Advanced .NET with C# - Lab Manual

Advanced .NET with C# - Lab Manual

Module 1: Reflection and Code Generation

Lab 1.1: 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 1.2: Custom Attributes

1. Create a new console application project and name it **AttribDemo**.
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 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.

Lab 1.3: Create 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

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

Module 2: Generics

Lab 2.1: 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. In **Main**, Create a **Customer** array (or a List<>), display it, then use **Array.Sort** (or **List<>.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 2.2: 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);
   void CreateNewValue(k key);
   bool Remove(K key);
   bool Remove(K key, V value);
   void Clear();
   bool ContainsKey(K key);
   bool Contains(K key, V value);
   ICollection<K> Keys { get; }
   ICollection<V> Values { get; }
   int Count { get; }


}
```

3. 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<V>** instance.
4. The class should implement the **IMultiDictionary<K,V>** interface as well as the **IEnumerable<KeyValuePair<K,V>>** interface.
5. The internal dictionary will be of type **Dictionary<K, LinkedList<V>>**.
6. Implement the methods with these guidelines:
    a. The **Add** method should create a new **LinkedList** instance if the key does not exist, or add an item with the **AddLast** instance method.
    b. The **CreateNewValue** method should create a new value of type V and add it to the key. If the key doesn't exist in the collection, the method should add the key first.
    c. 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.

    d. The **Keys** property should return a collection of all keys.
    e. The **Values** property should return a collection of all values.
    f. **Clear** should remove all items from the collection.
    g. The **Count** property should return the total items in the collection.
    h. The key must have the custom attribute **[KeyAttribute]**, if not the method add should throw exception.
7. Implement the **IEnumerable<KeyValuePair<K,V>>** interface appropriately.
8. 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 in Japanese)
9. Test the various methods.
10. **Strict** the dictionary only for Value types.

Module 3: Advanced Delegates and Events

Lab 3.1: Using Delegates

1. Continue from exercise 2.1.
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:
    a. Create a list or array of Customer objects.
    b. 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.
    c. Call **GetCustomers** with the delegate you created in (b) and display the result.
    d. Create another such delegate that returns all customers whose names begin with the letters L-Z. Use an anonymous delegate.
    e. Call **GetCustomers** again with the new delegate and display the results.
    f. Create another such delegate that returns all customers whose ID is less than 100. Use a lambda expression.
    g. Call **GetCustomers** again with the new delegate and display the results.

Lab 3.2: 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.

Lab 3.3: Asynchronous Delegates

1. Create a new Windows Forms application called **AsyncDemo**.
2. Add two textboxes, a list box and a button.
3. Create a delegate that accepts two `int` arguments and returns `IEnumerable<int>`.
4. Create a method with the prototype of the above delegate, called **CalcPrimes**, and implement it to return the prime numbers in the range given by the two arguments.
5. When the button is clicked, create an instance of the delegate and call it asynchronously.
6. Harvest the results using a callback notification and update the list box with the results.
7. Make sure that during the calculation the user interface remains responsive.

Module 4: C# 3.0 & LINQ

Lab 4.1: LINQ to Objects

1. Create queries and display results for the following:
    a. Display all the public interface names in the mscorlib assembly and the number of methods in each type. Sort by type name.
    b. Display all processes running on your system (process name, id and start time), whose thread count is less than 5. Sort by process id.

c. (*) Extend (b) by grouping the processes by their base priority.
d. 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 4.2: 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>
```

3. (*) Create the following queries on the resulting XML:
    a. List all types that have no properties. Order them by name. Also display how many are there.
    b. Count the total number of methods, not including inherited ones.

c. Do some more statistics: how many properties are there? What is the most common type as a parameter?
d. 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.
e. 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.

Lab 4.3: LINQ to Entities

1. Create a new console or Winforms or WPF application.
2. Connect to some database (such as Northwind), and do some queries with LINQ.

Module 5: Managing Resources

Lab 5.1: 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"**:
    a. Call **NativeJob.CreateJobObject** passing **IntPtr.Zero** and the **name** argument. Store the returning handle in **_hJob**.
    b. If the handle is zero (**IntPtr.Zero**), throw an **InvalidOperationException**.
    c. Create the **_processes** object.
4. Implement the **IDisposable** interface on the **Job** class. Use the following guidelines:
    a. Make use of the Dispose pattern, i.e. create a **Dispose(bool)** method as discussed in the course.
    b. 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 6: Processes, AppDomains and Threads

Lab 6.1: Dynamic Assembly Unloading

1. In this lab you will create a second AppDomain in a process, and demonstrate the ability to unload assemblies by unloading the AppDomain.

2. Create a new Class Library project. Name it **SomeBigLib**
3. Add a new class, call it **TestBig**.
4. Add a method called **Hello**, taking a string and returning a string.
5. Implement the method by returning the text “Hello “, appending the input string.
6. Add a finalizer and a constructor to the class and display some text to that affect.
7. Add a console application project to the existing one. Reference the **SomeBigLib** assembly.
8. In the Main method, create a new AppDomain and call it “Second”.
9. Call the new AppDomain’s **DoCallback** method passing some delegate (you can use an anonymous delegate for this).
10. Implement the delegate method by creating an instance of the **TestBig** class and calling the Hello method.
11. Continue with the **Main** method, use **Console.WriteLine** to display the friendly name of the current AppDomain. Add the same code to the delegate method.
12. Add a call to unload the second AppDomain.
13. Run the application and watch the domain name changes.
14. Open a tool such as Process Explorer. Set a breakpoint in the delegate method before creating the TestBig instance.
15. Run with the debugger. Look at **Process Explorer** view of loaded DLLs. Note that SomeBigLib is not there. Or is it? Can you explain it? You can also use the “Modules” window (Debug->Windows->Modules) if running under the Visual Studio debugger.
16. Step over the creation of the instance. Look at Process Explorer again. Any changes?
17. Let the application run until the unloading of the second AppDomain. Look at Process Explorer again.
18. Add breakpoints to the constructor and finalizer and make sure the behavior of the application is understood.

Lab 6.2: AppDomains and Static Fields

1. Add a public static field to the **TestBig** class, and initialize it in a static constructor to the calling AppDomains’s friendly name.
2. Add a breakpoint to the static constructor.
3. Display the value of the static field in the Main method and in the delegate method.
4. Run the application and make sure you understand the results.

Lab 6.3: Marshalling Across AppDomains

1. Create a new console application named **AppDomainMarshal**.
2. Create 3 classes: A (derives from Object and not serializable), B (which is **Serializable**), C (derives from **MarshalByRefObject**). Add console output for the constructors and finalizers.
3. Create a new AppDomain.
4. For each type A, B, C:

    a. Create an instance in the new AppDomain using **AppDomain.CreateInstanceAndUnwrap**.
    b. Check if the returned reference is a proxy by calling **RemotingServices.IsTransparentProxy**.
5. Unload the AppDomain.
6. Explain the results.

Module 7: Multithreading

Lab 7.1: Basic Multithreading

1. Create a new Windows Forms project (or WPF) and name it **PrimesCalculator**.
2. Add two text boxes, a "Calculate" button and a list box to the form.
3. When the user presses the "Calculate" button, spawn a new thread and calculate the prime numbers within the range indicated by the textboxes.
4. When done, populate the list box with the numbers. Note that you need to access the list box within the UI thread (hint: you can use `Control.Invoke` for WinForms or `Dispatcher.Invoke` for WPF)

Lab 7.2: Cancelling a Long Running Task

1. Continue from the previous exercise: Add a "Cancel" button to the form.
2. When the user presses "Cancel", the calculating thread should abort the calculation. Use an Event object to signal the thread to "bail out" early. Hint: you can call **WaitHandle.WaitOne** with a timeout of zero to just "check" if the object is signalled.

Lab 7.3: 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** (or a **SemaphoreSlim**) 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 7.4: 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 7.5: Using the BackgroundWorker Component

1. Create a new Windows Forms (or WPF) project called **BackgroundPrimeCalc**.
2. From the toolbox, add two text boxes, two buttons (“Calculate” and “Cancel”) and a listbox.
3. When the user clicks the calculate button, the program should calculate prime numbers in the range given by the two textboxes (similar to the previous exercise). This one, however, will use the **BackgroundWorker** component for the asynchronous operation. Add a BackgroundWorker component from the toolbox to the form.
4. From the properties window, add an event handler for the DoWork event. You can do that with code written without the designer.
5. Continue by following the instructions detailed in the course to use the **BackgroundWorker** for asynchronous operations.
6. Add a progress bar to the form and use the BackgroundWorker’s capability to notify periodically that progress has been made. Update the progress bar as appropriate.

Module 8: Tasks

Lab 8.1: Using the Parallel Class

1. Create a new Console application named **Primes**.
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 8.2: Cancellation 1

1. Continue from the previous exercise.

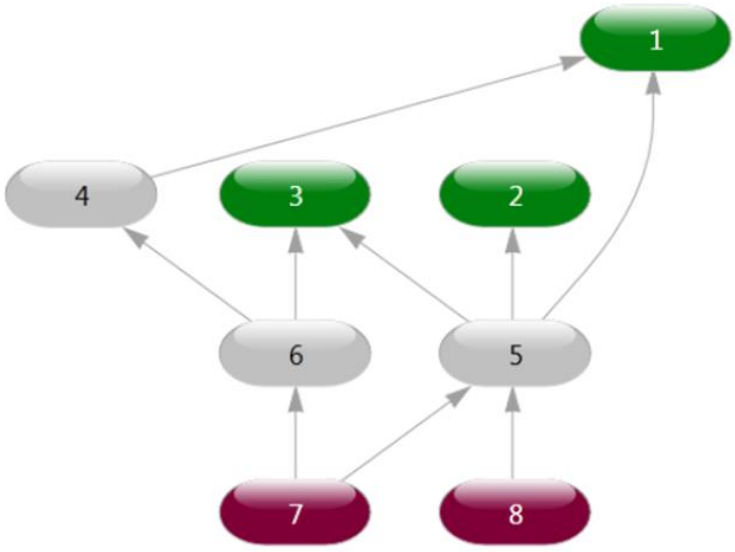
2. Change the signature of **CalcPrimes** to accept two integers and nothing else. Remove the **ParallelOptions** object and use the default degree of parallelism.
3. We want to cancel the operation randomly. Add a **Random** instance in the **CalcPrimes** method before the parallel loop starts.
4. The lambda expression should accept a **ParallelLoopState** variable as well as the loop index. Look at the other overloads of **Parallel.For**.
5. In the lambda expression, call **Random.Next** with 10000000 as argument, and check for the return of zero. If it is, call the **Stop** method of ParallelLoopState.
6. Call **CalcPrimes** from Main with the last number as 30000000 and display how many numbers returned before being cancelled.

Lab 8.3: Cancellation 2

1. Copy the solution of Lab 7.2 to a new folder.
2. Change the cancellation mechanism to use **CancellationTokenSource** and **CancellationToken**.

Lab 8.4: 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 4 depends on project 1, project 6 depends on projects 3 and 4, etc.

3. Write code that "builds" these projects (sleep for one second for each build step) sequentially.
4. Write code that builds concurrently using tasks, **Task.ContinueWith** and **Task.Factory.ContinueWhenAll**.

Module 9: Async Await

Lab 9.1: Async with C# 5.0

1. Create a new Windows Forms project (or WPF) and name it **PrimesCalculator**.
2. Add two text boxes, a "Calculate" button and a Label to the form.
3. When the user presses the "Calculate" button, call an **async** method called **CountPrimesAsync** that counts the number of prime numbers between a first and last number.
4. Call the method and await the result.
5. Finally, show it on the Label.
6. Add cancellation support using a **CancellationToken** and **CancellationTokenSource**.

Lab 9.2: Custom Awaiter

1. Create an awaiter that will allow awaiting on a integer. The integer represents the amount of miliseconds to delay.
2. Create a custom awaiter that will allow awaiting on a process and continue when the process exit

(*)Lab 9.3: More C# 5.0 Async

1. Create a new WinForms or WPF application named **ImageManip**.
2. Add appropriate UI to allow loading some image file.
3. The application should do some processing on the image (e.g. turning it to grayscale, inverting the colours, etc.) all in an asynchronous way with C# 5.0. You can start with a synchronous version, and modify it (this part should be easy) to become asynchronous (non-blocking) with the help of C# 5.0.

Appendix A: Serialization

Lab A.1: Automatic Serialization

1. Open the solution **AutoSerialize** from the **Start** folder.
2. Mark the classes **Company**, **Employee** and **Department** as serializable.
3. Create a new method called **SaveCompany** that accepts a **Company** object and a **Stream** object.
4. Serialize the Company instance to the stream with a **BinaryFormatter**.
5. Create a new method called **SaveToFile** that accepts a path name and a **Company** object. In the method, create a **FileStream** and call **SaveCompany** as required.

6. Create a new method called **LoadCompany** accepting a **Stream** and returning a **Company** instance. Implement as required.
7. Create a new method called **LoadFromFile** that accepts a file path and returns a **Company** instance. Implement as by calling **LoadCompany**.
8. Create a new method called **SaveToMemory** that accepts a Company instance. The method should create a **MemoryStream** object and use **SaveCompany** to save the data to that object.
9. In the Main method:
    a. Call **CreateCompany** and use the result to save the company to a file on disk using **SaveToFile**.
    b. Display the departments and employees of the company by calling **Display**.
    c. Set the Company reference to null and call GC.Collect. Make sure you see a finalizer called.
    d. Call **LoadFromFile**, passing the same path used to save the data. Display the results. Make sure the data is the same that was saved.
    e. Call **SaveToMemory** and save the resulting stream in a local variable.
    f. Set the Company reference to null and call **GC.Collect**.
    g. Load the company back from the **MemoryStream**. Display the results.

Lab A.2: Custom Serialization

1. Continue with the previous lab, or open the solution in the **Start** folder.
2. The Company class will support custom serialization. Add support for the ISerializable interface to the Company class.
3. Implement **GetObjectData** as follows:
    a. Save the name, departments and employees using variants of **SerializationInfo**.**AddValue**.
    b. Save the number of departments that have managers.
    c. Save the current save time.
4. Add the required constructor for deserialization.
5. Implement as appropriate to get the data back using **SerializationInfo**.**GetInt**, **GetString** and **GetValue**. Display to the console the number of departments with managers and the save time.
6. Run and make sure all the data is saved and loaded correctly.