Name: Shant Ahuja Date: 4/18/20

Lab 1 – C# Potpourri

Objectives

Part 1 – Getting Started with Unit Testing

Part 2 – Iterators and Enumerations

Part 3 – Extension Methods

Part 4 – LINQ and Lambda Expressions

Part 5 – Assemblies and Reflection the Right Way

Part 6 – Assemblies and Reflection the Fun Way

Background/Scenario

Throughout the course, we will be working with data and collections quite frequently. This lab reviews some of the different ways you can manipulate data stored in collections and external assemblies using some of the features that makes C# fun to use.

This lab is the only lab which does not require the creation of a front end user interface. All code written will be run and verified by the creation of unit tests. There are many different unit test frameworks out there. For this lab, we will be using Microsoft Tests already installed and integrated with Visual Studio.

Required Resources

- Visual Studio 2017 (any edition)
- ILSpy with Reflexil Addin
  - See Appendix A for download and installation

Part 1: Getting Started with Unit Testing

Unit testing is an integral part of the development process. When sitting down and preparing to write code, you will almost always have an idea of what you want the code to do. While you can write the unit tests at any point in the development process, writing unit tests before writing any code can be pretty useful especially if you refactor your code quite often. This is called Test-driven Development. We'll explore some of those benefits as the lab goes on.

Step 1: Creating the test project

Create the solution for this lab if you have not done so already. Because we're not going to be creating a user interface, a console application will work just fine.

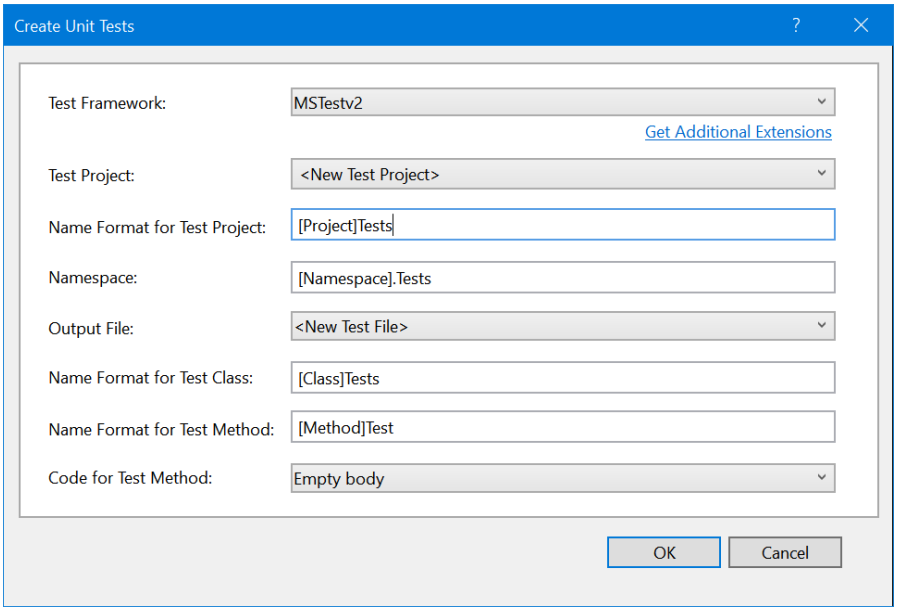
Note: Be sure you selected a .NET Framework project and not a .NET Core project.

In the console project you created, create a new class named Lab1 and mark it public. This is where you will be writing all the code for the lab that we will be testing. Tests are generally written only for public API classes and methods.

Next, write a Hello function that simply returns the string "Hello From Lab1!".

Now, right click anywhere inside the method body and select Create Unit Tests.

In the unit test creation dialog window that opens, ensure that the Test Framework is “MSTestv2”. Change the Code for Test Method option to Empty body. Your window should look similar to this:



You can leave all the other fields set to their default values. This will create a new unit test project in your solution with “Tests” appended to the name and the project icon should have a beaker symbol.

Step 2: Running the Unit Test

Inside the test method body that you just created, create a string variable named expected and set it to “Hello From Lab1!”.

Next, create a new instance of the Lab1 class that you created in the previous step, call the Hello method, and then assign the return to a string variable.

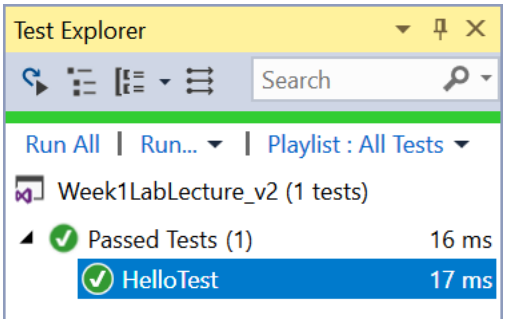
Print to the console the return value so we can see what it outputs.

Finally, call the `Assert.AreEqual(string expected, string actual)` method and pass in the appropriate values.

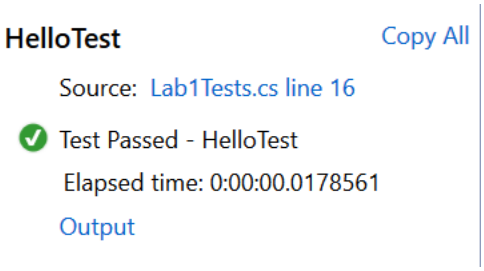
What is the Assert class and why is it important to unit testing?

Assert class is made up of many different verification methods to quality test your code. it is important to unit testing because we use it to easily access methods that can test our code.

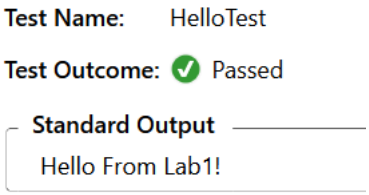
Right click anywhere inside the method body and select Run Tests. The test explorer window should open up and your test should run. If everything worked, you should see a green checkmark indicating the test has passed.



At the bottom of the Text Explorer window, you should see an Output link. Click on it.



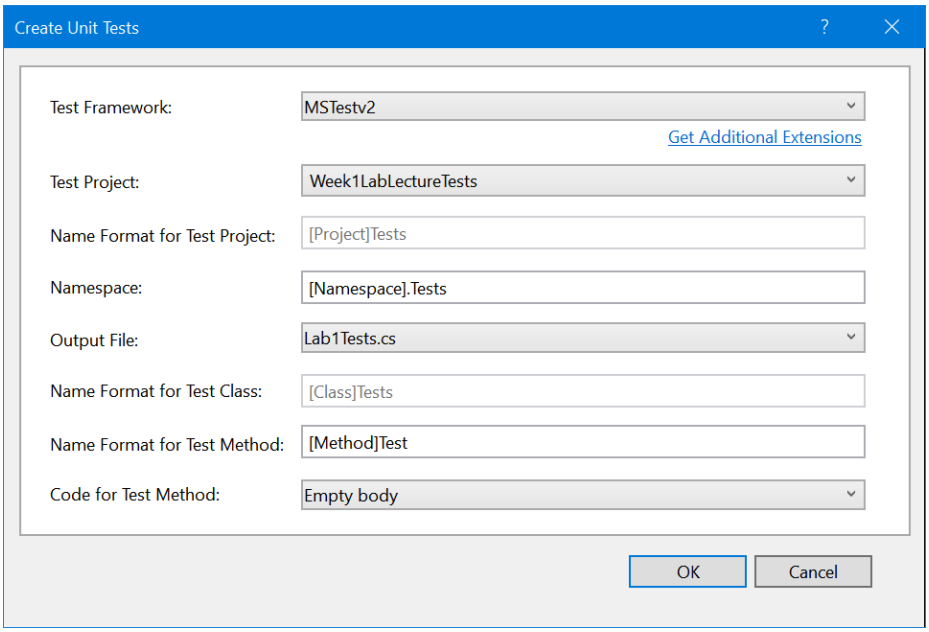
This will bring up the test output window and will show you everything that has been printed by running the tests.



Step 3: Creating more unit tests

There are a couple of ways to create new tests. You could either manually add additional methods in your created test class and adorn them with the `[TestMethod()]` attribute or you could generate them automatically.

To generate them automatically, simply right-click inside the method body of your Lab1 class after typing a new function you want to write a test for and click Create Unit Tests again. This time, the create dialog should have a new option where you can select your test project and test cs file. Select your existing test project and file and click create.



The function should appear right alongside your previous test method.

Part 2: Iterators and IEnumerable

Step 1: Fibonacci Numbers

Write an IEnumerable function that returns a series of ints that represent the Fibonacci sequence. The function should take one int parameter which identifies how many numbers to return in the series. Start the series at 0.

Make sure to use yield. Do not precompute a collection and return it.

What does the <T> mean after IEnumerable?

Type

Step 2: Unit Test

Write a unit test for the Fibonacci function.

Declare an array and initialize it manually with the first 10 Fibonacci numbers and name it expected.

What are the first 10 Fibonacci numbers?

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Declare a variable named result and initialize it by calling the Fibonacci function. Set a breakpoint on this line.

Next, use a foreach to iterate through the enumeration and print out the values to the console.

Finally, call the `CollectionAssert.AreEqual()` method and pass in the expected and result into the parameters. Now, you can't directly compare an int[] to an IEnumerable, so convert the result to an array by calling the `ToArray()` extension method when passing it as a parameter.

Why didn't we use the Assert class but instead used CollectionAssert? What would have happened if we used `Assert.AreEqual()` instead?

~~Assert only tests one object. We need to test a collection of objects, so we use~~ CollectionAssert. If we use Assert, it will fail because it can't look at the whole collection. it only sees the object (an array of 32 bit ints in this case).

Run the test (do not debug just yet). Verify that the output is correct and that the test passes.

Now debug the test by right-clicking inside the method body and select Debug Tests.

Once the breakpoint hits, step into the function call (or press F11). You should see that we actually stepped over the function call.

Why didn't the debugger step into the Fibonacci function?

the iEnumerable simply returns the instructions when the time comes, whereas a List (queue, stack, etc) will walk the collection and produce results immediately and step by step

Continue step-debugging through the foreach and observe when we enter and leave the Fibonacci function.

At what point do we leave the Fibonacci function and when to do we reenter? When does the Fibonacci function finally end?

we leave the fibonacci function when we send the "result" out to the number that we are using for the foreach. we enter the fibonacci function on the "in" keyword. the fibonacci function ends when we are within the scope of the foreach function

Part 3: Extension Methods

Step 1: Extending IEnumerable

Write a generic extension method for the IEnumerable<T> type. Name it EveryOther. Have it skip all the elements of even or odd indices. Use a Boolean parameter to specify: true to skip even, false to skip odd indices.

Example list:

	Original
0	Red
1	Green
2	Blue
3	Cyan
4	Magenta
5	Yellow
6	Black

	Skip Even
0	Green
1	Cyan
2	Yellow

	Skip Odd
0	Red
1	Blue
2	Magenta
3	Black

Similar to before, make sure to use yield. Do not precompute a collection and return it.

Also, be sure to keep the generic type for the extension method.

Hint: Remember, extension methods must be marked static and be added to static classes!

Why do you have to use the this keyword and what does it represent when creating extension methods?

we have to use this keyword because it cannot be modified and we must be able to use it everywhere in the code solution. it represents a method that cannot be modified and can be used everywhere.

Step 2: Unit Test EveryOther

Write two separate unit tests to test your EveryOther extension method. The first on a list of strings and the second on an array of ints.

Use the example list above to test a string collection and use the Fibonacci function to test a collection of ints.

Within each unit test, test both the even and odd parameters.

Part 4: LINQ

Step 1: Using the Where method

The Where extension method was introduced as part of the System.Linq namespace. The purpose of the Where method is to filter items in the collection that match a predicate.

What is a predicate delegate?

predicate delegate involves a function that returns true or false. the delegate is the function pointer, so we point to a function that returns true or false.

Create a new test method called WhereTest.

Using the Fibonacci function you wrote in Part 1, generate a list of the first 10 Fibonacci numbers.

Using the Where method, filter the list of Fibonacci numbers and keep only even numbers.

Do it twice using both the method syntax and query syntax.

For the method syntax version, use a lambda expression for the predicate.

Step 2: (Extra Credit) Using the GroupBy method

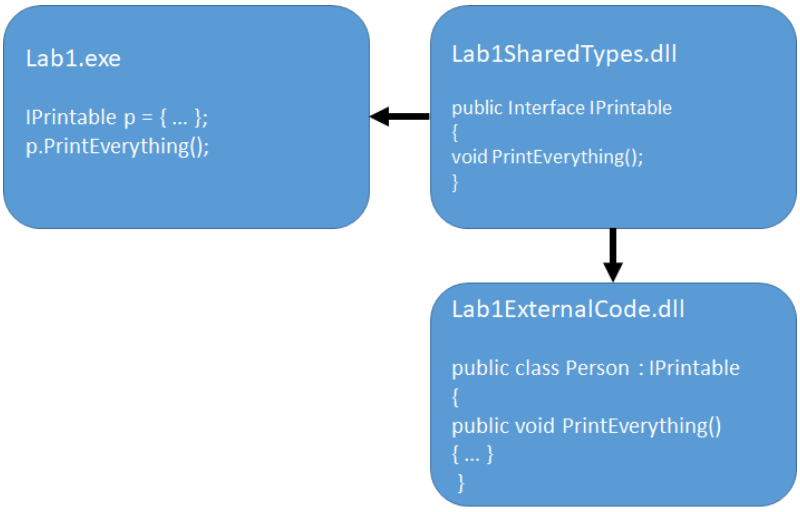
Create a new test method called GroupByTest.

Extend what you just did in the previous step but instead of only filtering just the even numbers, use the GroupBy method to return both the even and odd numbers in two separate groups.

Do it using both the GroupBy extension method using lambda expressions and query syntax.

Part 5: Assemblies and Reflection the Right Way

When working with assemblies, it's always best to interact with them with some sort of shared type or interface so you always have a type to work with. Not only is reflection slower, but most of the Invoke methods require casting to and from the object type which could cause a tremendous amount of overhead depending on the types used. Let's explore one of the ways to keep a shared type by way of an interface.



From looking at the diagram, the main project and the external DLL projects both reference the shared types DLL. The main project has no idea of any types in the external project but because we implement a shared interface, the main project would have no problem using the code.

Step 1: Create the library projects

Create two library projects. One library will hold the shared interface and the other will have the code implementation.

Create the interface in the shared library and create the class in the external code file.

For the PrintEverything() function, print to the console, “Hello, John Smith”. Well, it is a person class after all.

Step 2: Reference the libraries

Add a reference to the shared library in both the main and the external code projects. Make sure you do not add a reference to the external code library from the main project.

Step 3: Use reflection to call the PrintEverything() method

Use the `Assembly.LoadFrom()` method to get a handle to the external code assembly and pass in the filename of the external DLL that gets compiled.

Using the handle to the assembly, call the `CreateInstance()` method and use the typeof() method to pass in the type of the IPrintable interface. The `CreateInstance()` method returns an object type so cast it afterwards.

Call the PrintEverything() method and verify that the console outputs the message.

Note: You will need to manually copy the built external code dll to the same directory as the main executable. To make things easier, you could also configure a post build command to automatically copy it.

Part 6: Assemblies and Reflection the Fun Way

Step 1: Download the Lab1SecretCode.dll from Canvas.

Attached to the Lab assignment on Blackboard is the Lab1SecretCode.dll library. This library contains a few methods which return various messages.

Note: Depending on your computer’s security policy, you may need to unblock the DLL. See Appendix B for instructions on how to unblock files downloaded from the internet.

Add the DLL to your project but do not add it as a reference to your project. Open the properties page for the DLL and instruct the compiler to always copy the DLL to the output.

Step 2: Inspect the Assembly

Launch ILSpy and open the assembly.

What are the three public method names in the Lab1Secret class?

GetBrokenMessage()	GetMessage(string)	GetSecretMessage()

Step 3: Call the methods

Use the `Assembly.LoadFrom()` method to load the assembly.

Use the `GetExportedTypes()` method to find the Lab1Secret class type which then can be passed into the `CreateInstance()` method to create an instance and get a handle.

Use the `GetMethods()` method to find the all the methods for the Lab1SecretClass.

There will be more than just three methods returned by this function even though we don't see them in ILSpy. What are they and why are they there?

Boolean Equals(System.Object)
Int32 GetHashCode()
System.Type GetType()
System.String ToString()
they are there because everything in C# inherits from object type. the reason why is because it allows the user to cast an object to a user specified class type. Everything inherits from object, so you can cast every conceivable type to an object and vice versa.

Call the three methods identified in the previous step using the `Invoke()` method.

What are the method's default string responses?

Method Name	Default Response
GetBrokenMessage()	null
GetMessage(string)	"Hello from Lab1!"
GetSecretMessage()	null

Step 4: Go back to ILSpy and view the source of each method

Back in ILSpy, if you view the method source code by clicking the [+] button to expand the hidden sections, you will see that each method has another hidden message.

Step 5: Get the first hidden message

One of the methods takes a string parameter. Use ILSpy to identify what the string the method is looking for, then enter the correct string as the parameter to call the method with. What is the result of the updated method call?

String Parameter	Message Result
string code	"You have found the secret code!"

Step 6: (Extra Credit) Get the next two hidden messages

The next two methods are a little trickier. The next method checks to see if a flag is a certain value before giving you the string. Use reflection to modify the private flag and then call the method.

The last method will never actually give you the message no matter what you do. For that, you will have to use Reflexil to modify the assembly's IL to get it to output the correct message.

What are the messages?

Method Name	Secret Messages
GetSecretMessage()	You have modified the flag!
GetBrokenMessage()	Good job! You have fixed the code and found the message!

Reflection

1. This lab, you had used the ILSpy tool to view the disassembled code from assemblies. Search for a few other alternatives that are out there. Are they free or paid? Compare some of the features. Which tool do you find the best?

dnspy, dotpeek, just decompile. they are free. I like just decompile because the user interface looks good to me

2. Given how easy it was to disassemble the source code for an assembly, there are a few methods of combating that, such as code obfuscators. What is code obfuscation and how does it work? While protecting your code is very important, what are some of the disadvantages of using code obfuscators?

code obfuscation puts a layer of security on top of your code. the disadvantage of using code obfuscator is that it is harder to go back into the code later if you want to modify it.

Part 7: Appendix A – Installing ILSpy with Reflexil Addon

Step 1: Download ILSpy

As of writing this lab, the latest version of ILSpy is version 5.0.2. It can be downloaded from here:

https://github.com/icsharpcode/ILSpy/releases

While there may be newer versions released after this lab has been written, I cannot personally verify if the any later versions have feature changes that do not exactly match the lecture information or compatibility problems with the Reflexil Addon.

Although I suggest to download the version listed to match lecture and lab instructions, I also do want to encourage trying out the newest versions anyway and just play around. Remember, this is supposed to be fun!

Step 2: Download the Reflexil Addon

As of writing this lab, the latest version of Reflexil is version 2.2. It can be downloaded from here:

https://github.com/sailro/Reflexil/releases

I suggest downloading the AIO or ILSpy package since all the assemblies are merged into one simple to copy dll.

Step 3: Extract everything

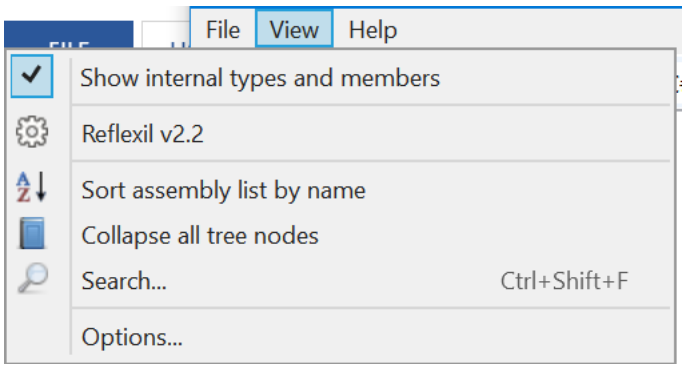
Extract both the ILSpy and Reflexil compressed archives somewhere.

Step 4: Copy the Reflexil dll to the ILSpy directory

Copy the Reflexil.ILSpy.Plugin.dll file into the ILSpy directory so the Reflexil dll is in the same directory as the ILSpy exe.

Step 5: Verify the installation

Launch ILSpy and click on the View menu. You should see a Reflexil menu option along with version number.



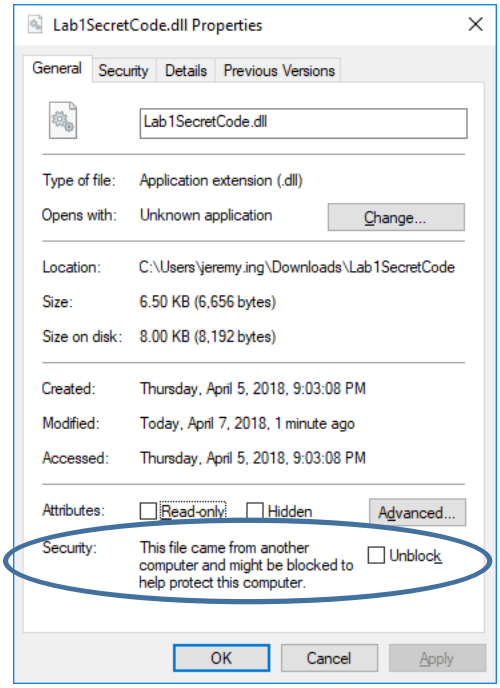
Part 8: Appendix B – Unblocking Files Obtained from the Internet

As a default security policy in Windows, all files downloaded from the internet is sandboxed. Normally this wouldn't be a problem, but when it comes to code libraries and executables, Windows does not allow execution of code that has mixed sandbox types.

For example, any code that you program in Visual Studio will execute with full user privileges. Once you try to load a DLL from the internet, it will be blocked causing Windows to notice you have a mixed permission error.

In most cases, you won't normally encounter this issue since if you downloaded a program from the internet with multiple DLLs, the program will by default execute the program and all the DLLs as the same sandboxed level.

Step 1: Open the properties page of the DLL



Step 2: Click Unblock

Depending on the version of Windows you are running, the Unblock user interface element may be different. In the case of the screenshot above, click the checkbox and then click OK. In over versions of Windows, the Unblock could be a button to click.

That's it!