Programming .NET 4 With C# Lab Manual

# Module 1: Introduction to .NET and C#

## Lab 1.1: Hello World Program

1. Open Visual Studio 2010 by navigating the Start menu to Programs -> Microsoft Visual Studio 2010.
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#!");

}

1. Note the “Intellisense” feature that pops up candidate types and members. Press Tab to accept the current option.
2. Build the program by selecting **Build | Build Solution** in the menu. The output pane should indicate the successful completion of the build.
3. 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.
4. 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.
5. 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.
6. 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).
7. Remove all breakpoints.
8. 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:
   1. Display a “What’s your name?” prompt using **Console.WriteLine**.
   2. Get a string from the user using **Console.ReadLine**.
   3. Print to the console “Hello “ with the user’s name.
   4. Get a number from the user in the range 1-10. You can turn a string into an integer using the **int.Parse** method.
   5. 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);

1. The user should try to guess the number. The program should respond with “too big” or “too small”.
2. 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:
   1. Create an internal constructor accepting an account id (int).
   2. Add a read only property named **ID** that returns that id.
   3. Add a **Deposit** method that allows depositing money to the account.
   4. Add a **Withdraw** method that allows withdrawing money from the account. Do not allow the account to go into overdraft.
   5. Add a read only property named **Balance** that should return the current balance.
   6. 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:
   1. Add a static **CreateAccount** method that accepts an initial balance.
   2. The method should create a new account with a running id number and deposit the initial sum.
   3. Return the account to the caller.
6. Add a new console application project to the solution.
7. In the Main method:
   1. Create an account.
   2. Allow the user to deposit, withdraw or query its balance.
   3. Create another account.
   4. 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:
   1. Create a constructor that accepts two integers being the numerator and denominator.
   2. Create another constructor that accepts a single integer. The denominator should be set to 1.
   3. Add properties that return the numerator and denominator.
   4. Add a property that returns the value as a double.
   5. Add an **Add** method that adds two Rational objects. Make this method return a new Rational instance.
   6. Add a **Mul** method that multiplies Rational objects. Make this method return a new Rational instance.
   7. (\*) Add a **Reduce** method to simplify the Rational object. This method should return void.
   8. (\*) Override **ToString** and **Equals** and provide appropriate implementations.
4. In the Main method:
   1. 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

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

## Lab 4.2: Working With Arrays

1. Create a new console application named **TicTacToe**.
2. Add a new class called **TicTacToeGame**. This game should manage a simple TicTacToe game. Use the following guidelines:
   1. The game board should be a 3 by 3 matrix. Each cell should be of an enumerated type indicating X, O or empty.
   2. Add a method that displays the board appropriately.
   3. Add a method that sets a move in a specific cell.
   4. Check for move legality.
   5. If legal, make move.
   6. If move causes a game over (a draw, or someone wins), indicate that through an **IsGameOver** property.
3. In the Main method, create an instance of the **TicTacToeGame** class, and use it inside some loop until the game is over.
4. 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:
   1. Calculate all the prime numbers in the range of the numbers passed as arguments. Collect all primes in an **ArrayList**.
   2. Return the result as an array of integers (hint: you can use ArrayList.CopyTo).
3. In the Main method:
   1. Accept two numbers from the console.
   2. 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:
   1. A **Color** property of type **ConsoleColor**.
   2. A constructor accepting a color and setting the color.
   3. A default constructor that uses a white color.
   4. A virtual **Display** method that changes the current console color to the Color property value.
   5. An abstract read only property named **Area**.
3. Create a class named **Rectangle** that inherits from Shape:
   1. Create an appropriate constructor that accepts a width and height of the rectangle.
   2. Add relevant properties.
   3. Implement the abstract property inherited from **Shape**.
   4. 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:
   1. An **ArrayList** field holding shapes.
   2. A public **Add** method that accepts a Shape and adds it to the collection.
   3. A public **DisplayAll** method that calls **Display** and **Area** for all shapes in the collection.
   4. A public read only indexer that returns a shape in a specified index.
   5. A read only property named **Count** returning the total number of managed shapes.
9. In the Main method:
   1. Create an instance of **ShapeManager**.
   2. Add several different shapes to the ShapeManager you just created.
   3. 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:
   1. Define an interface named **IPersist** like so:

public interface **IPersist** {

   void Write(**StringBuilder** sb);

}

* 1. Implement the interface in the Rectange and Ellipse classes like so:
     1. Use the **StringBuilder.AppendLine** method to add the width and height of the rectangle.
     2. Implement similarly in Ellipse.

1. In the **ShapeManager** class, add a public method called **Save** that accepts a StringBuilder and calls **Write** on all shapes that implement IPersist.
2. In the Main method, call the **Save** method and display the resulting **StringBuilder** using its **ToString** method.
3. Implement the standard interface **IComparable** in the Rectangle and Ellipse classes.
4. 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:
   1. Add some code at the end of the method that displays the fact that a transfer attempt has been made.
   2. 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.

# Module 7: Generics

## Lab 7.1: Generic Interface Implementation

1. Create a new console application named **CustomersApp**.
2. Create a new class, called **Customer** with the following properties: Name (string), ID (int), Address (string).
3. Implement the interface **IComparable<Customer>** by using the **Name** property only in a case insensitive way.
4. Implement the interface **IEquatable<Customer>** by comparing **Name** and **ID** of the customers.
5. Create a **Customer** array, display it, then use **Array.Sort** to sort the array and display the sorted results.
6. Create a new class named **AnotherCustomerComparer** that implements the **IComparer<Customer>** interface. The implementation should compare IDs only.
7. Test the comparer by passing it to **Array.Sort** call.

## Lab 7.2 (\*): Generic Class

1. Create a new console application named **GenericApp**.
2. Create a generic interface named IMultiDictionary<K,V> defined as follows:

public interface **IMultiDictionary**<K, V> {

   void Add(K key, V value);

   bool Remove(K key);

   bool Remove(K key, V value);

   void Clear();

   bool ContainsKey(K key);

      bool Contains(K key, V value);

**IEnumerable**<K> Keys { get; }

**IEnumerable**<V> Values { get; }

      int Count { get; }

}

1. Create a generic class named **MultiDictionary** with two generic arguments named K and V. This collection class will be similar to a dictionary, but the key does not have to be unique. Multiple values may be attached to a single key by storing the values in a **LinkedList** instance.
2. The class should implement the **IMultiDictionary<K,V>** interface as well as the **IEnumerable<KeyValuePair<K,V>>** interface.
3. The internal dictionary will be of type **DictionaryLinkedList<V>>**.
4. Implement the methods with these guidelines:
   1. The **Add** method should create a new **LinkedList** instance if the key does not exist, or add an item with the **AddLast** instance method.
   2. The **Remove** method with a key only should remove all values with that key (hint: Dictionary can do this). **Remove** with a specific value should remove just this value. Both methods return true on success.
   3. The **Keys** property should return a collection of all keys.
   4. The **Values** property should return a collection of all values.
   5. **Clear** should remove all items from the collection.
   6. The **Count** property should return the total items in the collection.
5. Implement the **IEnumerable<KeyValuePair<K,V>>** interface appropriately.
6. In the Main method, create an instance of **MultiDictionary<int, string>** and add the following pairs: { 1, “one” }, { 2, “two” }, { 3, “three” }, { 1, “ich” }, { 2, “nee” }, { 3, “sun” }. (The strange strings are numbers are in Japanese)
7. Test the various methods.

# Module 8: Reflection and Attributes

## Lab 8.1: Dynamic Invocation

1. Create a new console application named **DynInvoke**.
2. Create 3 classes named A, B, C, all with a method called **Hello**, which accepts a string and returns a string.
3. Implement the methods by returning “Hello “ + the input string for A, “Bonjour “ + input string for B, and “Nihau “ + the input string for C.
4. In the **Program** class, create a static method named **InvokeHello**, accepting an Object and a String.
5. Implement the method by looking for the “Hello” method and dynamically invoke it with the string argument provided.
6. In the Main method, create an instance for A, B and C and call **InvokeHello** with each object and some string.
7. Make sure you understand the results.

## Lab 8.2: Custom Attributes

1. Create a new console application project and name it **AttribApp**.
2. Create a new attribute class named **CodeReviewAttribute**.
3. The class should contain 3 properties: the reviewer name, the review date and whether the code is approved.
4. Add a constructor that accepts all 3 properties. Make sure they are also accessible as standalone properties.
5. Add the **AttributeUsage** attribute to the class to allow the attribute on classes and structs only (multiple usages are allowed).
6. Add 3 classes named A, B, C and decorate them with the **CodeReviewAttribute** with various parameters.
7. Create a method called **AnalayzeAssembly** which should receive an **Assembly** object as input and return whether or not all reviewed types are approved.
8. The method should traverse all types (**Assembly.GetTypes** or **Assembly.GetExportedTypes**), looking for the **CodeReviewAttribute**. If found, it should output to the console all the review details. Eventually it should return the correct value as described.
9. Call the methods from the **Main** method, passing the current assembly (**Assembly.GetExecutingAssembly**) and look at the results.
10. What happens if you pass a reference to some other Assembly to the **AnalyzeAssembly** method? Try it.

# Module 9: Delegates and Events

## Lab 9.0 Fun with Delegates

1. Define an array of strings holding several strings in different lengths
2. In the next section, we will define several **Test functions**:
   1. Test functions take a single string argument
   2. Test functions return Boolean
   3. Each test function has a different logic that it tests
   4. What C# type best fits these definitions?
3. Write a test function that returns true only if the string is longer than 6 chars
4. Write a test function that returns true only if the string starts with a letter between A-K
5. Write a test function the returns true only if the string contains the letters E or O
6. Now Add to the main program an array of test functions, with delegates to the 3 test functions written above
7. Add a **WriteSomeStrings** method that takes a string enumerable, a console color and a test function, and uses the color to display all the strings in the enumerable that the test function returns true on.
8. In the main program, call the **WriteSomeStrings** for each of the test functions, each time with a different console color.

## Lab 9.1: Using Delegates

1. Continue from exercise 7.1.
2. Define a delegate named **CustomerFilter** that accepts a **Customer** object and returns a Boolean.
3. Add a static method to the Program class named **GetCustomers** that accepts a collection of customers and an instance of the **CustomFilter** delegate and returns a collection of Customers.
4. This method should use the supplied delegate to filter the customer list so that only certain customers are returned.
5. In the Main method:
   1. Create a list or array of Customer objects.
   2. Create a delegate of type **CustomerFilter** that should return a Customer if its name starts with the letters A-K. Use a separate method to implement the delegate.
   3. Call **GetCustomers** with the delegate you created in (b) and display the result.
   4. Create another such delegate that returns all customers whose names begin with the letters L-Z. Use an anonymous delegate.
   5. Call **GetCustomers** again with the new delegate and display the results.
   6. Create another such delegate that returns all customers whose ID is less than 100. Use a lambda expression.
   7. Call **GetCustomers** again with the new delegate and display the results.

## Lab 9.2: Publisher – Subscriber Pattern

1. In this lab, you will implement the publisher/subscriber pattern.
2. Create a new console application named **MailSystem**.
3. Create a class called **MailManager**. This class should expose an event called **MailArrived** based on the **EventHandler<T>** delegate where T is a new class called **MailArrivedEventArgs**.
4. The **MailArrivedEventArgs** should expose two read only properties called **Title** and **Body**, which is the data the mail message exposes.
5. Create a protected virtual method called **OnMailArrived**, taking a **MailArrivedEventArgs** argument. Inside, raise the event.
6. Create a simple method called **SimulateMailArrived**, that calls **OnMailArrived** with some dummy data.
7. In the Main method, create an instance of **MailManager** and connect to the **MailArrived** event.
8. In the handler, output the title and body to the console.
9. Call the **SimulateMailArrived** to check the event connection.
10. (\*)Create a **System.Threading.Timer,** and in the timer callback call **SimulateMailArrived** every 1 second. Call **Thread.Sleep** in the main thread to keep the application alive, or call **Console.ReadLine**.
11. Make sure you get the expected results.

# Module Bonus: Extension Methods

## Lab B.1: Extend int

1. Write an extension method to int, called **Normalize**, that takes an upper limit and lower limit and returns the closest number to the int that is between the boundaries. So:
   1. 12.**Normalize**(10, 20) will return 12 because 12 itself is inside the boundaries
   2. 22. **Normalize** (10, 20) will return 20 because 22 is above the boundaries
   3. 5. **Normalize** (10, 20) will return 10 because 5 is below the boundaries
2. Create an array of random numbers
3. Loop over the array and display the original number and the normalized number

## Lab B.2 Extend IEnumerable<T>

1. Write an extension method to IEnumerable<T> called **TakeFirst**(int n) that returns the first n items of the enumerable
2. Write an extension method to IEnumerable<T> called **Reverse**() that returns the enumerable in reverse order

# Module 10: Resource Management

## Lab 10.1: The Dispose Pattern

1. Open the solution **Jobs** from the **Start** folder in this lab.
2. General note: you’ll have to run this app from the command line.
3. Implement the constructor of the **Job** class, that accepts a string called “**name”**:
   1. Call **NativeJob.CreateJobObject** passing **IntPtr.Zero** and the **name** argument. Store the returning handle in **\_hJob**.
   2. If the handle is zero (**IntPtr.Zero**), throw an **InvalidOperationException**.
   3. Create the **\_processes** object.
4. Implement the **IDisposable** interface on the **Job** class. Use the following guidelines:
   1. Make use of the Dispose pattern, i.e. create a **Dispose(bool)** method as discussed in the course.
   2. Make sure calling **Dispose** multiple times is harmless, while calling anything substantial if the object is disposed throws a **ObjectDisposedException**.
5. Implement the Kill method by calling **NativeJob.TerminateJobObject**.
6. In the Main method, create a Job object, and assign some processes to it (Use **Process.Start** to create some simple processes, such as “notepad” or “calc”).
7. Call **Console.ReadLine** and after the user hits <enter> kill all processes in the job using the **Kill** instance method.

# Module 11: Namespaces & Assemblies

## Lab 11.1: Assembly Probing

1. Continue from exercise 5.1 or 5.2. Make sure the application runs correctly.
2. Move the **ShapesLib.Dll** file from its location to a subfolder named **ShapesLib** (create it). Double click the **ShapesApp.Exe** file in Windows Explorer. Does the application run correctly? Why?
3. Rename the **ShapesLib** folder to **Bin**. Double click the **ShapesApp.Exe** file again. Does it run? Why?
4. Let’s see if we can get some help in figuring out why the application fails to run. Open the Visual Studio command prompt (Start -> Programs -> Microsoft Visual Studio 2010 -> Visual Studio Tools). If you’re running on Vista or Windows 7, right click the shortcut and select “Run as administrator”.
5. Type **fuslogvw.exe** in the command window.
6. Click the “Settings…” button and select “Log binding failures to disk”. Click OK.
7. Run **ShapesApp.Exe** again. Click “Refresh”. You should see a log showing where Fusion looked for the **ShapesLib** assembly.
8. Double click the entry to view the details.
9. Let’s create a configuration file and fix this issue.
10. Right click the project and select “Add New Item…”, select “General” on the left and “Application Configuration File” on the right. Click OK.
11. Enter the following XML:

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

  <runtime>

    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

      <probing privatePath="Bin" />

    </assemblyBinding>

  </runtime>

</configuration>

1. Run the application again. Does it work now?

# Module 12: Advanced Language Constructs

## Lab 12.1: Iterators

1. Revisit exercise 7.2. Implement the **IEnumerable<KeyValuePair<K,V>>** interface using the yield keyword.

## Lab 12.2: LINQ

1. Create queries and display results for the following:
   1. Display all the public interface names in the mscorlib assembly and the number of methods in each type. Sort by type name.
   2. Display all processes running on your system (process name, id and start time), whose thread count is less than 5. Sort by process id.
   3. (\*) Extend (b) by grouping the processes by their base priority.
   4. Display the total number of threads in the system.
2. (\*) Create an extension method that extends object with a method called **CopyTo**, that copies all public properties from this object to another object passed as argument. Make sure only writable properties are written and readable properties are read. Use LINQ.

## Lab 12.3: LINQ to XML

1. Create a new Console application named **XLinq**.
2. (\*) In the Main method, create an **XElement** object encapsulating the following information: all the classes (not structs or interfaces) in the mscorlib assembly that are public. For each class, add elements for each public instance property with its type and name, and all public instance methods with their name (not including inherited ones that are not overridden), return type and parameters (type and name). The resulting XML should look something like this:

  <Type FullName="System.Exception">  
    <Properties>  
      <Property Name="Message" Type="System.String" />  
      <Property Name="Data" Type="System.Collections.IDictionary" />  
      <Property Name="InnerException" Type="System.Exception" />  
      <Property Name="TargetSite" Type="System.Reflection.MethodBase" />  
      <Property Name="StackTrace" Type="System.String" />  
      <Property Name="HelpLink" Type="System.String" />  
      <Property Name="Source" Type="System.String" />  
    </Properties>  
    <Methods>  
      <Method Name="GetBaseException" ReturnType="System.Exception">  
        <Parameters />  
      </Method>  
      <Method Name="ToString" ReturnType="System.String">  
        <Parameters />  
      </Method>  
      <Method Name="GetObjectData" ReturnType="System.Void">  
        <Parameters>  
          <Parameter Name="info" Type="System.Runtime.Serialization.SerializationInfo" />  
          <Parameter Name="context" Type="System.Runtime.Serialization.StreamingContext" />  
        </Parameters>  
      </Method>  
      <Method Name="GetType" ReturnType="System.Type">  
        <Parameters />  
      </Method>  
    </Methods>  
  </Type>

1. (\*) Create the following queries on the resulting XML:
   1. List all types that have no properties. Order them by name. Also display how many are there.
   2. Count the total number of methods, not including inherited ones.
   3. Do some more statistics: how many properties are there? What is the most common type as a parameter? Create a new XML report for the results.
   4. Sort the types by the number of methods in descending order. For each get the number of properties and the number of methods. Create a new XML for the results.
   5. Group all the types by the number of methods, in descending order based on the number of methods. Within each group sort by type name in ascending order.

# Module 13: Introduction to C# 4.0

## Lab 13.1: Dynamic Object Provider

1. In this exercise you’ll implement a dynamic object that handles its own dispatch.
2. Open Visual Studio 2010 and create a new C# Console Application project named “DynamicXml”.
3. Add a new class named “DynamicXElement”. Make it inherit from **DynamicObject** (you’ll have to add a using directive to **System.Dynamic**).
4. This class should be able to handle accessing XML elements by using a property and an indexer. For example, consider the following XML data about planets:

<Planets>

  <Planet>

    <Name>Mercury</Name>

  </Planet>

  <Planet>

    <Name>Venus</Name>

  </Planet>

  <Planet>

    <Name>Earth</Name>

    <Moons>

      <Moon>Moon</Moon>

    </Moons>

  </Planet>

  <Planet>

    <Name>Mars</Name>

    <Moons>

      <Moon>Phobos</Moon>

      <Moon>Deimos</Moon>

    </Moons>

  </Planet>

  <!-- other data -->

</Planets>

With this data, the following code should compile ok and produce the results shown after the code:

dynamic planets = **DynamicXElement**.Create(**XElement**.Load("planets.xml"));

var mercury = planets.Planet;       // first planet

**Console**.WriteLine(mercury);

var venus = planets["Planet", 1];   // second planet

**Console**.WriteLine(venus);

var ourMoon = planets["Planet", 2].Moons.Moon;

**Console**.WriteLine(ourMoon);

var marsMoon = planets["Planet", 3]["Moons", 0].Moon;   // mars second moon

**Console**.WriteLine(marsMoon);

Results:

Mercury

Venus

Moon

Phobos

1. Add a private readonly field to the class named **\_element** of type **XElement**.
2. Add a private constructor that accepts an **XElement** and initializes the above private field.
3. Add a static factory method named **Create** that accepts an **XElement** and returns a **dynamic**, calling the constructor appropriately.
4. Override the **TryGetMember** method that should use the **XElement.Element** method to get to the required item, returning it as a new **DynamicXElement**.
5. Override the **TryGetIndex** method as follows:
   1. Check to make sure the **indexes** array has exactly two elements and that their types are string and int, respectively.
   2. Get the immediate children of the current element using the **XElement.Elements** method.
   3. The result of the method should be the element in the supplied index position (second indexer).
6. Override the **ToString** method that should return the **Value** property for the XElement. This will help with things like **Console.WriteLine**.
7. Create a test XML and some test code (you can use the above example) and make sure everything works correctly.
8. Make any other enhancement you want, such as allowing modifications to the data.

# Module 14: Data Streams & Files

## Lab 14.1: Accessing Files

1. Create a new console application named **Personnel**.
2. Create a file manually (e.g. with notepad) that has lines of names (first and last), like so:

Bart Simpson

Peter Parker

Clark Kent

Etc.

1. Write a method that reads all data from the file (you can use a StreamReader) and places it in a List of strings.
2. Display the list.

## Lab 14.2: File Manipulation

1. Create a console application named **FileFinder**.
2. Use the first parameter from the command line as a directory path, and search for all files that contain a string passed as the second parameter to the command line.
3. Display a list of all those files and their length.

# Module 16: Introduction to Threading

## Lab 16.1: Simple Threads

1. Create a console application named **SimplePrimes**.
2. Create a static method named **CalcPrimes**, that returns the number of prime numbers within a range of numbers given as arguments (first and last).
3. Time the method using some fixed range (say 3 to 5000000) using the **System.Diagnostics.Stopwatch** class.
4. Create a multithreaded version of **CalcPrimes**, with the number of threads as a parameter.
5. The multithreaded version should split the range into more or less equal chunks and process them concurrently.
6. Try various thread numbers and measure elapsed time. What is the optimum number of threads?

## Lab 16.2 (\*): Limited Queue

1. In this lab, you’ll use a semaphore to control a limited queue. Create a new Console Application named **Queues**.
2. Create a class called **LimitedQueue<T>** that uses a Queue<T> underneath. It should expose an **Enque** and **Deque** methods.
3. These methods should use a **Semaphore** created in the constructor (accepting a maximum queue size).
4. Build some test code that adds and removes items from the queue from various threads (you can use the thread pool) randomly so that at each point the queue is no larger than the maximum specified at creation time.

## Lab 16.3 (\*): Inter-process Synchronization

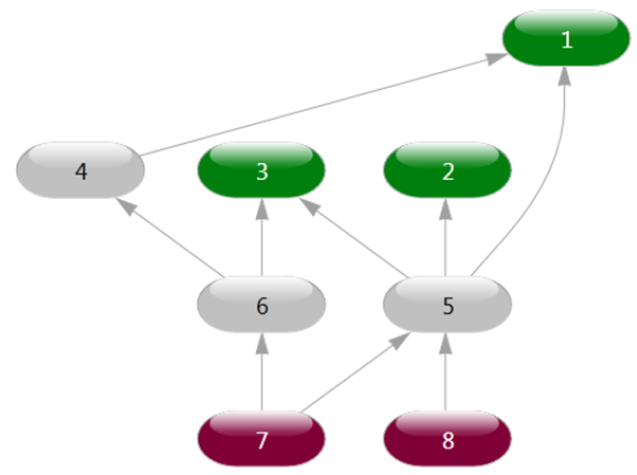
1. In this lab you will synchronize two processes accessing a shared file using a Mutex.
2. Create a new console or WinForms application named **SyncDemo**.
3. Create a **Mutex** object with the name “SyncFileMutex”, starting in the non-signaled state.
4. Open or create a file named “data.txt” in the “c:\temp” folder (create the folder using Windows Explorer or in code if it does not exist).
5. Create a loop that writes 10000 times a string that includes the process identifier (look at **System.Diagnostics.Process** class), while providing a **WaitOne**/**ReleaseMutex** around the code.
6. Run two instances of the application simultaneously and check the resulting file.
7. Comment out the mutex operations and rerun. Make sure you understand the results.

## Lab 16.4: Using the Parallel Class

1. Create a new Console application named **ParallelPrimes**.
2. Create a static method called **CalcPrimes** that accepts a first and last number and a maximum degree of parallelism (an integer) and returns a list of prime numbers in that range. Use the **Parallel** class. Make sure you add items to the collection in a thread-safe way. Use a lambda expression to execute the body of the parallel loop.
3. Call **CalcPrimes** from the Main method and compare its performance with a sequential implementation, by creating a **ParallelOptions** instance and setting its **MaxDegreeOfParallelism** property to the passed number.

## Lab 16.5: Tasks

1. Create a new console application named **ProjectBuilder**.
2. Suppose you have “projects” that need building, but they have dependencies, like so:

  
Read: project 2 depends on project 1, project 6 depends on projects 3 and 4, etc.

1. Write code that “builds” these projects (sleep for one second for each build step) sequentially.
2. Write code that builds concurrently using tasks, **Task.ContinueWith** and **TaskFactory.ContinueWhenAll**.

# Appendix: Operator Overloading

1. Continue from exercise 3.2.
2. Add operators for +,-,\*,/ for the Rational type.
3. Add casting operator to double and from an integer.
4. Test your code.