# Professional Software Engineering

Lecture 2: Exercises

## Exercise 1 Complex Numbers

Create a class to represent complex numbers in C#. The class should be created in a separate file, and have two properties of type double: the real and imaginary values ($Re(z)$ and $Im(z)$, respectively).

The class should accept both $Re(z)$ and $Im(z)$ in the constructor and contain a method to calculate the Euclidean norm of the complex number. The formula is presented in eq. (1), where $z^*$ is the complex conjugate of a complex number $z$.

$$\sqrt{zz^*} = \sqrt{Re(z)^2 + Im(z)^2} \tag{1}$$

After creating the class, create a new complex variable in your main function and output the norm to the console.

**Bonus**: Create a static method that initializes and returns a complex class object. The declaration of the method is provided below in italicized font (you can change the name of the method).

*public static Complex CreateComplexValue(double real , double imaginary);*

## Exercise 2 Convert To Complex

Create a static method in your complex class "*TryParseComplex*" that takes in a user's input (as a string) and converts it to a complex class object. The declaration of the method is provided below in italicized font.

*public static bool TryParseComplex(string userInput , out Complex complexUserInput);*

A step-by-step process of programming this method is presented below.

1. Check if the string is a valid complex number (an example of a valid complex number is "4.3+8.2*i*").
2. If the string is not valid return *false.*
3. If the string is valid extract the real and imaginary parts of the complex number.
4. Create a new instance of your complex class and initialize it with the real and imaginary values obtained in the previous step.
5. Save the complex class object in the variable *complexUserInput.*
6. Return *true*

**Hint**: It would be much easier (and cleaner) to move steps 1 and 3 into its own method. An example declaration is given below. Keep in mind that you have to reorganize your code to make this work, *i.e.* your function will not be structured in a similar way to the one described in the step-by-step process.

*private static bool IsValidComplex(string userInput , out double real , out double imaginary);*

**Note**: This exercise builds upon the class you created in Exercise 1.

## Exercise 3 Factorial

Write a C# program to take a non-negative number as an input from the console and returns it factorial. Factorial of a number is calculated as follow:

*n! = n*(n-1)! = n*(n-1)*...*3*2*1*

**Note:** Use a recursive function instead of loops.

## Exercise 4 Pass by value/ reference and Overloading

Create a simple class to store the point in a plane i.e. *(x,y)*. Create two *static* overloaded functions with *void* return type to replace one point by the other in the *main* program class i.e. reassign the new values from second point to first point. One of the function should use pass by value and other should use pass by reference. Test them by creating 2 point objects using instances of above class.
**Note:** Use '*this*' keyword to set the values of variable '*x*' and '*y*' in the constructor instead of '*get*' and '*set*'.
**Hint:** While testing, notice that the function which uses pass by value won't be able to replace the first point by other.

## Exercise 5 Inheritance

Create an abstract class to represent Vectors in C# with

1. an abstract method to get Norm. Norm of a 3D vector is defined as

$$\sqrt{x^2 \ + \ y^2 + z^2} \tag{2}$$

2. a public method to get DotProduct of two vectors and
3. a virtual method to print the vector to console.

Create two derived classes - one for 2D vectors and other for 3D vectors.
Implement constructors and member functions(methods) inherited from base class. Create a separate main program file to test the functionality of your classes by creating class objects and checking whether all the functions/methods are working correctly as intended or not.

**Hint:** DotProduct function from base class can have implementation for 2D vectors which can directly be inherited by 2D vector class but you will need to overload it for 3D vectors.
**Note:** Abstract and virtual methods must be overridden in derived class with their own implementations.