Module 02 Lab 01 Worksheet

C# Console Programming

# Overview

This lab is a continuation of the previous week's lab on developing C# applications. This week we are adding object-oriented program concepts, including:

* Classes
  + Methods
  + Properties
  + Constructors
  + Destructors
* Inheritance
* Abstract Classes
* Interfaces

|  |
| --- |
| **NOTE**: You should have a C# language reference handy. |

*It's important that you complete this lab before doing Lab 02.* You will be using the code you write in this lab for the next one.

|  |  |
| --- | --- |
| * Using the aspnet generator, create a console application called VarmintHunt. * Open it in VS Code * When prompted by VS Code, restore unresolved dependencies and add required assets to your project. * Define a new class called Varmint in a file called Varmint.cs. * Open Varmint.cs * Copy and paste the current Varmint class definition here. (5 pts.)   using System;  namespace VarmintHunt {  class Varmint {    } | 5 |
| Classes in C# are defined and instantiated similarly to classes in Java. There are one or two distinctions, which we'll cover in this lab.  Like Java, C# has access modifiers that specify the visibility of a class, method or property. C# adds two additional modifiers:   * internal - members can only be reached from within the same project * protected internal - similar to internal, except that classes that inherit from this class can also reach this class, even if they're in different projects.   Given that, what does the access modifier for the Varmint class mean? (5 pts.) | 5 |
| Varmints are hungry almost all the time. Their hunger is measured with a integer value, with 0 as 'not hungry' and larger number indicating greater hunger. With that in mind, add a property to the Varmint class called Hunger and make that property private.  Copy and paste the Hunger declaration here. (5 pts.)  private int Hunger {  get { return hunger; }  set { hunger = value; }  }  **NOTE**: In C#, properties have get() and set() methods. At least one of these is required but if not specified both are assumed and they work like Java class properties. One advantage of specifying these methods is that we can create *read-only* (get) or *write-only* (set) properties. | 5 |
| Varmints are also quite talkative and friendly. Add a method called Howdy() that uses Console.Writeline() to output a friendly greeting and report the Varmint's hunger level.  Copy and paste the Howdy method here. (5 pts.)  public void Howdy(){  Console.WriteLine("Hi! How are you?!")  } | 5 |
| Modify the Varmint constructor to take an integer variable hunger that has a default value of 0. When the constructor runs, the Hunger property is set and the message "Another Varmint has hatched!" should be output to the console.  **NOTE**: C# classes can also have destructors (similar to finalizers in Java), but since C# is garbage-collected (like Java), destructors are rarely if ever used.  Copy and paste your new constructor here. (5 pts.)  public Varmint(int value) {  Hunger = value;  Console.WriteLine("Another varmint has hatched");  } | 5 |
| Open Program.cs and create a new Varmint called fred with a hunger of 7.  Once you've made sure the code works, copy and paste the variable declaration here. (5 pts.)  Varmint fred = new Varmint(7); | 5 |
| Create a new class called VarmintHunter.  Modify the class definition to include:   * a private integer property called Photos (VarmintHunters hunt with cameras, not guns). * a constructor that it outputs the message "Be vewy, vewy quiet -- I'm hunting Varmints!" to the console and sets the value of Photos to 0. * A method called takePhoto() that increments Photos by 1. * A method called getPhotos() that outputs the number of photos taken by the given VarmintHunter   Copy and paste your completed class definition here. (5 pts.) | 5 |
| Write some code in Program.cs to test your VarmintHunter class and once you have it working, paste it here, along with the output. (10 pts.)  **Code**:  **Output**: | 10 |
| Abstract classes (ones that use the keyword abstract in their class definition) are used to define base classes in a class hierarchy. The main difference is that you can't create an instance of an abstract class, you can only use it for inheritance. In C#, a class is shown to be an inherited class with the notation on the class definition:  public class Cat : Mammal {  //methods and properties  }  If you want to extend or modify an inherited method, you need to use the override keyword in your method definition. In addition the method to be overridden must be defined in the abstract class with the virtual keyword:  public override void Speak() {  Console.Writeline("Meow!");  }  It turns out that Varmints are a subtype of the animal class Critters. Create an abstract class called Critter. Add a Howdy() method that outputs "Howdy, howdy, howdy!" to the console.  Modify the Varmint class to show the inheritance and then make such changes so that your code in Program.cs still works.  Copy and paste your Critter class definition. (5 pts.)  Copy and paste your new Varmint class definition. (5 pts.) | 10 |
| Try to create an instance of Critter. What is the error message from Intellisense? (5 pts.) | 5 |
| An interface is like an abstract class except it's even more abstract. Not only can't you create an instance of an interface, but the methods of an interface are empty. So when a class implements an interface, that class is obligated to implement all of the methods and properties of that interface. In a sense, an interface is a contract with the class that inherits from it. The other difference between interfaces and abstract classes is that a class can inherit from multiple interfaces.  An interface definition is the same as a class definition, except:   * It uses the interface keyword instead of class * The name traditionally starts with an I (ex. IAnimal) to indicate it's an interface. * The methods are displayed in a similar fashion to Java method signatures. (you don't need the override keyword) * Properties simply include get and/or set methods as appropriate.   Create a new interface called ICritter.  Modify the Varmint class to inherit from ICritter instead of Critter.  Test your code in Program.cs to make sure it still works.  Copy and paste your modified code here:  ICritter interface (5 pts.):  Varmint class (5 pts.): | 10 |
| Modify your Program.cs code to include the line  int[] trouble = new int[3];  trouble[3] = 5;  What's the problem with this new code? (5 pts.)  Does Intellisense have a problem with this? (5 pts.) | 10 |
| Compile and run the code.  What happens? (5 pts.) | 5 |
| Some errors are not caught by the IDE or the compiler but make your program do something impossible or illegal. We can catch these before they make our program crash, however, by using *exceptions*.  Surround the offending code with a try { } block, followed immediately by a catch { } block, like so:  try {  int[] trouble = new int[3];  trouble[3] = 5;  }  catch {  Console.WriteLine("An error has happened.");  }  Run the code again. What happens? (5 pts.) | 5 |
| This is a very simple example of *exception handling*. If your code is syntactically and logically correct, but under certain circumstance do something illegal or impossible (ex. Reading past the end of a file), it throws an exception. If the exception isn't caught your program will crash.  The try/catch pairs let us put a virtual airbag around our code. To get more useful error messages you can use the Exception class, like so:  catch(Exception ex) {  Console.WriteLine("An error occurred: " + ex.Message);  }  There are more specific exception classes that inherit from Exception:  catch(IndexOutOfRangeException ex){  Console.WriteLine("An index was out of range.");  }  You can even stack your catch blocks, from more to less specific. Whichever block catches it, the remaining catch blocks are skipped:  catch(IndexOutOfRangeException ex){  Console.WriteLine("An index was out of range.");  }  catch(Exception ex) {  Console.WriteLine("An error occurred: " + ex.Message);  }  There is also a finally block, which always runs, regardless if any of the preceding catch blocks trigger:  catch(IndexOutOfRangeException ex){  Console.WriteLine("ex.Message");  }  catch(Exception ex) {  Console.WriteLine("An error occurred: " + ex.Message);  }  finally  {  Console.WriteLine("The end of our try block. Clean-up in aisle 5!");  }  Copy the preceding stack of catch blocks into your Program.cs and run the code.  What error message appears? (5 pts.) | 5 |
| **Total** |  |

Complete this worksheet and submit it to your instructor.