

BA272 Parts of a Program Study Guide

We compiled a program using Visual Studio in class and talked about some of the features of the IDE (Integrated Development Environment).

Parts of a program (Through page 31 of the text)

You need to understand the things the text says about these components even if I don't repeat it here:

- Look at program (p. 24) – Is it as simple as you thought to compute area and length in a program?
- **using System** Begin to think about organizing bits of code. System is a library of things Microsoft coded the .Net framework to do so that you wouldn't have to. This notion of breaking things into pieces we can use again is a main theme.
- **Class GlazerCalc** We will only lightly discuss "object oriented" in BA272 – other basics are more important for now but these things you should be able to repeat, even if you don't completely understand:
 - o Modern computer programming languages create programs that do their work using electronic "objects."
 - o An object has a place in the computer's memory (an address) so that the operating system and other parts of the program can "find" it to do its work.
 - o An object "encapsulates" data and functionality together. Notice that GlazerCalc remembers some numbers (2 and 3.25) and also has some things it does (adds multiples accepts and displays).
 - o In part, a class is a template used to create objects. It tells the computer to make room for the data and contains the instructions the object needs to do its work.
 - o You "instantiate" a class to make an object. Thus, we call an object made from the class an "instance" of the class. Think of the mold (class) a factory might use to make an action figure (instance). If you make a bunch of Power Ranger figures from the same mold they are a lot alike. But each goes on to have different adventures when different kids play with them.
 - o We will be defining classes in our source code, creating objects in memory to do things, and using the defined instructions to accomplish our tasks.
- **static** Note the definition: "part of the enclosing class which is always here."
 - o So the pattern notion doesn't hold up after a while. If there are or can be any instances of a class (objects) then the class itself is always hovering nearby. For some activities, the instance may access data or instructions in that class. So when we make up a class definition most things automatically become part of the object but we make some things (static things) exist only in the class. It does not mean static things never change. It does mean that static things are always the same for all instances of the class. If the class thinks my static color is "blue" then the color of all objects is automatically blue. If we change the static color (the one stored in the class) to red, then all the objects are red. If color is NOT static, then one instance of the class may be blue while others are green or red. Other than giving you fits when you write programs, this distinction won't be very important for the programs we make in BA272.
- **void** means a method (set of instructions) will not hand back an answer. Much more on methods throughout the term.
- **Main** describes what a program does when it starts up. It is always *static* and *void*. Why? Because! But here is an attempt at an explanation.

- You need a running program to instantiate an object from a class. What do you do before you have any objects? Answer: run *static* methods on a class. All the classes start hovering around as soon as the operating system decides to load up the program. Static things (like the main method) are therefore readily available. The Main method makes us a place where we can start making objects from classes to get some work done.
 - *Main* is *void* because it is first in line. It isn't answering questions for another part of the program, it asks the first questions.
- **()** you always need to put these after when you ask an object to do something by calling a method. Did you catch that a method is something an object can do? You may want to send some last minute details. For example, when chased by a turtle you might want your action figure to run(slow) but when chased by a tiger run(fast). If there aren't any details you just say run(). You may want to remember that we call such passed details **parameters**.
- **{** and **;** Programming languages have syntax. Braces { } and semicolons ; are part of C#'s syntax. You'd better get happy with them now.
- **=** the equal sign is also C# syntax. Pay close attention here: gozzinta is an instruction NOT a question! Be sure you understand the difference between $4 = 2 + 2$ (a question or assertion) and `myWidth = 5` which says "put the value 5 into the variable myWidth. More later, but if we want to know if the value of myWidth is 5 we will say `myWidth == 5` – this is just a stupid computer syntax thing. If you don't say it the way the computer wants, too bad for you. It simply won't do what you want it to do.
- **double, string** Different variable types hold different kinds of things.
- **Width, height, woodLength, glassArea** Like GlazerCalc, these are names we chose for things in this particular program. Hopefully the names we choose will be informative.
- **Console** Console is an object that is automatically made available through the .Net framework. Most every other programming language uses the same name for the same thing. We don't have to write code to directly talk to the hardware that makes specific pixels light up or determines which key was pressed. We just talk to this Console object.
- **ReadLine()** You should know by now why Readline is followed by (). It is a method of the Console object and we don't give it any details to do its work. We just want it to read in the next line from the console.
- **height = double.Parse(heightString)** a class has both data and instructions. Static methods live in the class, not in the instance (see above). So here, we are calling a static method that is in the **double** class. This parse behavior is static so we don't have to talk to any particular double precision floating point number. We talk to the ever-present class double which makes it possible for us to have double instances such as **width** and **height**. The class double has a method such that when we pass it a parameter – thus the (heightString) - it will do its best to convert that parameter into an equivalent double precision floating point number. The gozzinta (=) shoves the resulting numeric value in the double instance **height**. Whew, that's a lot to take in. Try hard to get it. But you may not really understand yet. Come back to this each class. If you really understand this you are mastering some of the hardest concepts in BA272.
- **+ and *** programs can do simple math using **operators** such as +, -, *, and /. $A = B + C$ is not a question, it is an instruction. The gozzinta (=) puts the sum of B and C into the variable A.