery word, tell them that letter was found, and again output the updated array of letters found. If the letter occurs more than once in the word, it should show just 2 of those locations, you do not need a loop to find a 3rd, 4th etc.
  + If the letter was not in the word, tell them so.
* Now let them take a guess at the mystery word. If they get it, exit the loop and congratulate them, telling them how many guesses it took. Otherwise, the loop will cycle again, until they get it correct.

You will probably want to use both:

stringVaraible.IndexOf(letterGuess)

stringVaraible.IndexOf(letterGuess, startAt)

#### **Questions 7:**

Write a program that plays X’s And O’s. Create a 3 by 3 char array named gameBoard. Using a nested loop, set the value of each element to be a dash ( ‘-‘). Now write the board out by making a call to DrawBoard(gameBoard); and then paste this code into your project, right below the closing } at the end of your Main method.

public static void DrawBoard(char[,] gameBoard)

{

for (int colPointer = 0; colPointer < gameBoard.GetLength(0); colPointer++)

{

for (int rowPointer = 0; rowPointer < gameBoard.GetLength(1); rowPointer++)

{

Console.Write(gameBoard[colPointer, rowPointer] + " ");

}

Console.WriteLine();

}

}

Next, using the Random class, randomly pick the user to be either X or O and tell them which one they are. Now build a loop that executes 9 times. We will loop to pick an X or O for each spot in the gameboard until the game is over.

At the top of this loop, tell the user to click the Enter key to make the next move, or to type “q” if they can see there is a winner and the game is over. If they type a q, exit the loop using a break command.

Otherwise, build a loop that repeatedly randomly picks one of the 9 array locations until it finds one that is not yet and X or O. When it finds one, it marks it with an X or O, whichever is the opposite of what it used the last time (so it alternates back and forth, each cycle of the game). When it finds such an empty spot, after writing the new value there, it should break out of this inner loop. At the bottom of this outer loop, which is where you get to when you exit the inner loop, again make a call to DrawBoard(gameBoard); Your game then goes back to the top of the outer loop, where the user either quits as they see they have won or lost, or it goes back into the inner loop and writes another location.

#### **Questions 8:**

Write an encryption program that will encrypt a sentence and also de-crypt it. (Do not use any punctuation symbols, just letters.)

* Ask the user to enter a sentence.
* Split the sentence into words using the string Split(' ') method.
* Create a loop to make changes to each word in the newly created array.
  + Inside that loop, create a char[ ] from the current word using SomeArray[i].ToArray().
  + Then create another loop inside the outer loop, that takes each char in that char[ ] of the current word and adds one to each array element if that array element has an even index, or subtracts one if it is odd.
* After you leave that inner loop, make a new string variable out of the modified char [ ] and put that word back into the sentence word array where it came from. This code line might help

string temp = new String(charArray);

* Then when you leave the outer loop, convert the array of words back to a sentence and then write it out to the console.

use string someString = string.Join(" ", someArray);

Next, reverse all that logic and write the code to de-code your sentence back to its original form and write it out, verifying it matches the original sentence.

======================================================================================

**Hint1**: To add one to a char variable, cast the char variable to an int, add one, and then cast all that back to a char. Something like this:

charArray[j] = (char)(((int)charArray[j]) + 1);

**Here are two examples:**

enter a sentence:

The Quick Brown Fox jumped

The Quick Brown Fox jumped

Ugf Rtjbl Cqpvo Gny ktnofc

The Quick Brown Fox jumped

enter a sentence:

abcd efgh ijkl

abcd efgh ijkl

badc fehg jilk

abcd efgh ijkl

#### **Questions 9:** Advanced Concept!!

Write a program for a child’s soccer team with only 5 players. It should use 3 parallel arrays, one of which will be an array of arrays! The program will have one string array with 5 team members’ names. It will have a 2nd parallel int array that will hold each team members’ total goal count for 4 games. A third parallel array will be an array of int arrays; that means each location in this array will be an int array to hold the count of game goals for one player in each of 4 games.

Start with this array of 5 names:

string[] nameArray = { "Jafar", "Jeremy", "Liu", "Chipo", "Will" };

Now define a parallel array where each element is itself an int array of size 4. Now build a nested loop that tells the user which player they are working on, and asks the user for 4 goal counts for each of the 5 players. Then, write a nested loop that calculates the total number of goals each player got, and writes that into the 2nd array. Lastly, using a loop, write out the 5 names and their total goal counts.

# Chapter 9 – Adding Methods

#### **What you already know**

Hopefully you have seen classes, methods, objects, and properties in the previous chapters enough that you are starting to understand them. It’s probably still a good idea and go back and read: [Class, Object, Method, Property](#_Class,_Object,_Method,) in chapter 1.

Organizing your code into sub sections using classes, and then inside of classes with methods, is a very good idea, but in this book, we are not going to create classes. All of our code will be in the one class Program that VS auto-creates for us when we start a new console project. Up until this chapter, not only was all our code in that one class, it was all in one method, Main(). In this chapter we will learn how to break our code into logical, re-usable sub sections by writing and calling our own methods.

The system will always look for a static void Main() method as the “front door” of your code. It will always start there. And because there can only be one starting point, that method is marked static. For reasons that are beyond the scope of our current knowledge of classes, all the methods what we write insides of this Program class will also all be marked static. (*Someday you can come back and read this, but I suggest you not read this now: By marking them static, the code in Main can call these methods by just using the method name. If they were not static, you would have to instantiate an instance of the Program class, as an object, and then use the name of that object to reference the method*.)

#### 

You have already been using methods that are in classes in the .NET libraries. We have used the Console class’s WriteLine and ReadLine methods to give us a way of displaying text and reading input from the user.

Let’s look more carefully at how we used them to demonstrate how much you already know about methods.

string outputString = "write this string to the console window";

Console.WriteLine(outputString);

Or more generally,

ClassName.MethodName(argument);

In this case you called a method named WriteLine, but where is this method? You have to say in which class does this method live. In this case, we want the WriteLine method from the class Console. ALL methods end with a set of parenthesis ( ). That is because you can pass information from the code you are in, over to the method code you are calling, using what are called arguments. Some methods do not need any extra data to do their job, so they have no arguments. However, they still end with the empty ( ). Our WriteLine method wants just one argument, and it must be a string data type. So in the example above, the argument is the string outputString.

Now let’s look at ReadLine.

string userInput = Console.ReadLine();

Again we have a method and its name is ReadLine and it is defined in a class name Console. Notice, as all methods do, it ends with a set of parenthesis, but as the ReadLine code does not need any information from you to do its job, there are no arguments inside the ( ). But there is another difference between the structure of WriteLine and ReadLine. The ReadLine method **gives you back** a piece of information. When a method does this, it is called “returning” a value. Since the ReadLine method reads letters typed on the keyboard, it will return a value of type string. We don’t always have to use a return value. This code is acceptable;

Console.ReadLine();

In this case, your program will stop, and wait for the user to type something, and then gather up the letters they typed. But because we did not code a place to save the answer, ReadLine will just through the string away. The more general point is, some methods do something, but give you no data back, while other methods do give you back one piece of information. If they do return something, you must put the method call on the right side of an = sign, and set some variable to save the returned value on the left side of the = sign. This is what we did just above. We will save the returned value from the method call in a string variable named userInput.

#### **Calling methods of classes or methods of objects**

I need to note that we are calling the method by using a class name. That is because the Console class is static, and therefore you can call its methods using the name of the class. If we were to use some object, created from some class that was not static, then we would call the method using the objectname.methodname(). Remember from chapter 6 where we used the Substring method?

string testString = "waterworld";

string newString = testString.Substring(start at, how many);

In this example, testString is an object, created from the class string, and so when we call a method of an object, we use the object’s name: testString.Substring(start at, how many)

Notice also that the Substring method wants two pieces of information to do its job, it wants two arguments. It wants an integer to say at what character position in the word you want to start at, and another integer to say how many letters you want to get back.

### Why methods are good

There are several reasons. While you are writing your code in the Main method, if you have to do a sequence of code statements two or more times at various places, it makes good sense to pull that sequence of code out and put it in its own method. Then you can just simply call that method with one line of code. A good example is if your code must ask the user to enter several pieces of information. Back in chapter 6 we went over a code segment that uses the TryParse method in a while loop to do this nicely. But it was quite a few lines of code. You would not want to repeat all those 16 lines of code 3, 4 or 10 times. Instead, it makes sense to put that code in a method, and then just make those 3, 4, or 10 single line calls to the method.

By doing this, your Main code stream will be much shorter and more clear for someone else to understand. A one line method call

string firstName = GetUserInput();

is very clear what the code is doing; it is getting input from the user. Had I coded in that while loop with 16 lines of code, and then again when I wanted the lastName, and again when I wanted the street name, etc, the code flow in Main would be very hard to follow.

Code reusability is almost always a goal in software engineering. By creating a well-defined method that does exactly one thing, that method can be coded once, debugged once, and then re-used over and over again. Saving space, avoiding bugs, and making the code clear. Not in this course, but later you will learn to put your methods in classes and then put them in your own libraries, just like the .NET library. Then if you are working with a team of people, not only will you re-use your method, all your teammates will as well. And you will take advantage of the methods they write as well.

### The mechanics of writing methods

This section is a reference section, until this becomes second nature to you, come back here for help when you need to write a method. To define a new method it takes 4 steps:

1. Write the method declaration (often called a signature). This is where you name the method, declare what inputs you need to have passed in, and what you will return to whoever called it.
2. Start a code block with the opening **{**
3. Write the code, called a method body, that does the job you wanted the method to do
   1. Optionally, return some value. Methods do not have to return anything, it’s your choice.
4. Close the code block with a closing **}**

Steps 2 and 4 are very easy, just one character. Step 3 can be very simple or tremendously intricate, it all depends on what you want the method to do. Step 1 is very formulaic. This is what we will cover next.

#### **Correct method declarations**

There are (at least) 5 parts in a method declaration, often called a signature. Let’s start with an example.

public static void DisplayTotal(double Amt, double taxRate)

[1] The first item is a keyword that declares the “accessibility” of your method. For the scope of this book, that can be either the words public or private. If you mark your method public you are saying that code in other methods in other classes outside of the class this method is in, are allowed to call and use your method. If instead, you mark it private, you are saying the only code that can call your method must be in other methods in the same class. For this book, since all are code is in one class, Program, it won’t matter which way you declare your methods. But the general rule is, always mark them private and only if you overtly decide you want to share them with other classes, then change them to public. There are other accessibility keywords beyond these two, but they are beyond the scope of this book.

[2] This is our static keyword. It is one of several modifiers that can place restrictions or conditions on your method. As stated above, in this book, all our methods will be marked static to make them easier to call from our static Main method.

[3] Next comes a data type keyword such as string, int, double[], or the word void. This seems to be a hard concept as I see many students make mistakes in this areas. Each method declaration **MUST** say what it will return to the caller. You do that by stating one and only one variable type, which can be any variable type. If, for example, you say double[], then somewhere in your code body of the method, you must in fact create an array of type double, and return it to the caller. We will see how to do this later. The important point here is, you must indicate what type of value you are going to return, and then the code must do that. The key word void says “I am not returning anything.” Which is fine, as sometimes, such as with Console.WriteLine(), methods do not return anything. But you don’t just not return something, you must mark your method signature with that void keyword to make it clear you do not intent to return anything.

[4] Next is the name of the method. You can pick any legal identifier name, but by convention, Class names and method names should start with a capital letter, and if the name is comprised of several words, each word should be capitalized. Method names should be descriptive, not obtuse. So good names could be GetTheCurrentTime, CalculateInterest, WriteDataToFile. Whereas writedatatofile or xxxxx are acceptable as far as the compiler is concerned, but they are just bad practice. Write Data To File will not be accepted by the compiler as it is not a legal identifier name as it has spaces in it.

[5] Lastly comes a list of pieces of data the method wants to be passed into it to do its job. Immediately after the method name comes a set of parentheses, always! Inside of the parenthesis, you list each and every piece of data you need to complete the goal of this method. If you don’t need any information, you leave the parenthesis empty. Otherwise you make a list of pieces of data, these are called parameters. You give a local-to-this method name for each piece of data, and you say what type of data it must be. In the declaration I wrote just above:

public static void DisplayTotal (double Amt, double taxRate)

It is saying that in order to do its job (show total including sales tax), it needs two piece of data, the amount of the sale and the sales tax rate. These are both doubles, and those keywords must be there.

public static void DisplayTotal (Amt, taxRate)

Is not going to work. You are effectively creating two new variables here, Amt and taxRate. In C#, every new variable has to be labeled to tell the compiler what kind of data is allowed to be stored there.

So that’s the end of how you define a method. We will look at several examples to hopefully make it more clear.

#### **Calling a method**

The person who writes the method defines how it must be used. The person who wants to use the method has no choice but to adhere to the method declaration (signature). So if we wanted to call the DisplayTotal method we just looked at, it would be something like;

Progam.DisplayTotal (299.99, .09);

In this case we are passing in literal values, not variable names. This would work just fine. We have the name of the method, the required parenthesis, and 2 double values.

We could also call the method using variable names instead;

double purchaseAmount = 129.24;

double localTaxRate = .085;

Progam.DisplayTotal(purchaseAmount, localTaxRate);

It turns out that when you are in a class method, in our case, in Main, and you want to call another method in the same class, you do not need to prepend the class name to the method call. So our two calls would work fine as;

DisplayTotal(299.99, .09);

DisplayTotal(purchaseAmount, localTaxRate);

The Progam. is not needed. Whereas if we want to call the ReadLine() method, it is not in the same Program class, so I must add the class name in front; Console.ReadLine();

#### **One person’s arguments are another person’s parameters**

Look at the method definition and then right after that, a correct calling statement.

public static void DisplayTotal(double Amt, double taxRate)

DisplayTotal(purchaseAmount, localTaxRate);

When you are calling a method, you pass in values. These are generally local variables that you have created and they have names that make sense in your Main method. However in the context of the actual method you are calling, what were arguments are now called parameters. Also notice that the method gets to pick its own name for these variables. What is really happing is something like this.

Code in the Main method makes a call:

double purchaseAmount = 129.24;

double localTaxRate = .085;

DisplayTotal(purchaseAmount, localTaxRate);

But before the code in the DisplayTotal starts to execute, on your behalf, the system does this:

Amt = purchaseAmount;

taxRate = localTaxRate;

So the method has its own variables (parameters) but they have been given the values selected by the coder in the Main method when they passed in its variables (arguments).

This is a bit confusing and a source of many problems for new students. Please read this section over until you are sure you understand it.

Also, I am sorry to say, many people, web article authors, and especially me, confuse these two words all the time. They will use the words parameter and argument interchangeably. While this is definitely incorrect, it is generally clear from the context if they are talking about values being passed into, or instead, the values received by, a method.

#### **Some examples of methods**

[1] Method with **no** return value, and takes in **no** parameters

public static void DisplaySomeText()

{

Console.WriteLine("Always the same text.");

}

Calling this method

DisplaySomeText();

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[2] Method with **no** return value, and takes in **one** parameter

public static void DisplayTotal(double Amt)

{

double total = Amt + (.09 \* Amt);

Console.WriteLine("The total bill is {0:c}", total);

}

Note, adding the c here, {0:c} , makes the output formatted as currency, so the output looks like:

The total bill is $463.25

Calling this method

DisplayTotal(425.00);

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[3] Method with **no** return value, and takes in **two** parameters

public static void DisplayTotal(double Amt, double taxRate)

{

double total = Amt + (taxRate \* Amt);

Console.WriteLine("The total bill is {0:c}", total);

}

Calling this method

DisplayTotal(425.00, .09);

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[4] Method that **does** returns a value, and takes in **two** parameters. In this example, I show the full code for calling the method, using the return value from the method call, and the method definition.

static void Main()

{

double purchaseAmount = 425.00;

double localTaxRate = .09;

Console.WriteLine("The total bill is {0:c}", CalculateTotal(purchaseAmount, localTaxRate));

// notice you can call a method (CalculateTotal) as an arguement to anothor method

Console.ReadLine(); // all my Main method end with this, it keeps the window open

}

public static double CalculateTotal(double Amt, double taxRate)

{

double total = Amt + (taxRate \* Amt);

return total;

}

A few things to note specifically in that example. The CalculateTotal method returns a value to the calling code in Main. Main code uses that value and writes it to the console. It could have been coded like this instead.

static void Main()

{

double purchaseAmount = 425.00;

double localTaxRate = .09;

double answer = CalculateTotal(purchaseAmount, localTaxRate);

Console.WriteLine("The total bill is {0:c}", answer);

Console.ReadLine();

}

public static double CalculateTotal(double Amt, double taxRate)

{

double total = Amt + (taxRate \* Amt);

return total;

}

In this version, I created a variable answer and set it to hold the return value form the CalculateTotal method call. And then I use that answer variable in the Console.WriteLine statement. If there was more code in Main that used the answer value then this version would be better. But if all you were going to do was get the answer and write it out, there is no reason to create the answer variable. And the first code example would make more sense.

You will need to get comfortable with this nesting of method calls inside of other method calls, as it is done very often. Nested method calls are similar to complicated arithmetic statements. With math code, the compiler starts inside the deepest parenthesis level and works its way out to evaluate the entire line of code. The same thing is done with methods.

string stringAnswer = ( Class1.Method1( Class2.Method2 \* Class3.Method3) ).ToString();

In this pseudo code example, the compiler would first do the code in Class2.Method2 and then the code in Class3.Method3 and then multiple the resulting returns values from those two method calls. Then the resulting value from that multiplication would be used as the argument that is passed into Class1.Method1. Class1.Method1 would do something with that passed in number (maybe round it off to the nearest 10th) and since the result of the Class1.Method1 returns a number, and since a number is an object, it has a ToString method, which we use so that we can write the final number, converted to a string, into our stringAnswer variable.

Don’t skip over that last paragraph. I know it’s a hard one, but you must work it out in your mind, since the concept is important and heavily used.

Here is a full working code sample. *(Personal preference: I like to name my parameter values with a starting p, just so as I’m looking through the method code it reminds me where that variable came from, it was* ***p****assed in as a* ***p****arameter”.)*

static void Main()

{

double length = 10.0;

double width = 5.0;

string stringCost = CalcCost(CalcPerimeter(length, width), CalcArea(length, width)).ToString();

Console.WriteLine("The cost is calculated at $2.50 per linear foot and $1.25 per square foot");

Console.WriteLine("The total cost is $" + stringCost);

Console.ReadLine();

}

private static double CalcPerimeter(double pLength, double pWidth)

{

return 2 \* (pLength + pWidth);

}

private static double CalcArea(double pLength, double pWidth)

{

return (pLength \* pWidth);

}

private static double CalcCost(double pPerimeter, double pArea)

{

double doubleCost = (pPerimeter \* 2.50) + (pArea \* 1.25);

return doubleCost;

}

The output of running this is;

The cost is calculated at $2.50 per linear foot and $1.25 per square foot

The total cost is $137.5

Try and work through this code. This line;

string stringCost = CalcCost(CalcPerimeter(length, width), CalcArea(length, width)).ToString();

is complicated. It is a method call ToString, on the return value from the CalcCost method call, and that method call, contains two arguments, both of which are method calls! In English, it is something like …

Dear CalcCost, please run your code by first calling CalcPerimeter and CalcArea to get the values for your arguments, and when you are done, hand your answer to the ToSting method so that it is converted from a double to a string.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#### **Argument List, Parameter List must match**

Methods others or you write can take any number of parameters. But anyone wanting to use a method must strictly adhere to the methods declaration (signature). This means when you call a method, your argument list must have the same number of arguments, in the same order, and of the same data type as the parameter list in that method definition. And if the method returns a value, and you want to save that returned value, your variable to hold it must be of the same type as specified in the method definition. For example, to use this method;

private static double ConvertTemperature(double pInputTemp, string PFarOrCent)

Your call must pass in a literal double or a variable of type double, followed by a literal sting or a variable of type string, in that order. It cannot leave one or the other out and only pass in one, nor can it pass in three. To mis-quote the film [*Monty Python and the Holy Grail*](https://en.wikipedia.org/wiki/Monty_Python_and_the_Holy_Grail) *.*”then shalt thou count to two, no more, no less. Two shall be the number thou shalt count, and the number of the counting shall be two. Three shalt thou not count, neither count thou one, excepting that thou then proceed to two. Four is right out. Once the number two, being the second number, be reached, then lobbest thou thy Holy Hand Grenade of Antioch towards thy foe, who being naughty in My sight, shall snuff it.”

Here are five examples of code calling that method above, four are mistakes, only one is correct. Try to identify the mistake in each of the four that are wrong.

double answer = ConvertTemperature(98.2)

double answer = ConvertTemperature(98.2, 23)

double answer = ConvertTemperature((98.2,"F");

double answer = ConvertTemperature(("98.2","F");

string answer = ConvertTemperature((98.2,"F");

C# allows for two things that I am not going to cover, but you should be aware they exist. First, you can name the arguments in a method call, and then they do not have to be in the exact order. Second, you can provide pre-set values for parameters in the method definition, and then the user can leave them out of the method call. My personal belief is these “features” make the language more complicated for a very small gain, so I never use them.

### Implementation Hiding

A major advantage of proper object-oriented programming is called “implementation hiding”. The idea is that when you write a method, other people can write code that now depend on your method. However, the **only** dependency they can take on your method is that the method signature does not change. The method signature (declaration) tells users how to interact with your method. The person who owns the code for the method is free to continue to make improvements to the code inside of the method. Maybe they can make it faster, or more secure. Other people using the method should not look at the code inside the method and take any dependency on how the code works (meaning, expect it to work a particular way). Because large complex programs are written using hundreds of methods by many people on a team, each team member can work effectively in isolation. Because of this principal of implementation hiding, each method owner can improve their code and make bug fixes without fear of breaking someone else’s code, as long as they do not change the message signature.

You have been using Microsoft’s Console.WriteLine all through this book, Microsoft could release new versions of the .NET library (which they do) and you do not need to worry if those changes in Console.WriteLine will introduce new bugs into your code. All we have to care about is how we call it, and they will not change that.

### Method Overloading

Method overloading is a name given to a capability of the C# compiler. You can create more than one method within the **same class** using the **exact same name**. You “overload” the name. BUT, each method declaration of this set of same-name methods MUST have different declaration signatures. Look at these 4 method declarations:

public static void CalculateTotal(double Amt)

{

// fantastic code goes here

}

public static void CalculateTotal(double Amt, double taxRate)

{

// fantastic code goes here

}

public static void CalculateTotal(double Amt, int taxRate) {

// fantastic code goes here

}

public static void CalculateTotal(double Amt, double taxRate, string state) {

// fantastic code goes here

}

Notice they all have the same name. It will be acceptable to the compiler to have all 4 of these methods defined in the same class, because even though they have the same name, their declaration signatures are different. They either have a different **number** or **type** of parameters. When the user of these methods make the method call, based on what arguments they specify, the compiler is smart enough to link them up to the correct method.

### Parameter Passing By Value and By Reference

There is one last complication that you must understand with methods. It’s a bigger concept than I want to cover in this course, but we will cover the critical aspect that applies to the code we will write. All variable types are not quite the same.

#### **Value Types**

Some, such as int, string, double, etc, which are often called the simple data types, are called “by value” types. When you pass in to a method, as an argument, one of these “value types”, you are not giving the method access to your variables. You are giving that method an at-this-instant value of your variable which is used to initialize the parameter list. Look at this code.

static void Main()

{

int valueInMain = 4;

int returnFromMethodValue = 0;

returnFromMethodValue = AddOne(valueInMain);

}

private static int AddOne(int pParameterValue)

{

pParameterValue = pParameterValue + 1;

return pParameterValue;

}

In the Main method we start with the variable valueInMain set to 4. Then we use that variable to pass into the AddOne method. The method does not have access to the variable valueInMain, it cannot change its value. Its local variable pParameterValue, has been set to the value that valueInMain was when the method was called, so for an instant they are both = 4. But they are independent. We passed in the VALUE specified in the argument, we did not pass in the actual variable. When the AddOne method adds one to its pParameterValue variable, it has no effect on Main’s valueInMain variable. valueInMain is still 4 while pParameterValue is now 5. In Main, my other variable returnFromMethodValue, will be set to 5 by the return of the pParameterValue variable.

#### **Reference Types**

Reference types work differently. The only reference type we have covered are arrays. Any array is a reference type. Later, when you learn to write class definitions and then then create object copies from them, you will see that objects are also reference types. You can tell which variable are value and which are reference types fairly easily. When you create a new value type variable, you do not use the key word new. When you create a reference type, you do use the new keyword. We did this for all arrays, and when we created our myRandom object from the Random class, we also used the new word.

int valueInMain = 4; // the word new does not appear

string firstName = "George";

int[] manyValuesArray = **new** int[10]; // the word new does appear

string[] nameArray = **new** string[14];

Random myRandom = new Random();

Here is the key point, when you pass an array into a method, you **really are giving** that method your array. You are not giving the method a copy of the current values, it now has the array. Any changes it makes to the array does change the values of the array in Main that you passed in. Look at this code.

static void Main()

{

int[] manyValuesArray = new int[3] { 1, 2, 3 };

DoubleTheArray(manyValuesArray);

Console.WriteLine(manyValuesArray[2]);

Console.ReadLine();

}

private static void DoubleTheArray(int[] pManyValuesArray)

{

for (int i = 0; i < pManyValuesArray.Length; i++)

{

pManyValuesArray[i] = pManyValuesArray[i] \* 2;

}

}

In this case, since manyValuesArray is an array, which is a “by reference” type, we are giving the method a reference to our manyValuesArray array, basically, pManyValuesArray is now a 2nd name. Our one array has two names, manyValuesArray and pManyValuesArray. So when the method doubles all the values of its array named pManyValuesArray, it is also doubling the values of the Main manyValuesArray array, since they are just two names for the same array. Notice that the method did not return the array; there is no need to as Main already has it, just with a different name. So then what will this line;

Console.WriteLine(manyValuesArray[2]);

write to the console?

A bit complicated. Just remember this, when you pass in simple data types to a method, the method has the information to use, but it cannot change the values of the variables you passed in. If you pass in an array, anything you do to that array is being done to the array you passed in (since there really only is one array.)

Maybe this will help. If you have a book, and you want to allow your friend to use it, you could

* Make a copy of your book, and give them a copy. If your friend takes a yellow highlighter pen and makes marks all through the book, YOUR book is fine, and does not have yellow marks. This is passing by “value” as we do with int, double, float, etc.
* Give your actual book to your friend. Now if they make yellow marks, when you get your book back from them, your book has yellow marks. This is passing by “reference”, as we do with arrays.

## Check your understanding

#### **Questions 1:**

Improve this program by moving the repeated code into a method and then call that method twice.

static void Main(string[] args)

{

bool ok = false;

double Value1\_radius = 0;

double Value2\_radius = 0;

while (!ok)

{

Console.Write("Input the first radius size: ");

string userInput = Console.ReadLine();

ok = Double.TryParse(userInput, out Value1\_radius);

if (ok)

{

Console.WriteLine("Good job entering a valid value.");

}

else

{

Console.WriteLine("nope, that input just won't work!");

}

}

ok = false;

while (!ok)

{

Console.Write("Input the second radius size: ");

string userInput = Console.ReadLine();

ok = Double.TryParse(userInput, out Value2\_radius);

if (ok)

{

Console.WriteLine("Good job entering a valid value.");

}

else

{

Console.WriteLine("nope, that input just won't work!");

}

}

double area1 = Value1\_radius \* Value1\_radius \* 3.1415;

double area2 = Value2\_radius \* Value2\_radius \* 3.1415;

Console.WriteLine("The difference in areas is: {0}", area1 - area2);

Console.ReadLine();

}

#### **Questions 2:**

Write a program that asks the user for the width and length of a room in **inches**, and writes back out an answer of how many square **feet** the room is.

Before doing the width and length multiplication, call a method two times to convert each dimension from inches to feet.

#### **Questions 3:**

Write a program to compute the cost of a new car. The customer has 4 models to choose from, and can add zero to 3 options. The base model of the car, DL, costs $15,000. The XL model costs $16,000, the SXL model cost $18,000, and the SXL convertible costs $20,000. The customer can add to any of these models, Deluxe Pearl Paint for $500 more, 5 Year Maintenance for $900 more, and Self Driving Feature for $2000 more. Here is the Main method:

static void Main(string[] args)

{

int model; // holds 0, 1, 2, or 3 indicating which model

bool[] options; // array holds which options user selects

double totalPrice;

model = AskModel(); // ask which model, return 0, thru 3 for which chosen

options = AskOptions(); // let user select which options, returns array[3]

totalPrice = CalcPrice( model, options); // calculate the price and return it

Display(model, options, totalPrice); // Tell the user what they ordered and the cost

Console.ReadLine();

}

Do not modify Main, but add the 4 missing methods to make the program work correctly. Here are two sample outputs:

===============================================

Which model would you like,

1: DL, 2: XL, 3: SXL, 4: SXL Convertible

2

Would you like Deluxe Pearl Paint (y or n)?

n

Would you like 5 Year Maintenance (y or n)?

n

Would you like Self Driving Feature (y or n)?

n

You selected the XL model

and you added no options

and your price will be $16,000.00

===============================================

Which model would you like,

1: DL, 2: XL, 3: SXL, 4: SXL Convertible

3

Would you like Deluxe Pearl Paint (y or n)?

y

Would you like 5 Year Maintenance (y or n)?

n

Would you like Self Driving Feature (y or n)?

y

You selected the SXL model

and you added Deluxe Pearl Pain Self Driving Feature

and your price will be $20,500.00

===============================================

#### **Questions 4:**

Write a program that asks the user to enter a homework score for each of 4 homework assignments and also asks if the assignment was done on time or not. Save the scores in a decimal array and save the late or on-time in a parallel bool array. Now call a method that processes this data. The method should have both the arrays passed into it. It should then use the parallel bool array to subtract 30% for any grade that was turned in late. After you return from that method, call a second method that returns the average score to Main. Now as the last step in Main, write out that average score.

#### **Questions 5:**

Re-write the #1 Question from Chapter 5, the dice game, making good use of methods. Specifically,

* The Main code should introduce the player to the game, then, using a loop of size 5, call a method (OneTurn) to call a round of the dice game, which returns a true or false, based on if the user won or lost. After it leaves the loop, it should then write out the final tally of wins and losses.
* The OneTurn method should
  + Call a method which returns 2 numbers between 1 and 6.
  + The OneTurn method should then call a method, Display, that displays a pattern for the 2 dice, for example

\* \* \*

\* \* \*

\*

\*

\*

Would be a 6 and 3. The Display method should make two calls to one of 6 methods, each of these 6 methods write out one of the 6 patterns.

* Next the OneTurn method should call a method to determine if it was a win or a loss. A win is a total of 7, or 11, or if the two dice are equal.
* The OneTurn should write out if they won or lost that round.

#### **Questions 6:**

Write a simple “draw one” card game. Below is all the code you should paste into a new project.

static Random myRandom = new Random();

static void Main(string[] args)

{

int PlayersCardValue;

string PlayersCardSuit;

int ComputersCardValue;

string ComputerssCardSuit;

bool won = false;

PlayersCardValue = GetCardValue();

PlayersCardSuit = GetCardSuit();

ComputersCardValue = GetCardValue();

ComputerssCardSuit = GetCardSuit();

Console.WriteLine("Your card {0} of {1}", PlayersCardValue, PlayersCardSuit);

Console.WriteLine("Comp card {0} of {1}", ComputersCardValue, ComputerssCardSuit);

won = DecideWhoOne(PlayersCardValue, PlayersCardSuit, ComputersCardValue, ComputerssCardSuit);

if (won)

{

Console.WriteLine("You won!");

}

else

{

Console.WriteLine("You lost.");

}

Console.ReadLine();

}

You should write the required methods to make it work correctly. The virtual card deck has playing cards that are one of 4 suits, Clubs, Diamonds, Hearts, or Spades. Each card also has a value, which is a number from 1 to 13. The game creates a “card” for the player and one for the computer; each card has both a card value and suit. Then it decides who won and tells the user. The decision for who wins is:

If the computer’s cards value is greater than the player’s card value, the computer wins. If it is less, the player wins. If both cards are of the same value, then it is decided based on suit. Clubs beats all 3 other suits. Diamonds beats Hearts and Spades. Hearts beat Spades. If the computer’s card and the player’s card are exactly the same value and suit, the computer wins. The methods you have to write are:

* GetCardValue – returns a random integer between 1 and 13.
* GetCardSuit – returns one of 4 random strings, Clubs, Diamonds, Hearts, or Spades.
* DecideWhoOne – returns true if the player won, based on the logic above.

To debug your program, I suggest you set the allowed values for card value to be from 1 to 2, that way you will get many more ties on value and can make sure your suit logic is correct. Below are a few sample outputs from running the game a few times. You must test this extensively to make sure all the various combinations are scored correctly. I will take off points not only for a simple mistake, but also if it is clear you did not test it well.

Your card 2 of Diamonds

Comp card 2 of Clubs

You lost.

========================

Your card 1 of Hearts

Comp card 2 of Diamonds

You lost.

=======================

Your card 1 of Clubs

Comp card 1 of Hearts

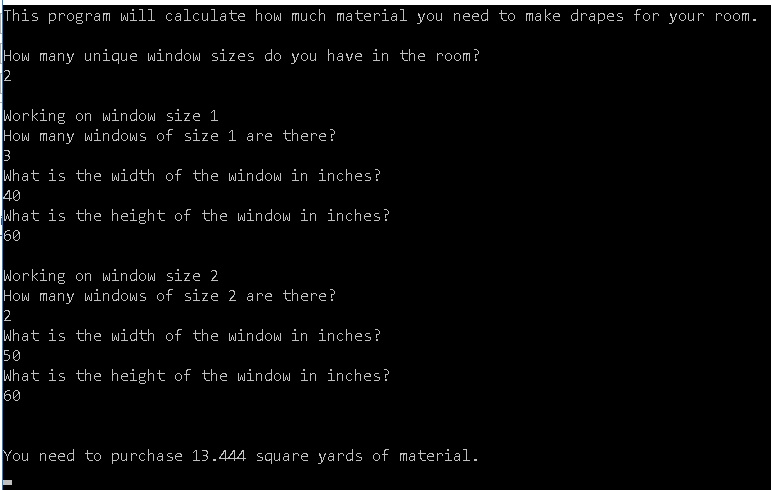
You won!

#### **Questions 7:**

Write a program that will calculate the total amount of material needed to create drapery for a room. Specifically:

* It should ask the user how many unique window sizes the room has.
* It should ask the user how many windows are of each particular size.
* Any user output should covert “computing counting” (which starts at zero) to “human counting” (which starts at one).
* You do not have to validate input.
* For each unique windows size, the program should ask for the height and width in inches.
* It should add 6 inches to the height of each window to allow for the drapery rod.
* It should add 20% to the width to allow for pleats
* It should write out the total material needed in square yards.
* The design should be a loop in the Main method, which calls a method one time for each unique window size, and that method should in turn, call another method to calculate the area for one particular window.

Here is a sample output:



# Chapter 10 – Windows Forms

### The utility of Windows Forms

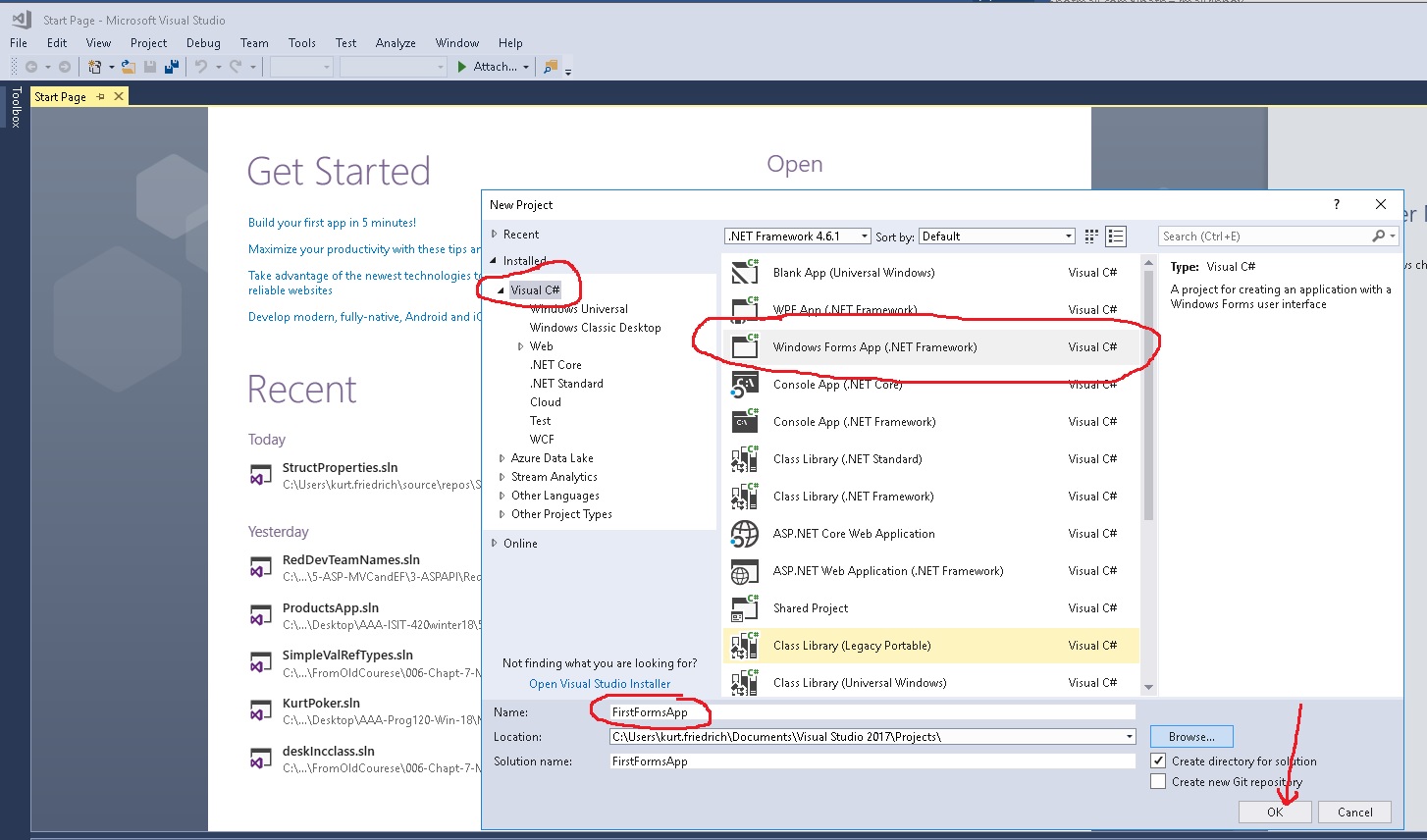
Before the internet changed the entire landscape of computing, a high percentage of applications that ran on Windows based Personal Computers used a forms framework from Microsoft called Windows Forms. Since so many current applications interact with the internet, it is used much less than before. However if you ever need to create a standalone Windows application, you can generate a nice application with a pleasing user interface very quickly. It has many built in objects that do lots of “heavy lifting” for you; you just drag an icon onto a blank form, type a few lines of code, and almost magically you have a good looking, useful application. I make use of Forms in most of my follow-on courses as it lets the students focus on the topic at hand, without spending lots of time writing the user interface. At this point in your learning, it also gives you a great introduction to making use of objects, properties, and methods from class definitions supplied by someone else. This all leads to increased productivity. Here is a good site with reference information from Microsoft’s on line documentation at:

<https://docs.microsoft.com/en-us/dotnet/framework/winforms/>

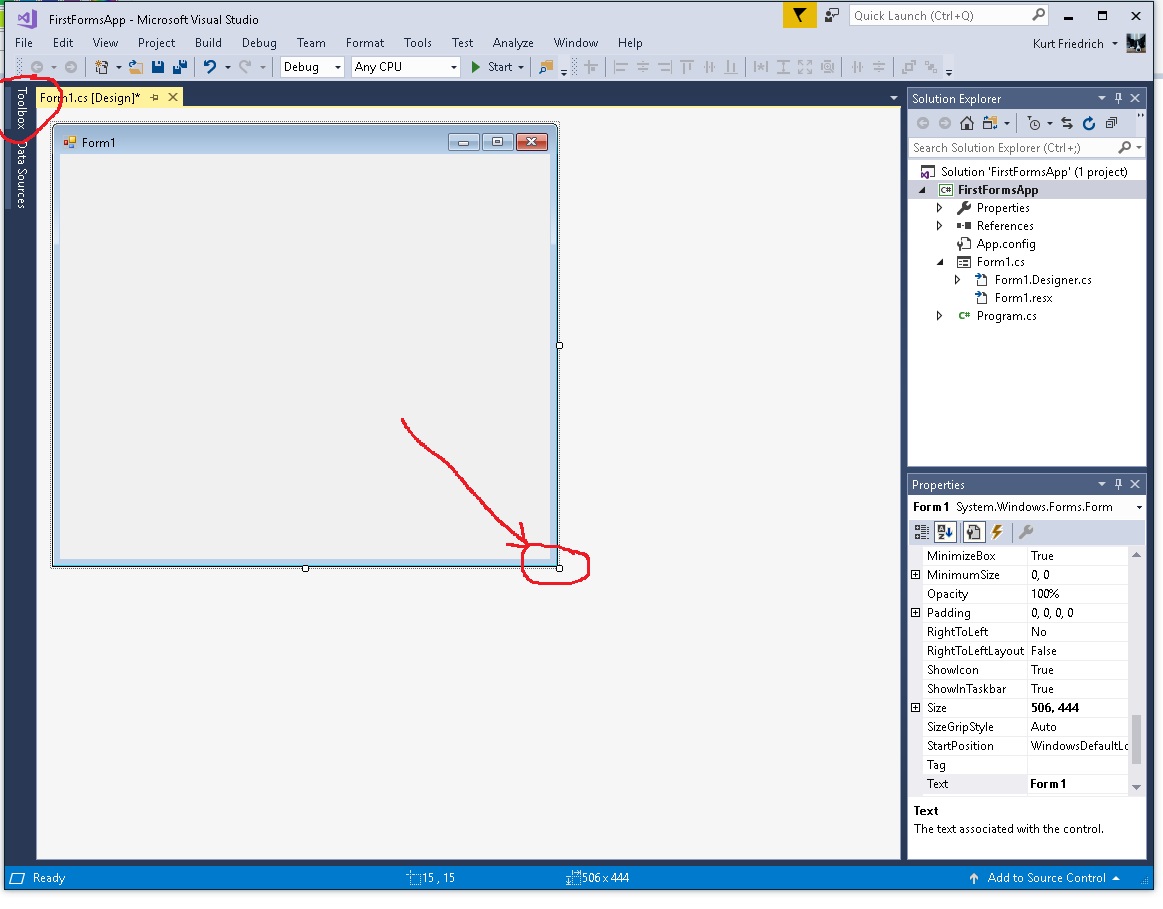
The goal of this chapter is not to make you a highly qualified Windows Forms developer, but rather, a programmer that is well aware of what Forms are and how they work. We will work through two sample programs and when we are done, you should understand enough that you should be able to write some very useful Forms programs, and with a little searching on the web, any Forms program that you set your mind to.

### Quick Start, Our First Forms Program

We will just build a simple app so that it is clear what we are talking about and how to use Forms Start VS and select a Windows Forms App instead of our usual Console App.

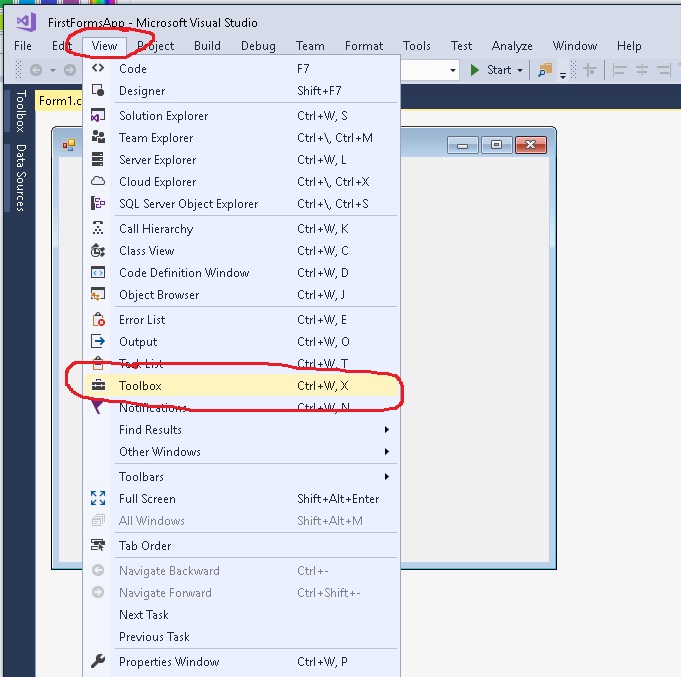


You are now looking at an empty form. Grab the lower right corner and drag it to make it a larger form.

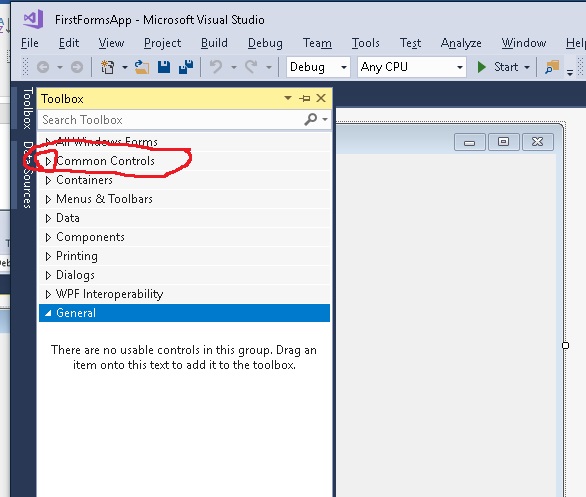


Now run your new application, but clicking the green arrow “start” icon. You will get a window added to your screen that is your empty form. So now you have your first Form app.

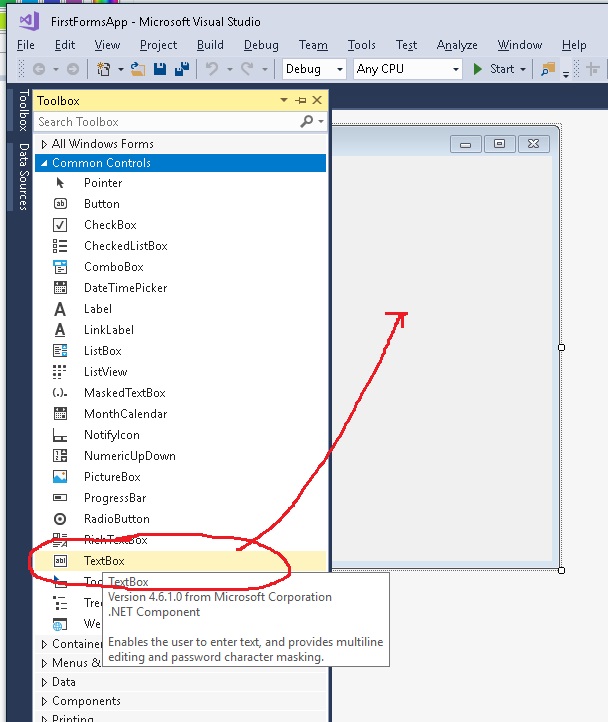
We will now add to this application to have it add 2 numbers and tell us the total. We will add items (objects) to our form, by clicking on the “Toolbox” on the left border, it will open up and offer us many objects we can add to the form. (If for any reason, that Toolbox item is not showing there, you can also get to it through the menu, see this next picture:



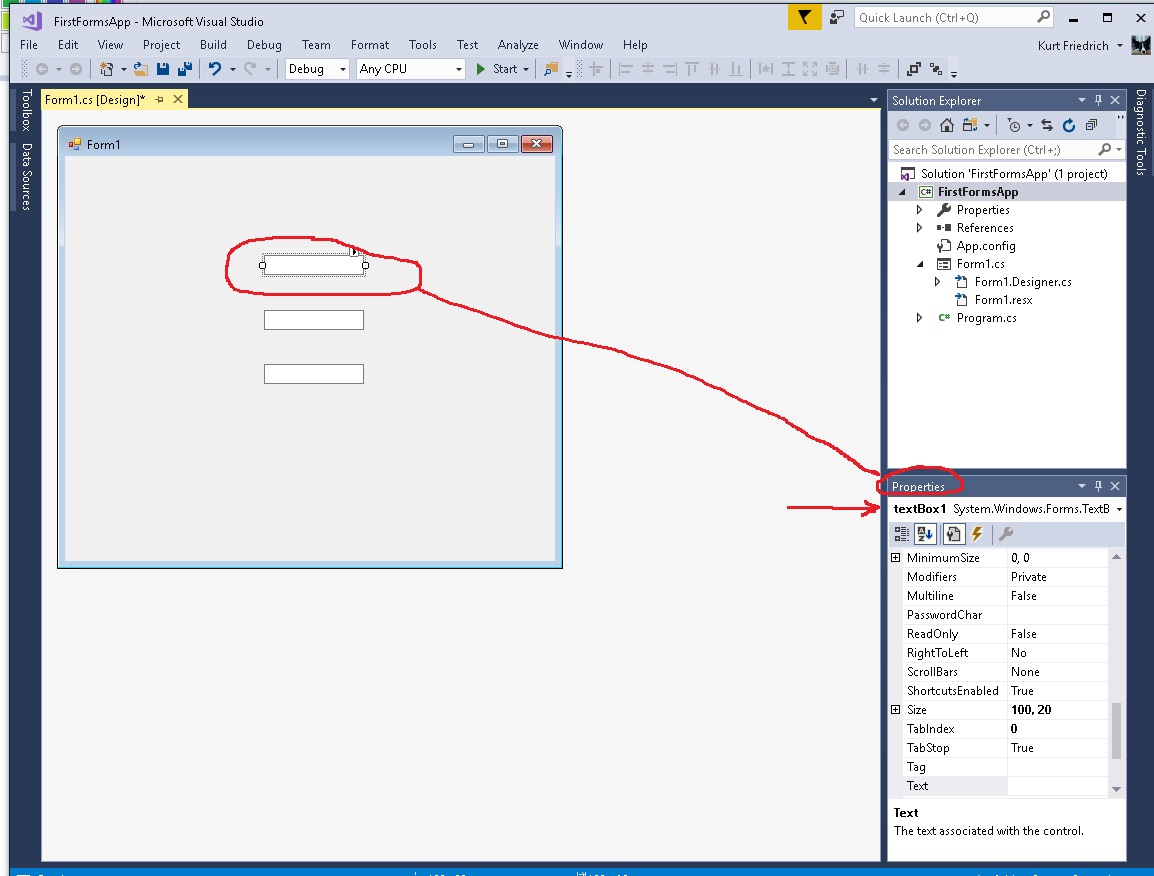
Now find the Common Controls



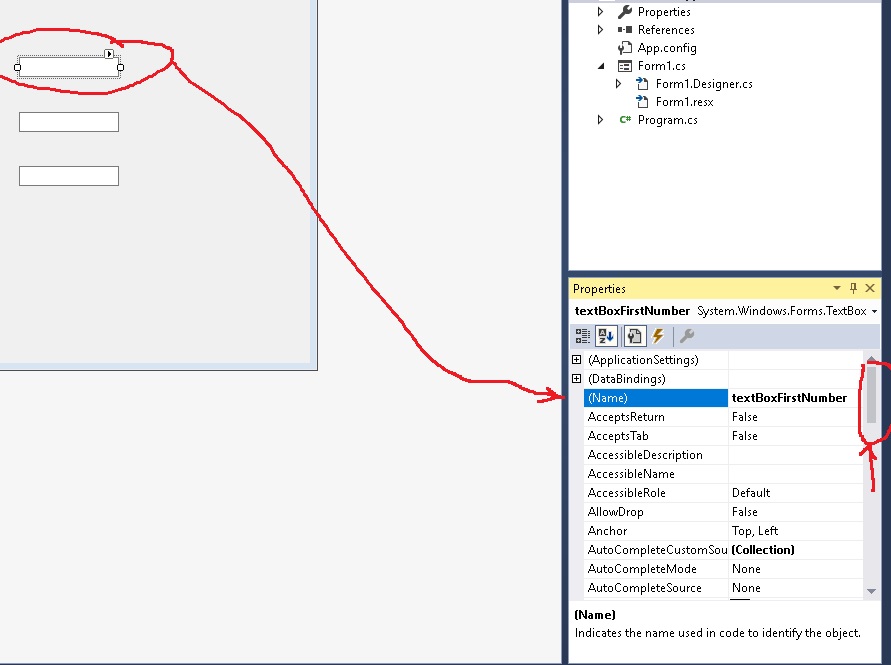
Click on the menu for “Common Controls” that is where we will find all the objects we want. Windows Forms refers to these objects as “Controls”. Whenever you see that, in your head, substitute the word object instead.



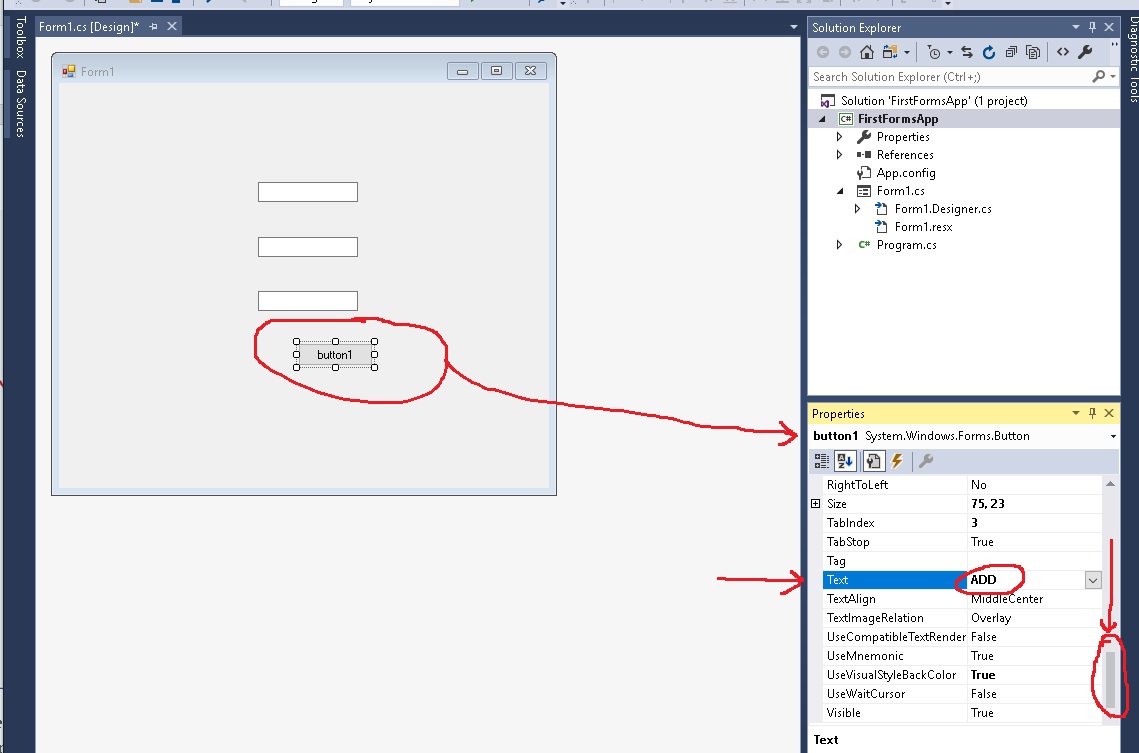
Click and drag/drop the TextBox object onto your form. Do that two more times until you have 3. Move them around to align them until you see:



Now comes an important part of this lesson. The Form1, and each of the 3 textboxes you have created, are C# objects. And like all C# objects, they come with many “properties” (nouns) and also “methods” (verbs). Notice that if you click on the empty form, or any of the 3 textboxes, the Window in the lower right changes to show all the properties of the object you have selected. One of the properties is the name of the object. Right now our textboxes defaulted to textBox1, textBox2, and textBox3. Let’s change their names to be more useful.

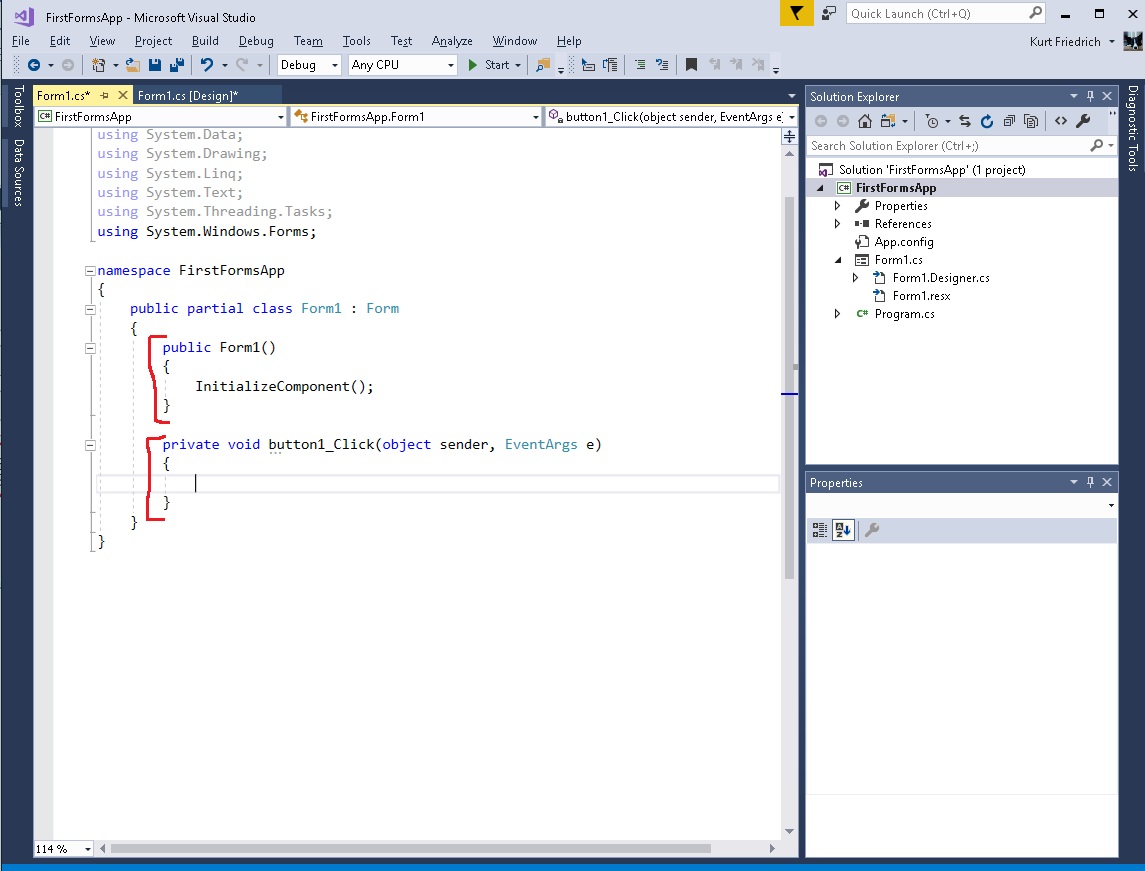


Click on the first textbox, over in the Properties window, use the scroll bar to scroll to the top. Find the property called (Name) (which is technically not a property, but never mind that for now!) and edit the right hand side to read textBoxFirstNumber . Then click on the 2nd textbox and rename that one to textBoxSecondNumber . Then click on the 3rd one and rename it to be textBoxAnswer. Now if you run your application, it will draw the form with the 3 textboxes. You can enter numbers in them, but they don’t do anything. Windows Forms uses a type of programming called Event Programming. The program has code that does nothing until it is “woken up” because an event has happened. We code our Forms programs to do something when an event happens. There are many events to choose from, such as, a new message arrived, a certain time of the day, or some mouse event. In our case, we want to program our project such that when the user clicks a form button with the mouse, our code will add the 2 numbers in the upper two textboxes, and write the answer into the bottom one. Start by dragging in a button to your form, similar to how you dragged in the textboxes.



After the button is on the form, click it ONCE! Be careful not to double click it! The Properties window will now show values for the button1. Drag the slider to the bottom and find the property called Text. Edit to the right of that to change the text that the button displays, edit it to say ADD.

NOW double click on the button, and Visual Studio will open a file called Form1.cs This is C# code that the form uses.



This code has a special method (called a constructor) named the same as the form, Form1(). It runs the code that creates the form and all the objects when you start the program. The 2nd method,

private void button1\_Click(object sender, EventArgs e)

was auto created by Visual Studio when you double clicked the button1. This is a method that will be run every time the button1 is clicked with the mouse by the user. If you run the project now, and look very closely, you’ll notice that the button flashes briefly when you click it. It ran the code in the new method, but since these is no code there, it didn’t do anything. For now, we are done editing the form, it’s time to edit this method and have it do its job. We want the method to read the values in the two textboxes, add them together, and write the answer back to the 3rd textbox.

You might think this code would do it:

textBoxAnswer = textBoxFirstNumber + textBoxSecondNumber;

But it won’t. Textboxes are objects, you really don’t want to add them. You want to add the **text property** of each box. So try this:

textBoxAnswer.Text = textBoxFirstNumber.Text + textBoxSecondNumber.Text;

Run the app, enter 44 is the first box, 88 in the second box, and click the button. Your Answer box will show 4488. That is because you just concatenated two strings. To make this do real math, we need to convert the textbox strings into numbers, then add them, then convert the answer back to a string and store that in the answer textbox, like this:

private void button1\_Click(object sender, EventArgs e)

{

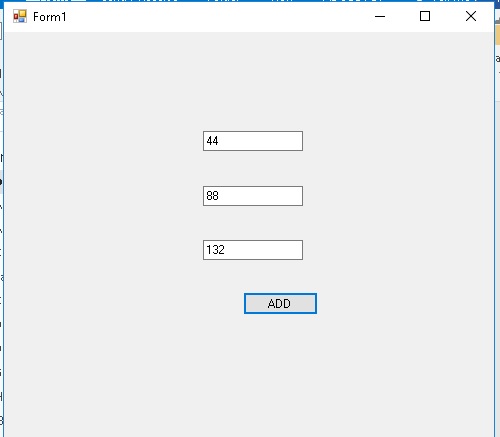
decimal first = Convert.ToDecimal(textBoxFirstNumber.Text);

decimal second = Convert.ToDecimal(textBoxSecondNumber.Text);

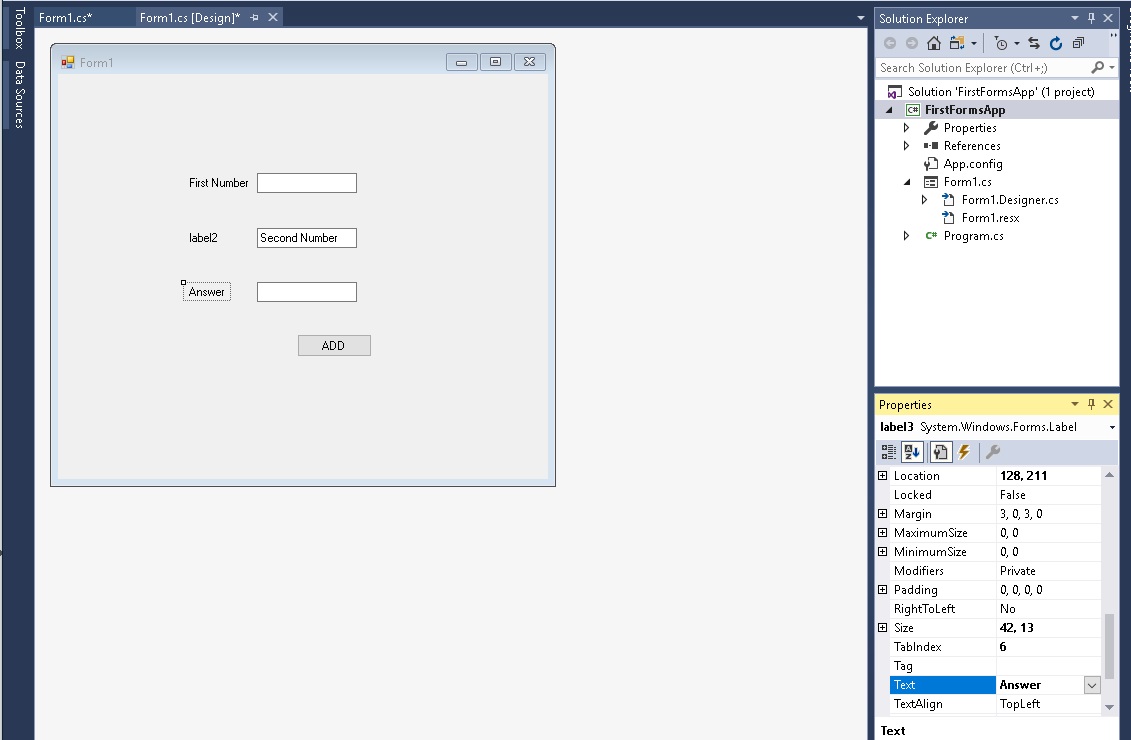
textBoxAnswer.Text = (first + second).ToString();

}

Now try your program. 44 plus 88 should result in 132.



One last step, open the ToolBox menu and drag in three labels, place them beside the 3 textboxes. Change their Text property to read “First Number”, “Second Number” and “Answer: It should look like this:



You have now completed your first Forms project. Save it, as you will extend this in Probem 1 at the end of the chapter.

### Pizza Parallel Arrays, Our Second Forms Program

Now we are going to revisit the Parallel Array Pizza problem we saw as Question 4 back in Chapter 5. But we will code it with Forms to look nicer, and I am making one requirements improvement. Here is the new version of the problem here:

*Write a program that will calculate the total cost of a pizza. The Form should allow the user to select if they want a S, M, L or XL pizza (small, medium, large, extra large). Based on their answer, they should add $5.00, $7.00, $9.00, or $11.00 to the total bill. The form should also offer check boxes, so they can select from one to four extra toppings (mushrooms, sausage, pepperoni, anchovies). Based on the size of the pizza, add this amount for every topping they select, .75, 1.00, 1.50 or 2.00. Write out a message telling them how much the total bill is. Keep track of the cost with a type decimal.*

Start a new Windows Forms project. Drag the corner of the form to make it bigger. Drag a comboBox onto your form, this will provide our dropdown selection for the size of the pizza. We will want the values in the dropdown to be set as the form is shown to the user, so we need code to execute before the form is drawn on the screen. To do that, double click on the empty form, and it will take you to the C# program file, and it will have created a special method.

private void Form1\_Load(object sender, EventArgs e)

The code in this method will be run before the form is shown. But first, go to the very top of this class and add the three parallel arrays and price variables we need.

public partial class Form1 : Form

{

// add these 4 lines here

string[] Sizes = new string[4] { "S", "M", "L", "XL" };

decimal[] Prices = new decimal[4] { 5, 7, 9, 11 };

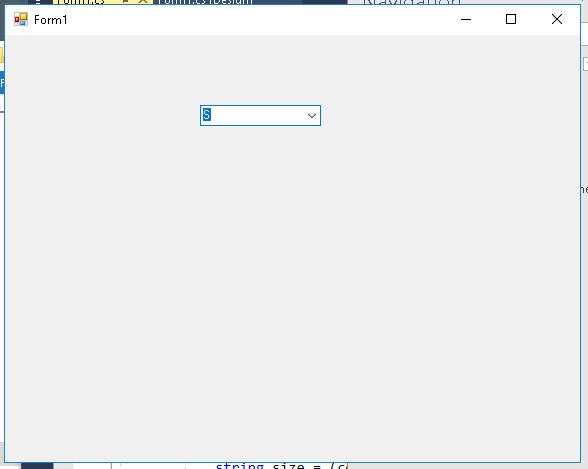
decimal[] Toppings = new decimal[4] { .75m, 1, 2, 3 };

decimal price = 0;

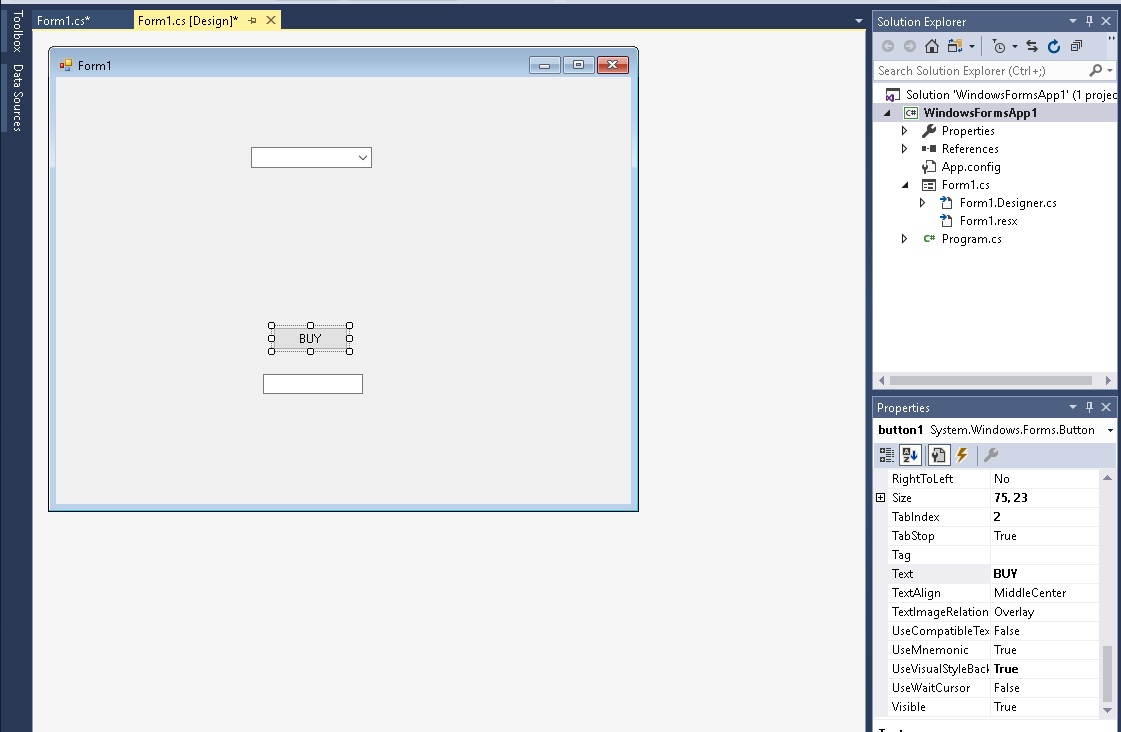
Now go to you newly created Form1\_Loaded method initialize the comboBox1 by adding one line of code. It has a property name DataSource, that you can set to be a “collection” of data. In our case, our collection of data is our Sizes array.

comboBox1.DataSource = Sizes;

Run your Form program and you should see your dropdown list populated with our pizza sizes. We have just taken advantage of a considerable amount of code that is supplied in the comboBox Class, as it was very easy to load it with data.



Now drag in a textBox towards the bottom of your form, and rename it to textBoxTotal. This is where we will show the customer the cost for their pizza. In order for us to have code that reacts to the dropdown setting, we need an event to happen to run some code. As before, we will use a button click event. Drag a button into the form, being careful not to double click the button. You can leave the name of this button object as button1, but modify its Text property to say BUY.



NOW double click it and you will be taken back to your C# code, and a new method will have been created by Visual Studio for you;

private void button1\_Click(object sender, EventArgs e)

In this method, we need to calculate the cost. So we have to walk through our Sizes array to find a match between the selected pizza size, and an element in the Size array. Once we have that index, we can use that index to get the pizza cost from the Prices array, and write it out to the user. So first, we need to get the string value of the selected drop down choice. This code will do that:

string size = (comboBox1.SelectedValue).ToString();

That reads a property, SelectedValue, form the comboBox1 object, and ask for it in the form of a text string. Now we need a loop to match that value to one in our array:

for (int i = 0; i < Sizes.Length; i++)

{

if (Sizes[i] == size)

{

price = Prices[i];

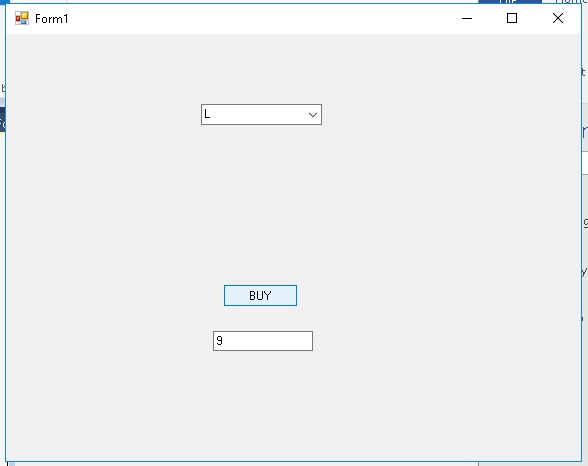
}

}

When we find the match, we use the index value of i over on the Prices array to get the corresponding price. Now we can tell the user the price by writing it out to our textBox;

textBoxTotal.Text = price.ToString();

This is the end of the button1\_Click method. If you run the project, you should see the price in the textBox when you click the button.



Now we need to allow for toppings. Drag a checkedListBox onto your form. We will leave it with the default name of checkedListBox1. As with our dropdown list, we need to load this with data before the form is drawn for the user. So we will add more code to the private void Form1\_Load(object sender, EventArgs e) method.

Our checkedListBox1 has a property called Items, which is a List where we add each checkbox we want. So this code will add 4 checkboxes:

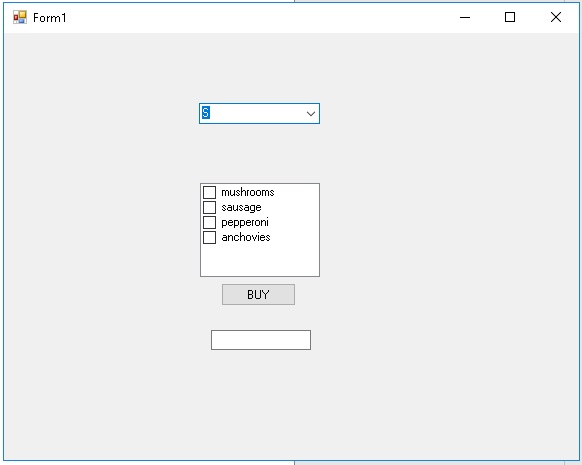
checkedListBox1.Items.Add("mushrooms");

checkedListBox1.Items.Add("sausage");

checkedListBox1.Items.Add("pepperoni");

checkedListBox1.Items.Add("anchovies");

If you run the program now, you should see our new set of four checkboxes. To actually select one, you have to click on the line item, and then click a second time on the checkbox such that the box really has a check showing. It will allow you to select from zero to all four.



For our last step, we need to add the price of the selected toppings to the price of the pizza. Again we will take advantage of parallel arrays, so the best place to add this code is in our button1\_Click method, inside of the loop where we match the pizza size, and inside of the if code block when we have found the match. So modify that if to look like this.

if (Sizes[i] == size)

{

price = Prices[i];

for (int j = 0; j < checkedListBox1.Items.Count; j++) // loop for as many boxes

if (checkedListBox1.GetItemChecked(j)) // if that box is checked …

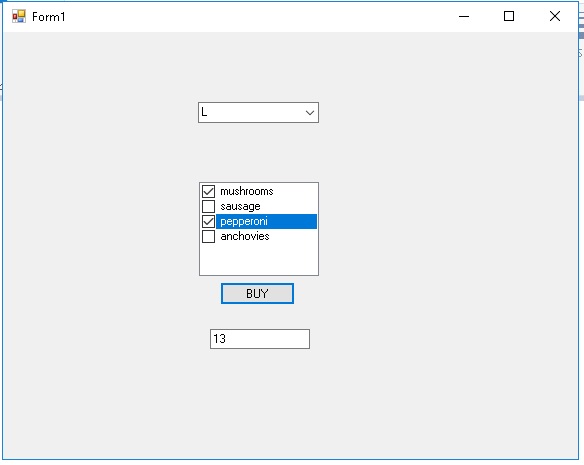
{

price = price + Toppings[i]; // use the parallel array to get price

}

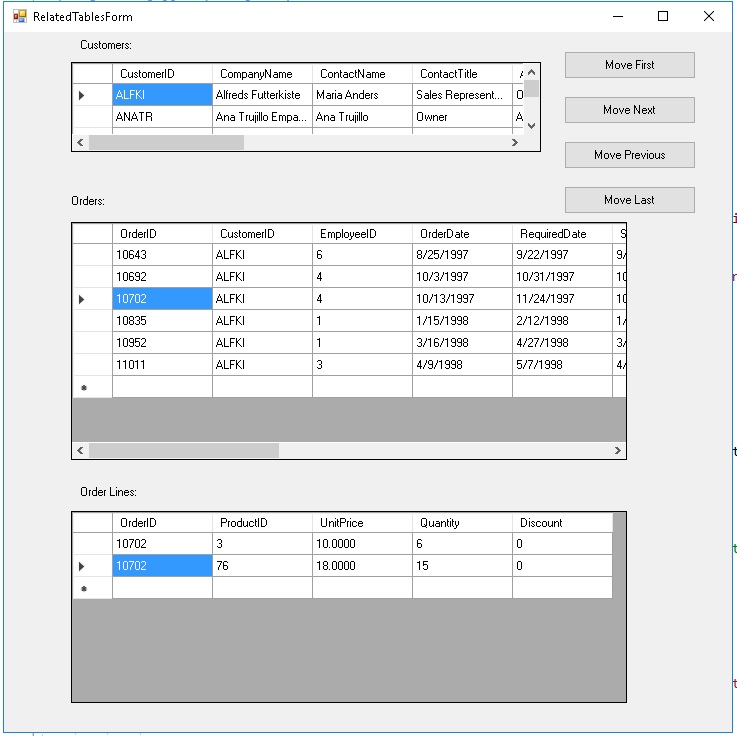
}

Now run your program a few times and verify it is getting the correct prices.



### Forms Summary

If you stand back 20 feet and think about Forms, what they are is a different way of reading and writing information to the screen. All the material covered in the prior chapters is entirely re-used. As we saw when processing the price calculation in the pizza button1\_Click method, we still used if logic, looping logic, parallel arrays. Often a simple form program will have a method or two that contain very complex code, with a button click method making many calls to other methods of yours, as well as .Net library methods. For example, a button event method might make calls to two different SQL database tables, compute some results from data, and present the results as a nicely formatted screen of data.



## Check your understanding

#### **Questions 1:**

Extend the first program by adding 3 new buttons, SUBTRACT, MULTIPLY, DIVIDE, and, after double clicking each of the 3 new buttons to create event methods, add the code in the methods so that the correct answer is shown.