Programming .NET 4 With C# Lab Manual

Module 1: Introduction to .NET and C#

Lab 1.1: Hello World Program

1. Open Visual Studio 2013 by navigating the Start menu to Programs -> Microsoft Visual Studio 2013.
2. Once you have the IDE open with the Start page showing, create a new project using the "File | New | Project..." menu item.
3. When the dialog opens, select Visual C# from the list on the left, and choose **Console Application** from the list of project types. Give your project a suitable name (e.g. HelloWorld) and select some directory to hold the project.
4. Ensure the **Program.cs** file is open in the editor window; if it isn't then double click it in the Solution Explorer window inside Visual Studio.
5. Locate the Main method and add code to display a string to the console like so:

```
static void Main(string[] args) {
   Console.WriteLine("Hello, C#!");
}
```

6. Note the "Intellisense" feature that pops up candidate types and members. Press Tab to accept the current option.
7. Build the program by selecting **Build | Build Solution** in the menu. The output pane should indicate the successful completion of the build.
8. Select **Debug | Start without Debugging** from the menu (or Press Ctrl+F5) to run the program. A new console window will open and the program output will be displayed. Press any key to close the window.
9. To run the program under the debugger, select **Debug | Start Debugging** (or press F5), but the debugger closed the console window when the application finished.
10. Let's add a breakpoint to make it easier to understand the code. Place the caret on the **Console.WriteLine** line and press F9 (or click on the left margin where the line is located). A red circle should appear.
11. Run the program again (press F5). The debugger should stop in the breakpoint. You can examine the Stack window, and step execution with **Debug | Step Over** (F10) or **Debug | Step Into** (F11). To stop debugging before the program ends, select **Debug | Stop Debugging** (Shift + F5).
12. Remove all breakpoints.
13. Add code to display your name and address. Make sure it works as expected.

Module 2: C# Language Fundamentals

Lab 2.1: Basic Console I/O

1. Create a new Console application named **HelloPerson**.
2. In the Main method:
    a. Display a "What's your name?" prompt using **Console.WriteLine**.
    b. Get a string from the user using **Console.ReadLine**.
    c. Print to the console "Hello " with the user's name.
    d. Get a number from the user in the range 1-10. You can turn a string into an integer using the **int.Parse** static method.
    e. Display the user's name the number of times you received in (d). For each line, add a space before the name, like so: (example with n=3)

```
John
 John
  John
```

Lab 2.2: Simple Calculator

1. Create a new console application named **Calculator**.
2. Input two numbers of type double from the user and an operator (+,-,* or /). Calculate the result and display it. Make sure you reject a bad operator.

Lab 2.3: Guess My Number Game

1. Create a new console application named **GuessingGame**.
2. The computer should select a secret number in the range 1-100 like so:

```
int secret = new Random().Next(1, 100);
```

3. The user should try to guess the number. The program should respond with "too big" or "too small".
4. When the user finally guesses correctly, display the number of guesses she took. If it's more than 7, display a "you failed" message and show the correct number.

Lab 2.4: Command Line Arguments

1. Create a console application named **Quad**.
2. The application should use the first three parameters on the command line as coefficients to the quadratic equation a*x^2+b*x+c=0.
3. Make sure there are actually 3 arguments.
4. Convert the arguments to doubles using **double.TryParse**. Display errors if there are any.
5. Calculate the solutions to the equation and display them (there may be two, one or no solutions).

Lab 2.5: Using Loops

1. Create a new console application named **MulBoard**.
2. Display a multiplication board 10x10 by using two nested for loops. Make sure numbers are aligned to the right, by using formatting with something like "{0,4}".

Lab 2.6: Binary Displays

1. Create a new console application named **BinaryDisplay**.
2. Input an integer number from the user and display it in binary. Be as efficient as you can.
3. Calculate the number of "1"s in the number using the AND (&) operator and the right shift operator (>>). Display the result.

Lab 2.7: Using Loops II

1. Create a new console application named **DollarStairs**.
2. Input a number n from the user and display n stairs of $ signs, like so (n=5 in this example):

```
$
$$
$$$
$$$$
$$$$$
```

Module 3: Types

Lab 3.1: Creating Reference Types

1. Create a blank solution named **Accounts**.
2. Add a new project to the solution of type Class Library named **AccountsLib**.
3. Add a new class to the project named **Account**.
4. In the Account class:
    a. Create an internal constructor accepting an account id (int).
    b. Add a read only property named **ID** that returns that id.
    c. Add a **Deposit** method that allows depositing money to the account.
    d. Add a **Withdraw** method that allows withdrawing money from the account. Do not allow the account to go into overdraft.
    e. Add a read only property named **Balance** that should return the current balance.
    f. Add a **Transfer** method accepting another **Account** object and an amount, and make a transfer of money from the current instance to the instance passed as argument.
5. Create a static **AccountFactory** class that will be a factory for accounts. In this class:
    a. Add a static **CreateAccount** method that accepts an initial balance.
    b. The method should create a new account with a running id number and deposit the initial sum.
    c. Return the account to the caller.

6. Add a new console application project to the solution.
7. In the Main method:
    a. Create an account.
    b. Allow the user to deposit, withdraw or query its balance.
    c. Create another account.
    d. Perform a money transfer.
8. Test your code.

Lab 3.2: Creating Value Types

1. Create a new console application named **Rationals**.
2. Create a struct named **Rational** that holds a numerator and a denominator.
3. In the Rational type:
    a. Create a constructor that accepts two integers being the numerator and denominator.
    b. Create another constructor that accepts a single integer. The denominator should be set to 1.
    c. Add properties that return the numerator and denominator.
    d. Add a property that returns the value as a double.
    e. Add an **Add** method that adds two Rational objects. Make this method return a new Rational instance.
    f. Add a **Mul** method that multiplies Rational objects. Make this method return a new Rational instance.
    g. (*) Add a **Reduce** method to simplify the Rational object. This method should return void.
    h. (*) Override **ToString** and **Equals** and provide appropriate implementations.
4. In the Main method:
    a. Create some Rational objects, initialize with some values, and test the code you created to make sure all methods work as expected.

Module 4: Arrays, Collections and Strings

Lab 4.1: Working With Strings

5. Create a new console application named **Strings**.
6. In the Main method:
    a. Get a string from the user representing a sentence. If it's empty, exit the application.
    b. Split the sentence into separate words using the **String.Split** method.
    c. Display the number of words.
    d. Reverse the words and display the resulting array.
    e. Sort the strings with **Array.Sort** and display them in their sorted order.
    f. Repeat from (a) until an empty string is input.
7. Test the application with various strings.

Lab 4.2: Working With Arrays

8. Create a new console application named **TicTacToe**.
9. Add a new class called **TicTacToeGame**. This game should manage a simple TicTacToe game. Use the following guidelines:
    a. The game board should be a 3 by 3 matrix. Each cell should be of an enumerated type indicating X, O or empty.
    b. Add a method that displays the board appropriately.
    c. Add a method that sets a move in a specific cell.
    d. Check for move legality.
    e. If legal, make move.
    f. If move causes a game over (a draw, or someone wins), indicate that through an **IsGameOver** property.
10. In the Main method, create an instance of the **TicTacToeGame** class, and use it inside some loop until the game is over.
11. Test your game.

Lab 4.3: Working with Collections

1. Create a new console application named **Primes**.
2. Create a static method named **CalcPrimes** in the **Program** class that accepts two integers and returns an integer array. In the method:
    a. Calculate all the prime numbers in the range of the numbers passed as arguments. Collect all primes in an **ArrayList**.
    b. Return the result as an array of integers (hint: you can use ArrayList.CopyTo).
3. In the Main method:
    a. Accept two numbers from the console.
    b. Call **CalcPrimes** with the numbers, and display the results.
4. Test the application.

Module 5: Inheritance & Polymorphism

Lab 5.1: Inheritance & Polymorphism

1. Create a new project named **ShapeLib** as a class library. Make sure the "Create directory for solution" is checked, and name the solution **Shapes**.
2. Create an abstract class named **Shape**. The class should have the following members:
    a. A **Color** property of type **ConsoleColor**.
    b. A constructor accepting a color and setting the color.
    c. A default constructor that uses a white color.
    d. A virtual **Display** method that changes the current console color to the Color property value.
    e. An abstract read only property named **Area**.
3. Create a class named **Rectangle** that inherits from Shape:

   a. Create an appropriate constructor that accepts a width and height of the rectangle.
   b. Add relevant properties.
   c. Implement the abstract property inherited from **Shape**.
   d. Override the **Display** method and implement by displaying the rectangle width and height.
4. Create an **Ellipse** class inheriting from Shape and implement as appropriate.
5. Create a **Circle** class inheriting from **Ellipse** and implement as appropriate.
6. Add a new console application project to the solution, named **ShapesApp**.
7. Add a reference to the **ShapesLib** project you just created.
8. Create a class named **ShapeManager** that has the following members:
   a. An **ArrayList** field holding shapes.
   b. A public **Add** method that accepts a Shape and adds it to the collection.
   c. A public **DisplayAll** method that calls **Display** and **Area** for all shapes in the collection.
   d. A public read only indexer that returns a shape in a specified index.
   e. A read only property named **Count** returning the total number of managed shapes.
9. In the Main method:
   a. Create an instance of **ShapeManager**.
   b. Add several different shapes to the ShapeManager you just created.
   c. Call **DisplayAll** and make sure you get the expected result.

Lab 5.2: Interfaces

1. Continue from the previous exercise.
2. In the **ShapesLib** project:
   a. Define an interface named **IPersist** like so:

```
public interface IPersist {
   void Write(StringBuilder sb);
}
```

   b. Implement the interface in the Rectange and Ellipse classes like so:
      i. Use the **StringBuilder.AppendLine** method to add the width and height of the rectangle.
      ii. Implement similarly in Ellipse.
3. In the **ShapeManager** class, add a public method called **Save** that accepts a StringBuilder and calls **Write** on all shapes that implement IPersist.
4. In the Main method, call the **Save** method and display the resulting **StringBuilder** using its **ToString** method.
5. Implement the standard interface **IComparable** in the Rectangle and Ellipse classes.
6. Test your implementations.

Module 6: Exceptions

Lab 6.1: Catching and Throwing Exceptions

1. Continue from exercise 3.1.
2. Currently, there is no check when depositing and withdrawing amounts to make sure they're positive.
3. Add checks that make sure the amount is positive, and if not, throw an **ArgumentOutOfRangeException** object.
4. In the Main method, catch that exception and display appropriate message to the user.
5. Do you have to make similar checks in the **Transfer** method?
6. Modify the Transfer method like so:
    a. Add some code at the end of the method that displays the fact that a transfer attempt has been made.
    b. Add code to display the amount of money before and after the transfer in the current account.
7. The code you just added needs to run whether there is an exception or not. Add a **finally** block into the **Transfer** method that ensures this.
8. Test your code by calling **Transfer** with good and bad values.

Lab 6.2: Custom Exceptions

1. Continue from the previous exercise.
2. Create a new class named **InsufficientFundsException** deriving from **Exception**. You can use the "exception" code snippet in Visual Studio.
3. Throw this exception in the **Withdraw** method if withdrawing would cause the balance to become negative.
4. Catch the exception in Main and test it.