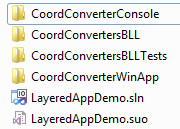
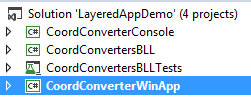
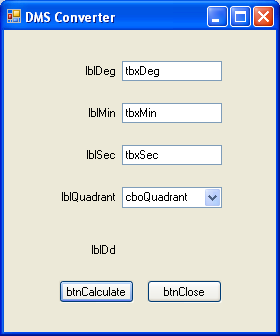
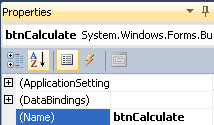
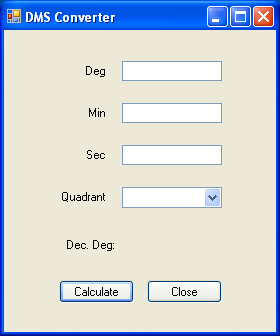
# C# Exercises: Advanced data types and operators, function overloads, layered architecture

In the first set of C# exercises you created console applications. The “startup” object / method in a console application is Program / Main respectively. That is, the method named Main in Program.cs is the first method to run when debugging the application in Visual studio and is the first method to run when executing the EXE assembly from the command line.

In addition to Main, you created other functions in Program.cs that contained business logic (e.g. Dms2Dd for converting DMS coordinates to decimal degree). You put tests for these functions in Main so the only way to test your business logic was to run the application. In the first set of exercises you did not write the code necessary to process command line arguments. If you had, you would have needed to move or comment out the tests in Main so they were not run every time the application was run. This is bad design. A better design is separating the presentation layer (Console project in this case), business logic, and testing so that they are independent. That is, the Console application passes command line arguments to the business logic and the tests can be run independently to test the business logic. In this exercise, you will create a layered application with business logic and tests in separate class libraries. You will also create two presentation layers: a Console project and a Windows Forms project that will reference and use the business layer.

1. Download and follow the instructions in <https://github.com/viljoed/gis4x07/raw/master/GIS4x07_ExerciseSetup.docx> to create a Git repository for this exercise.
2. Download and follow the instructions in <https://github.com/viljoed/gis4x07/raw/master/VS_SolutionSetup.docx> to create the Visual Studio solution for this exercise.
3. Replace / Rename from ExerciseSolutionName.sln to LayeredAppDemo.sln
4. Add a new Class Library project called CoordConvertersBLL to the src folder
5. Delete Class1.cs from the project
6. Add public Classes to the CoordConverterBLL project called DMSConverter and EnumsAndStructs
7. From EnumsAndStructs, remove the class definition (i.e. class statement and { })
8. In EnumsAndStructs, add a public enum called Quadrant that can has 4 possible values (N, S, E, W)
9. In EnumsAndStructs, add a public struct called DMSCoord that will contain four public members with “type name” pairs: int deg, int min, double sec, and Quadrant quadrant
10. Make sure the enum, struct, and class have pubic scope!
11. In the first set of exercises you created a function called Dms2Dd that took four parameters. In this exercise, you will add a public, non-static function called Dms2Dd to the DMSConverter class that has one parameter called dmsCoord of the type DMSCoord (the struct you created earlier) and returns a double.
12. In this function, simply add a statement to return 9999 so the project will build.
13. Press F6 or right-click and select Build to make sure this project currently builds.
14. Add a new Test Project called CoordConvertersBLLTests to the Solution
15. Add a new Console Application project to the Solution called CoordConverterConsole
16. Add a new Windows Forms Application project to the Solution called CoordConverterWinApp
17. Save all changes and exit from Visual Studio.
18. The contents of your src folder should now look something like the following  
    
19. Before going any further, it would be a good idea to get this Solution into version control. Add your exercise folder (e.g. BobMandDougM) to GitHub
20. Open LayeredAppDemo.sln
21. In the Solution Explorer you should see something like:  
      
    The Console and WinApp projects are both presentation layer projects. That is, they create assemblies that a user can execute interact with. The BLL project is the business layer project that will contain the business objects and processing for the Solution. The CoordConverterBLLTests project will contain the unit tests for the BLL project. In a test driven approach, the tests are written before or immediately after the bare minimum function definition. That is, a function definition that includes scope, type, name, parameters, and empty braces { } or a simple return statement inside the braces. That is the approach we will take here.
22. Right-click on the CoordConverterBLLTests project and add a Unit test.
23. This will create a test class called UnitTest1.cs. Rename this to DMSConverterTest.cs and click Yes to update references.
24. In CoordConverterBLLTests , add a Reference to CoordConverterBLL so CoordConverterBLLTests project is “aware” of the BLL project. Add a using statement to DMSConverterTest.cs. What is the significance of the reference and using statements?
25. Rename the test method to something like Dms2Dd\_ValidNcoord\_Calculated or Dms2Dd\_InvalidNcoord\_Returns9999. This follows the [recommended naming convention](http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html): MethodName\_StateUnderTest\_ExpectedBehaviour. Start with a tests for a valid arguments passed to Dms2Dd.
26. Each test will follow the Arrange – Act – Assert pattern. Arrange involves creating an instance of the class under test. Act calls the method under test. Assert is an Assert statement that checks expected vs actual values. One BLL method can have many test methods. Tests should have invalid and valid parameters to increase your confidence that the function is working as expected.
27. Once you have one test working, you can copy, paste, rename, and modify the test to create another.
28. Now your work involves alternating between the BLL method and running the tests until all tests pass.
29. Once all tests pass, then you will have some confidence that your business logic is robust and you can create a presentation layer. Open the CoordConverterWinApp project
30. Right-click on Form1.cs and select Rename and change its name to MainForm.cs. When you press Enter, Visual Studio will ask whether or not you want to rename all references to Form 1. Click Yes.
31. In the WinApp project, add a reference to the CoordConverterBLL project.
32. Right-click on CoordConverterWinApp and make it the StartUp project.
33. Double click MainForm.cs. This will open the form designer with a blank form.
34. Press F7 to switch to code mode and add a using at the top of MainForm.cs for CoordConvertersBLL.
35. Press Shift-F7 to switch back to design mode.
36. Under the View menu, select Toolbox. Make sure you “pin” it so it does not auto-hide  
    
37. Click the Form. By default, a Properties window should be displayed under the Solution Explorer. If it is not there, right-click on the form and select Properties.
38. Change the Text property to “DMS Converter”. Notice what happened?
39. Now add Labels, TextBoxes, a ComboBox, and two buttons to the form. Set the Name property of the controls to match what is shown below.  
      
    As an example, here is what the properties looks like for btnCalculate  
    
40. Please note that the above form control display text is not what is going to be displayed in the final form. The above is displayed this way to help you with control naming and position. The Text property of Labels and Buttons will control what is displayed. Fine final form should look something like the following:  
    
41. Double-click the Close button and add the following line of code to the method that was created:  
    Application.Exit();
42. Press F5 to run the application. Click Close to Exit the application.
43. Select the ComboBox. In the Properties window, scroll down to and click Items
44. Click the “…” beside Collection and then enter N, S, E, and W on separate lines and click OK.
45. Press F5 and select an item in the ComboBox.
46. Now you have the various interface elements of the Presentation Layer created. In general terms, here is what you need to do to connect this interface to your business logic. The Calculate button click will
    1. Get values entered in the text boxes using the Text property (e.g. tbxDeg.Text)
    2. Get the selected ComboBox value using cboQuadrant.SelectedItem.ToString();
    3. Ensure the coordinate values in the text boxes are numbers (HINT: int.TryParse, double.TryParse)
    4. Use the quadrant text value to set the appropriate Quadrant enum value
    5. Use the coordinate values and quadrant to create a DMSCoord value
    6. Create an instance of DMSConverter class
    7. Call the DMSConverter.Dms2Dd method with the DMSCoord value created above and set the lblDd Text property equal to the value returned (use .ToString()).

Once you have this Windows Forms Application working, this exercise will be considered complete.

OPTIONAL: Create the necessary code in the Console Application to have the business logic work from the command line with the following usage: CoordConverterConsole.exe <deg> <min> <sec> <quadrant>

The output will be simply the decimal degree value (e.g. 45.3641)