**Review and Research**

You can review the video of Session 3 here:

<https://blizzard.sharepoint.com/portals/hub/_layouts/15/pointpublishing.aspx?app=video&p=p&chid=8aa7fa80-bfa2-4021-bf44-543dba93f693&vid=e93490d1-fdb7-4374-bf05-8a6b7310d8c0>

As always, email me at [semerson@blizzard.com](mailto:semerson@blizzard.com) if you have any questions!

For a nice, simple overview of most of the concepts we discussed in class, check out this tutorial:

<http://www.tutorialsteacher.com/csharp/csharp-class>

In our most recent session I mentioned a cool string formatting feature available in modern C# called string interpolation. You can read about it here:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated>

We spent a good bit of time chatting about properties and accessors in this session. For a good, comprehensive look at properties, check out the MS documentation here:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties>

In our most recent session we started using the MSVS debugger for the first time. Here’s an excellent set of documentation that will help you review important debugging features across various versions of Visual Studio. I would strongly encourage you to read through this documentation and learn even more about the debugger; it is a very powerful tool!

<https://msdn.microsoft.com/en-us/library/k0k771bt.aspx>

**Glossary**

Class – You can think of a class as a type that you’ve created yourself (or that someone else has created for you to use). Classes can contain state (i.e., variables that hold data) and methods (functions) that are generally designed around a central unifying theme. For example, a class representing a bank account might contain variables that store the account number and current balance, as well as methods to allow transactions to take place.

Object – If a class is like a blueprint, then an object is like an item created using the blueprint. Objects are usually declared with names just like other variables. We often use the terms “object” and “instance” interchangeably.

Member – Any function, field, or property that belongs to a class is called a member of that class.

Field – A variable belonging to a class.

Method – A function belonging to a class.

State – Another name for data, usually in the context of a class or other structure. When a class contains one or more fields, we say that it has state.

Access Modifier – Access modifiers are keywords that indicate what degree of access should be granted to a class or the contents of a class. For example, if an object has a public field, that field can be accessed at will by any part of the program that has access to the object. If a field is designated private, it is off limits and can only be accessed from within the class itself. The four main keywords we use in C# to indicate access are public, private, internal, and protected.

Encapsulation – This is an object-oriented programming concept that refers to the idea of keeping data private as much as possible in order to protect it. Encapsulated data is usually controlled by class methods that only allow specific, approved access.

Property – A property is a class feature that assists in accessing data. It does this via special methods call accessors.

Accessor – An accessor is a specialized method that is used to read or write data within a class. There are two accessors: The *get* accessor is used to read data. The *set* accessor is used to write data. Note that a property doesn’t have to define both accessors; in this way, read-only or write-only properties can be created (by omitting the *set* or *get* accessors, respectively.)

Invariant – An invariant is a rule that determines how state may or may not be set. For example, if we’re creating a class that represents a triangle, we probably don’t want the side lengths to contain values that are less-than or equal-to zero. We should put an invariant in place to make sure this never happens. The *set* accessor of a property is an excellent tool you can use to enforce an invariant.

Constructor – Constructors are special functions that help us initialize an object. A constructor must have the same name as the class to which it belongs, and unlike other functions it will never be given a return type. Every class must have at least one constructor, and if we don’t define one the compiler will create a default constructor for us. Every time an instance is created via the *new* keyword a constructor must be called. Classes can (and often do) contain more than one constructor to allow different initialization options.

Function Overloading – In C#, you may define multiple functions with the same name within a single namespace, provided the parameters taken as input are different for each one. This is called function overloading. Defining multiple constructors for a single class is a common example of function overloading.

Static – The static keyword, when applied to a class member such as a field or method, indicates that the member belongs to the *class* rather than any individual *object*. For example, if a field is declared static, there is only a single copy of that field that resides within the class itself; individual instances do not receive their own copies of this field.

**Practice Exercises**

These exercises are grouped around concepts we’ve covered in class and range from very simple to more complex. As always, a few reminders:

1. When compiling and running these exercises in Visual Studio, be sure to do so in Debug Mode. You can do this with the hotkey combination ctl+F5.
2. These exercises are meant to be a start to your practice, but if you want more ideas, contact me and let me know. I’ll be happy to make additional suggestions!
3. Don’t be afraid to repeat an exercise several times. Repetition in your practice is very helpful, especially in the beginning when you’re trying to get comfortable with syntax as well as new concepts.
4. If you get stuck on a problem or don’t understand why something is happening, please contact me and let me know. I’ll be happy to help.
5. An exercise may occasionally require you to use a concept we haven’t covered in class; when this is the case, the exercise will be marked with an asterisk there will be a link to research you can do help you find the new information you need.
6. Don’t be afraid to use Visual Studio’s debugging features to help you understand how your program is working! This can be useful not only for fixing problems but also for gaining a better understanding of how your program is working.

*Creating and using classes*

*NOTE: Exercises 1 – 8 all incrementally build on the same class and should be completed in order.*

Ex. 1. Create a class called Employee that represents an employee record. Provide the class with three public fields: A string for the employee’s name, a string for a job title, and a uint for an identification number. In the Main function create an instance of an Employee and set its fields to any data you like. Write these various fields to the console.

Ex. 2. Change the identification field’s access modifier to private and initialize it with 0. Add a property to your class that allows you to get and set the ID field’s value.

Ex. 3. Return to the property you created in Ex. 2. Add an invariant by modifying the set accessor so that an employee ID can never be greater than 1000.

Ex. 4. Add a constructor to your Employee class that allows you to initialize a new Employee with a name, job title, and ID. Within your constructor, assign the name and job title fields directly from the input provided to the constructor. Assign the ID argument to the property you created in Ex. 3 (think about why this is a better approach than simply assigning directly to the field.)

Ex. 5. Add another constructor to your Employee class that takes no arguments and initializes the three fields to default values of your choice. In the Main function try instancing Employees using both constructors to get a feel for the difference. Create additional constructors if you want more practice!

Ex. 6. Add a public method to the Employee class that provides a report on the employee. It should write the Employee’s name, job title, and ID to the console. Try instancing an Employee in Main and calling its reporting method. Bonus: If you are using Visual Studio 2015 or later, use string interpolation to simplify the string you write to the console (see the research section above for a link on how to do string interpolation.)

Ex. 7. In Ex. 3., we created an invariant that prevents Employee IDs from being greater than 1000. If you wrote your *set* accessor in such a way that the number 1000 appears in the code, you have a magic number! We should try to fix that. Create a static uint field and initialize it with the value 1000. Modify your property so that, instead of checking against the integer literal 1000, it checks against your new static field. (Bonus research: Try to make your field both static *and* const. Does this work? If not, try to determine why!)

Ex. 8. Back in your Program class, create a static function that takes two Employees as input and writes the name of the employee with the higher ID number to the console. If both Employees have the same ID, write a warning to the console that this is the case.

Ex. 9. Think of things in the real world (or in fantasy worlds!) that you can represent with classes, then try to implement simple versions of them. Don’t be afraid to experiment by adding constructors, methods, fields, properties, and static members as necessary. Here are a few ideas to get you started:

* A class representing a Rectangle. Each Rectangle should have a floating-point width and height. These fields should be private; create properties to allow read-write access to them, but don’t allow either dimension to be less-than or equal-to zero. Provide a method that calculates and returns the area of the Rectangle.
* A class representing a dog. The Dog class should have private fields for breed, coat color, and name. Initialize all these fields with a constructor, then allow read-only access to them with properties. Feel free to create a Cat class instead if you prefer (or both if you’re industrious!)
* A Hero class representing a player avatar in an RPG. The Hero should have a level, experience points, and stats for strength, intelligence, and wisdom. These stats should be initialized by a constructor, but every Hero should start at Level 1 with no experience. Add a method that adds a given amount of experience to the hero. Every time a Hero gains 500 experience points its level should go up and its stats should increase by 5.
* A class representing a Bridge. Provide fields representing various dimensions and one or more constructors to initialize them. Add properties for access and add invariants as you see fit. Add a read-only field / property describing the maximum weight the bridge can support before collapsing.
* Bonus: Now create a Car class that contains, among other things, a weight that can be accessed via a property. Write a function in your Program class that, given a Bridge and a Car, returns true if the car can safely cross the bridge, and false if not!