**Inheritance & Polymorphism: The ToString method**

**The goal for this exercise** is to make sure that you can override the ToString method, which is a standard method that all C# objects inherit.

 As it turns out, all classes in C# (and .Net generally) inherit from a common base class, which ensures that all .Net objects will have certain methods. This affords us an excellent opportunity to practice overriding a method that we didn’t create ourselves.

 For this exercise, we’ll override the ToString method, which allows an object to generate a string that represents that object. For example, an object that represents the (X,Y) point (3,10) might produce the string “**(3,10)**” when it’s ToString method is invoked. Knowing that this method exists (and how to override it) is often useful when debugging programs – you can ask any object in your program to print itself to the console using something like Console.WriteLine(“Object: {0}”, obj.ToString() );

 All C# classes inherit from the confusingly named **Object** class. Consider the following:

|  |  |
| --- | --- |
| Line | Source Code |
|  | **class A : Object** |
|  | { // left out for clarity |
|  | } |
|  | **class B: A** |
|  | { // left out for clarity |
|  | } |
|  | **class C** |
|  | { // left out for clarity |
|  | } |

 On line 7, **C** inherits from **Object** directly, but implicitly. Basically, if we don’t specify a class to inherit from, the compiler will add some invisible code so that our new class inherits from **Object**.

 On line 1, **A** inherits from **Object** directly, but explicitly (meaning that it’s written out – as oppose to implicitly). Since leaving out the “**: Object**” part will result in our class inheriting from Object anyways, you almost never see this (instead, you’ll see code like the one on line 7). But you can do it this way, if you want.

 On line 4, **B** inherits from **Object** indirectly. **B** inherits from **A**, and since **A** inherits from **Object**, **B** also inherits all the data & methods from **Object**.

 There are several purposes for this, but the one we’ll look at here is this: by forcing every single class in C# (in .Net, really) to inherit from **Object**, we *know* that every single class has the option of overriding any virtual methods defined on the Object class. For this exercise, we’ll focus on the **Object.ToString** method.

 The ToString method should return, in **string** form, a concise summary of the **Point**’s attributes, and is commonly used for debugging purposes (i.e., so you can see a quick summary of the object, either in the debugger, or as output intended for the programmer to use for debugging purposes).

**What you need to do for this exercise:**

1. Look up (either in Visual Studio’s online help, or msdn.com) the **Object** class. Read the description, and skim through the methods & any properties it might have.
	1. Read through the ‘ToString’ method in detail.
2. Compile and run the code provided to you in Overriding\_ToString. RunExercise(), and note the output. You may want to copy the output someplace so that you can compare it against the ‘end result’ output you’ll get later.
3. In the provided starter project, in the file named Program.cs you can find the **MyPoint** class; you will note that part of it has already been written for you. You need to correctly override the ToString method. This should work pretty much the same as any other exercise around polymorphism – the only real difference is that instead of you defining the base class, Microsoft has already done that for you.
	1. Hint: Your ToString method must have the same signature and return type as the one defined in the Object class that you looked up online.
	2. Hint: The ONLY code that you write for this exercise should be to add the properly overridden ToString method to the MyPoint class, and nothing else.
	3. Hint: When you’re done the output from the in Overriding\_ToString. RunExercise()method should similar to the following (Your output does not need to be exactly the same, but it should be able to print numbers with decimal places):

	Your point is at:(3.0,7.6)
	Your point is at:(3.0,7.6)
	Your point is at:(3.0,7.6)
	Your point is at:(3.0,7.6)

	Note that are NO spaces in the string generated by MyPoint.ToString.
4. Once you’ve got that done, you should be able to accurately explain why you’re seeing this new, different output, in your own words. You are NOT required to write this down anywhere.