THE MAERSK MC-KINNEY MOLLER INSTITUTE

Lecture 4
C# Collections & Unit Testing

The slides from this lecture are taken from:
https://learn.microsoft.com/en-us/dotnet/csharp/

Agenda

1. *C# Collections*
   - *List Interface, List and LinkedList*
   - *IComparable and IComparer Interfaces*

   Repetition & Self Study

2. Set and Dictionaries
   - *HashSet,* → **Repetition & Self Study**
   - *SortedSet*
   - *Dictionary* → **Repetition & Self Study**
   - *SortedDictionary*
3. Testing as a Software Development Process & NUnit Framework
   - *NUnit in Rider*
   - *NUnit Attributes*
   - *NUnit Assertions*
   - *Homework Exercise*

Objectives

1. Revise the preliminary concepts of
   - *List and LinkedList*
   - *Comparable and Comparer Interfaces*
   - *HashSet and Dictionary*

2. Build on the Revision to explain the concepts of
   - *SortedSet and SortedDictionary*
   - *Mini Exercises*
3. Introduce the concept of Testing as a Software Development Process & NUnit Framework

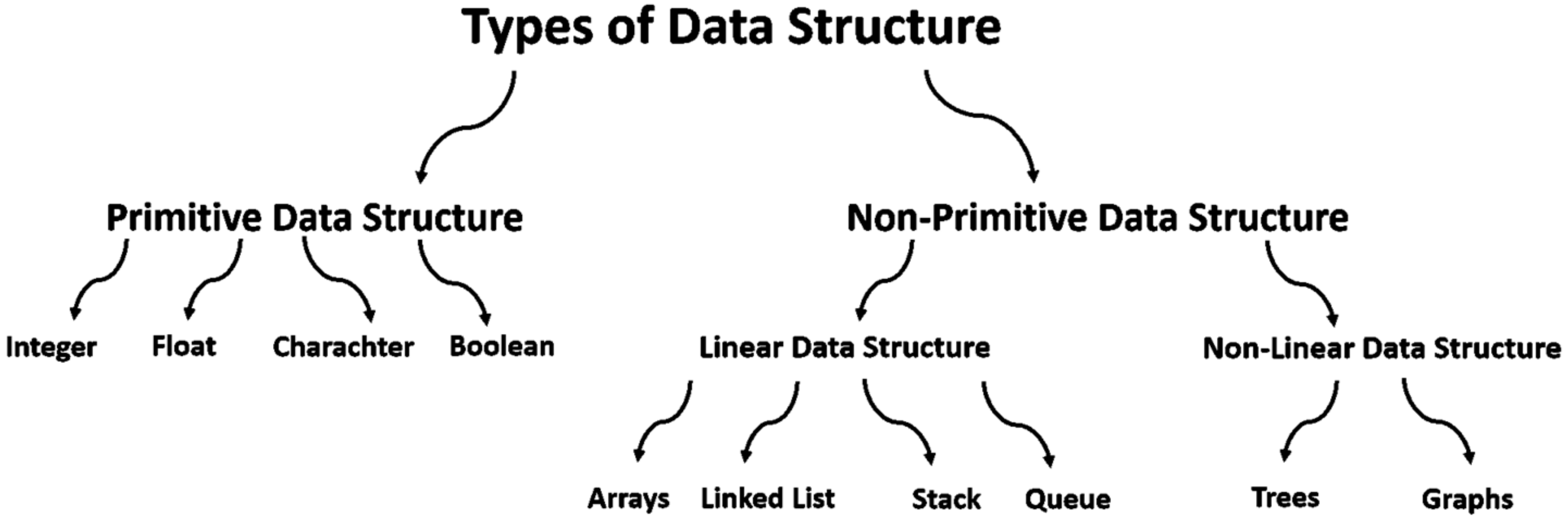
4. Learn how to create tests in Rider

5. Learn how to specify tests using NUnit Attributes

6. Learn how to use Assert class methods to compare actual and expectedresults

Data Structures

- A data structure is a collection of data organized in some fashion and that supports operations for accessing and manipulating the data.



- The data structure in C# is defined in the C# Collections namespace.

Image taken from :https://medium.com/@ivanaemberi/struggling-with-data-structures-d44661f976b2

Collections Namespace Hierarchy

The I**Enumerable** interface is the root interface and it contains the common operations of these collections, such as traversing elements using **foreach** loop**.**

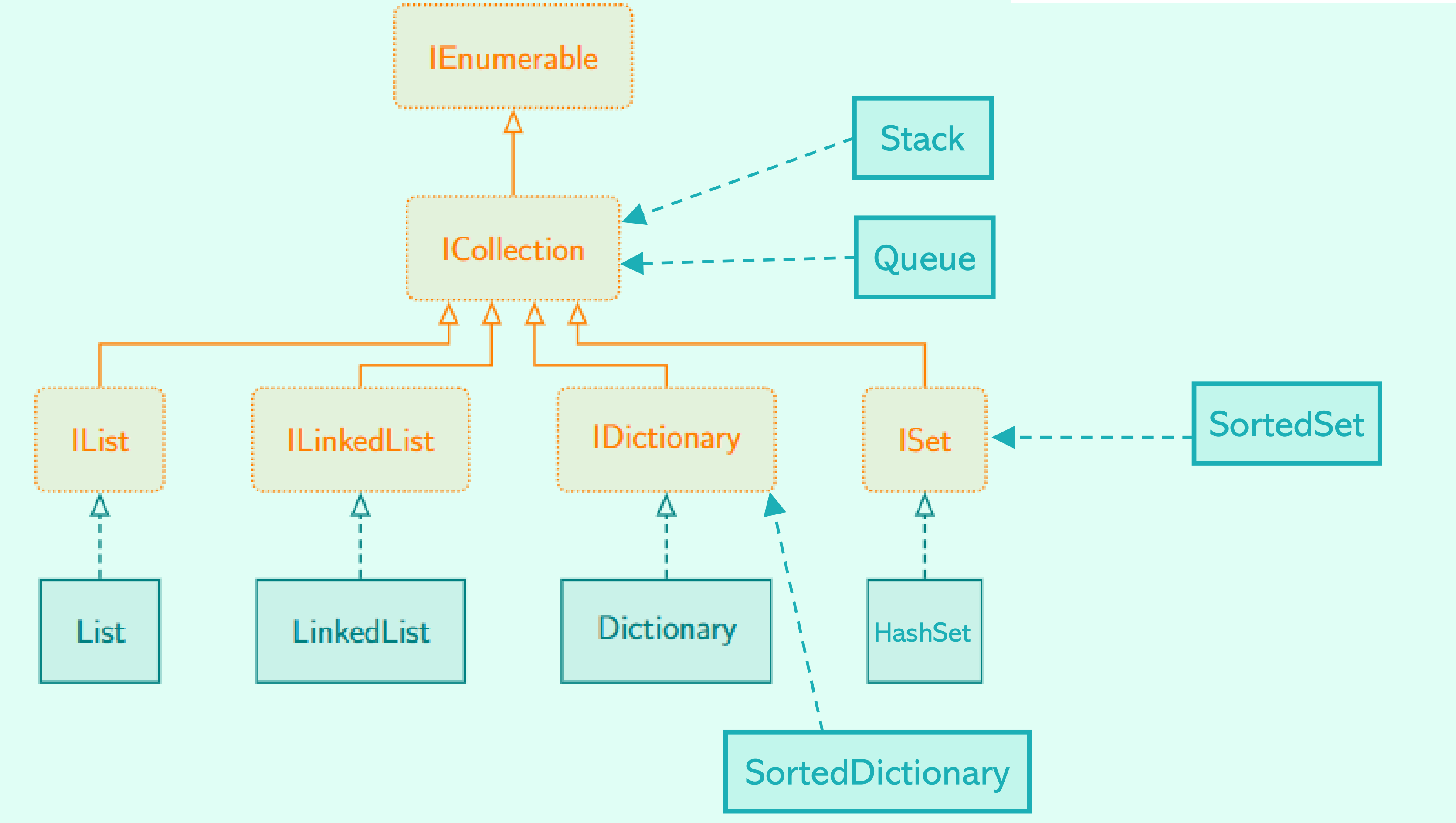
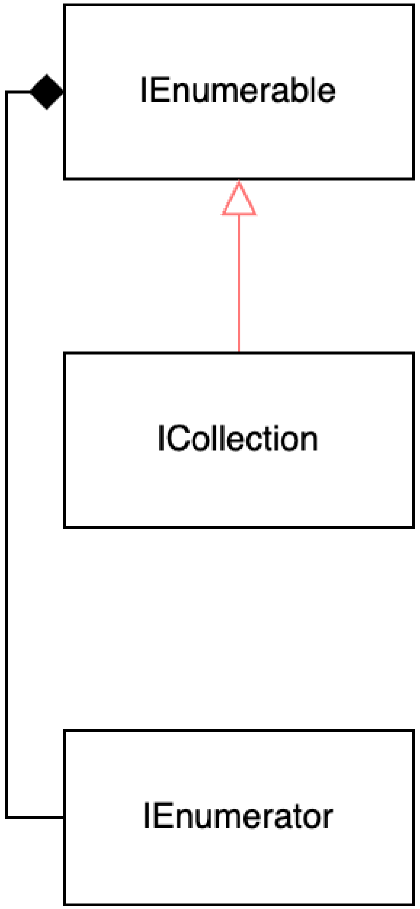


Image: Taken & Modified from Aslak & Peter's slides (OOP course)

IEnumerator Interface

- The IEnumerator interface provides a uniform way to traverse elements of various types.



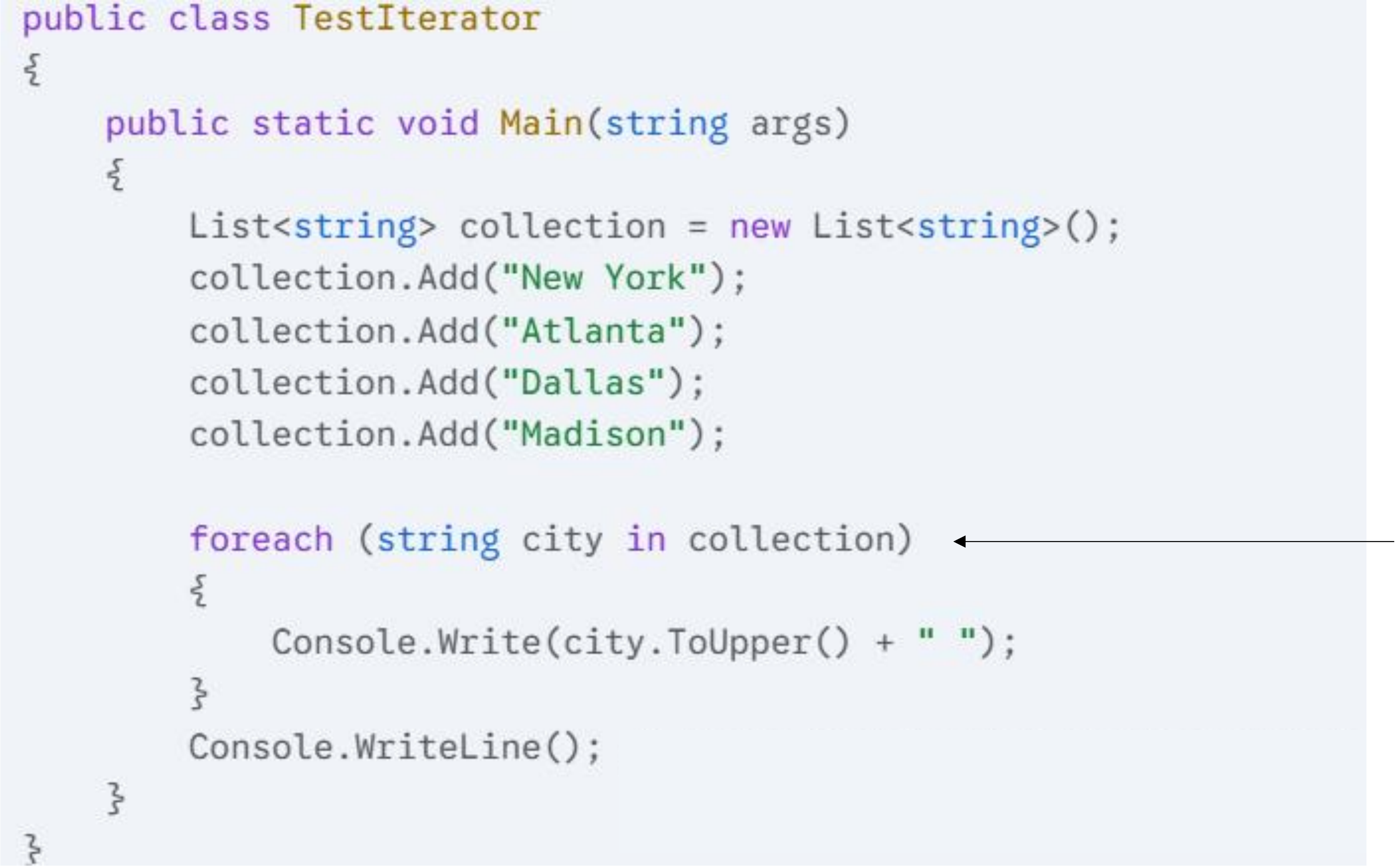
GetEnumerator(): Returns an enumerator that iterates through the collection.

Current: Gets the element in the collection at the current position of the enumerator.

MoveNext: Advances the enumerator to the next element of the collection.

https://medium.com/@lexitrainerph/c-ienumerable-and-ienumerator-an-in-depth-exploration-from-basics-to-advanced-b7217fd15e1c#:~:text=IEnumerable%20%3A%20This%20is%20an%20interface,cursor%20movement%20through%20the%20collection.

List Example with foreach loop

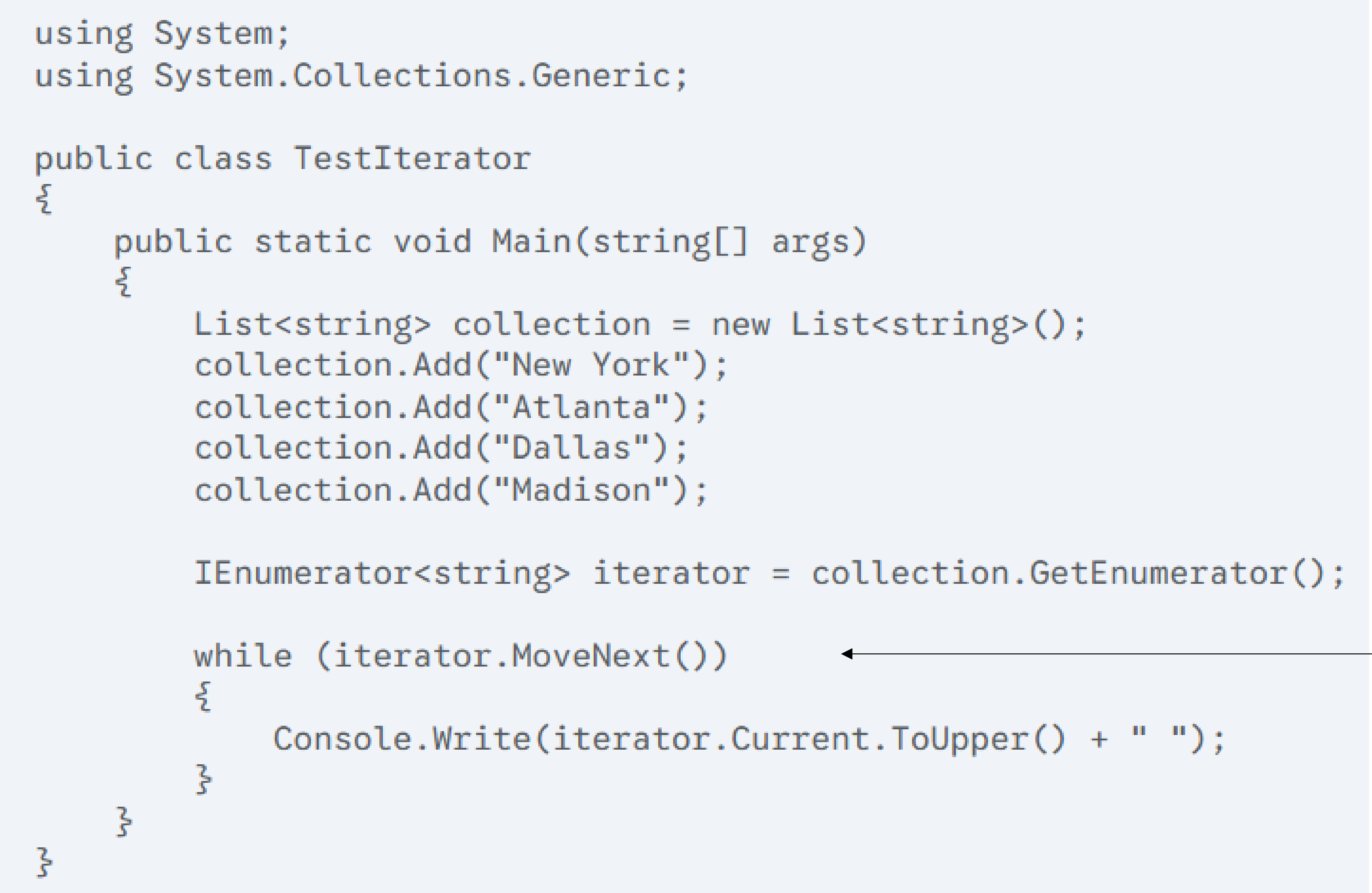


```
public class TestIterator
{
    public static void Main(string args)
    {
        List<string> collection = new List<string>();
        collection.Add("New York");
        collection.Add("Atlanta");
        collection.Add("Dallas");
        collection.Add("Madison");

        foreach (string city in collection)
        {
            Console.Write(city.ToUpper() + " ");
        }
        Console.WriteLine();
    }
}
```

- Simplified Iteration
- Less Control

```
NEW YORK ATLANTA DALLAS MADISON
```

List Example with Enumerator

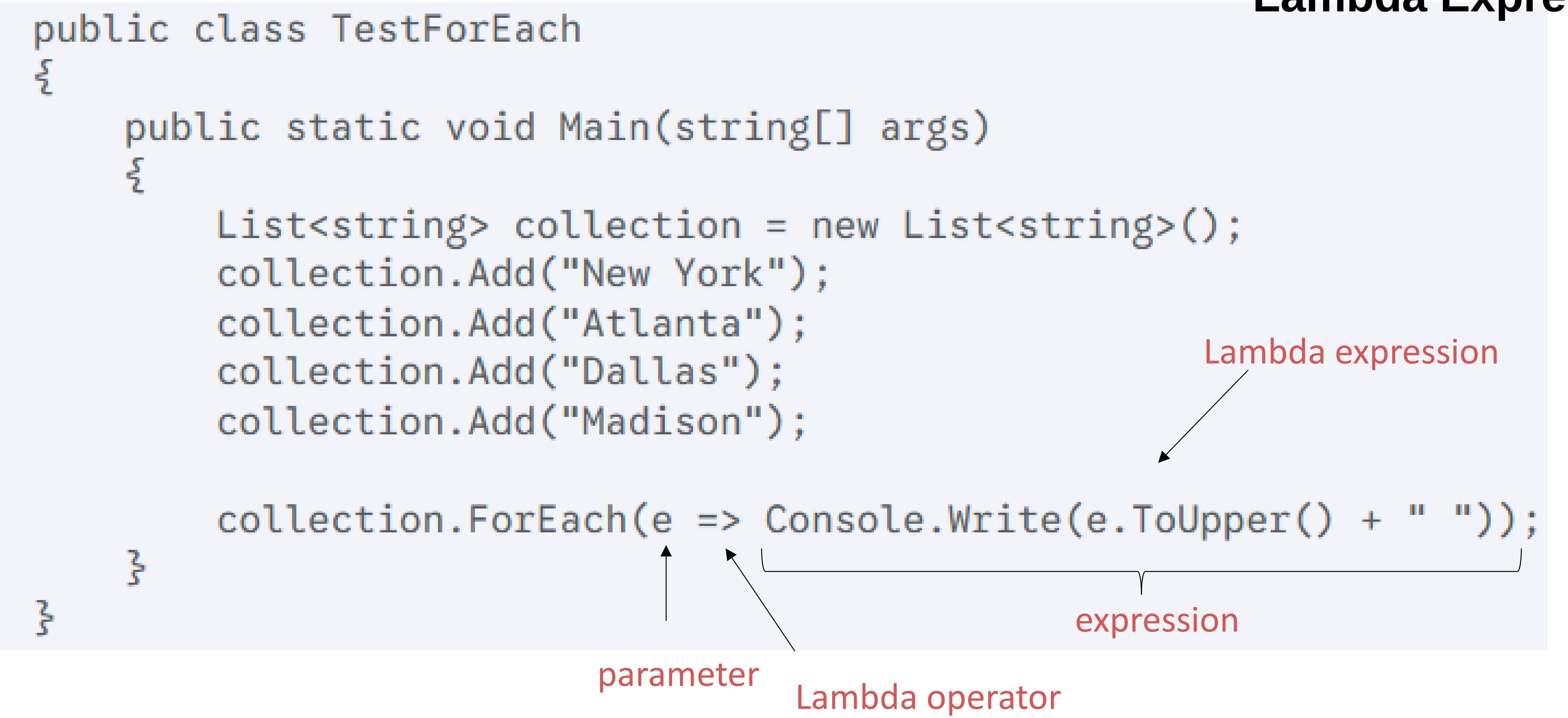


```
using System;
using System.Collections.Generic;

public class TestIterator
{
    public static void Main(string[] args)
    {
        List<string> collection = new List<string>();
        collection.Add("New York");
        collection.Add("Atlanta");
        collection.Add("Dallas");
        collection.Add("Madison");

        IEnumerator<string> iterator = collection.GetEnumerator();

        while (iterator.MoveNext())
        {
            Console.Write(iterator.Current.ToUpper() + " ");
        }
    }
}
```

- Manual Iteration
- Fine-grained Control

```
NEW YORK ATLANTA DALLAS MADISON
```

List Example using Lambda Expression



```
public class TestForEach
{
    public static void Main(string[] args)
    {
        List<string> collection = new List<string>();
        collection.Add("New York");
        collection.Add("Atlanta");
        collection.Add("Dallas");
        collection.Add("Madison");

        collection.ForEach(e => Console.Write(e.ToUpper() + " "));
    }
}
```

Lambda expression

expression

parameter

Lambda operator

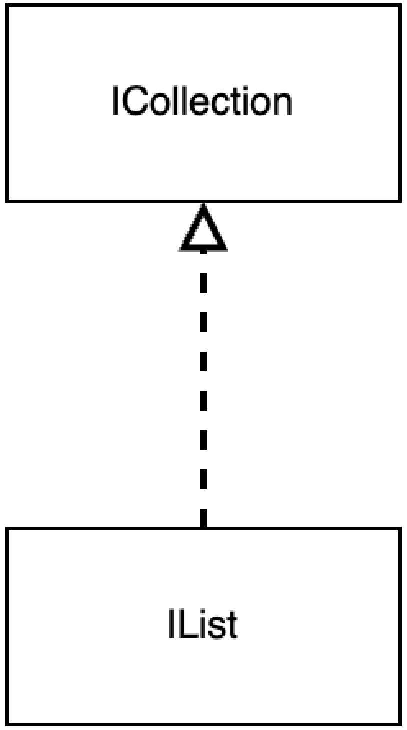
```
NEW YORK ATLANTA DALLAS MADISON
```

IList Interface

The slides from this lecture are taken from:
https://learn.microsoft.com/en-us/dotnet/csharp/

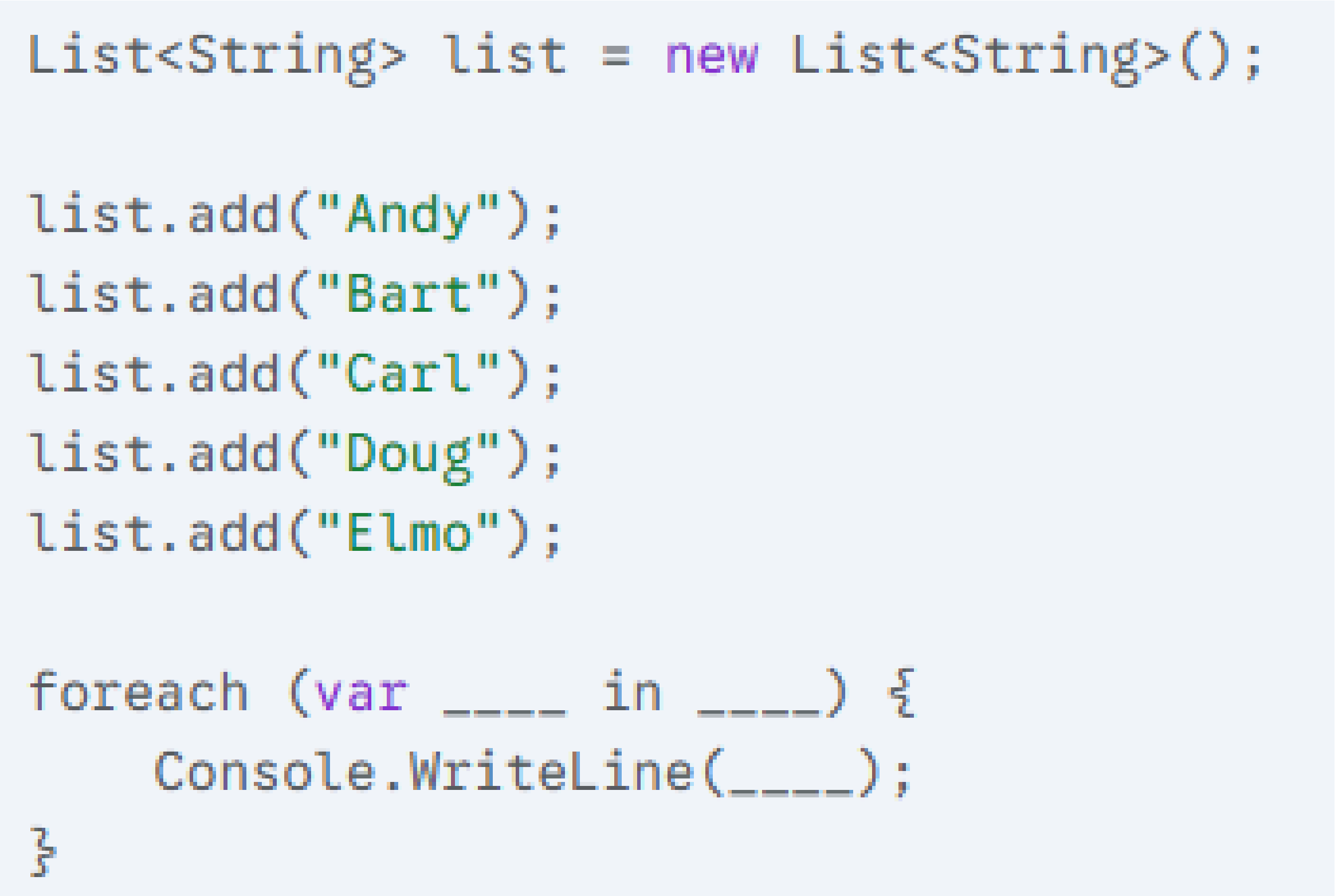
The IList Interface

- The IList interface extends the ICollection interface.
- A list supports random access through an index and unlike an array it can grow or shrink.
- A list stores elements in a sequential order and it allows a user to specify where the element is stored.



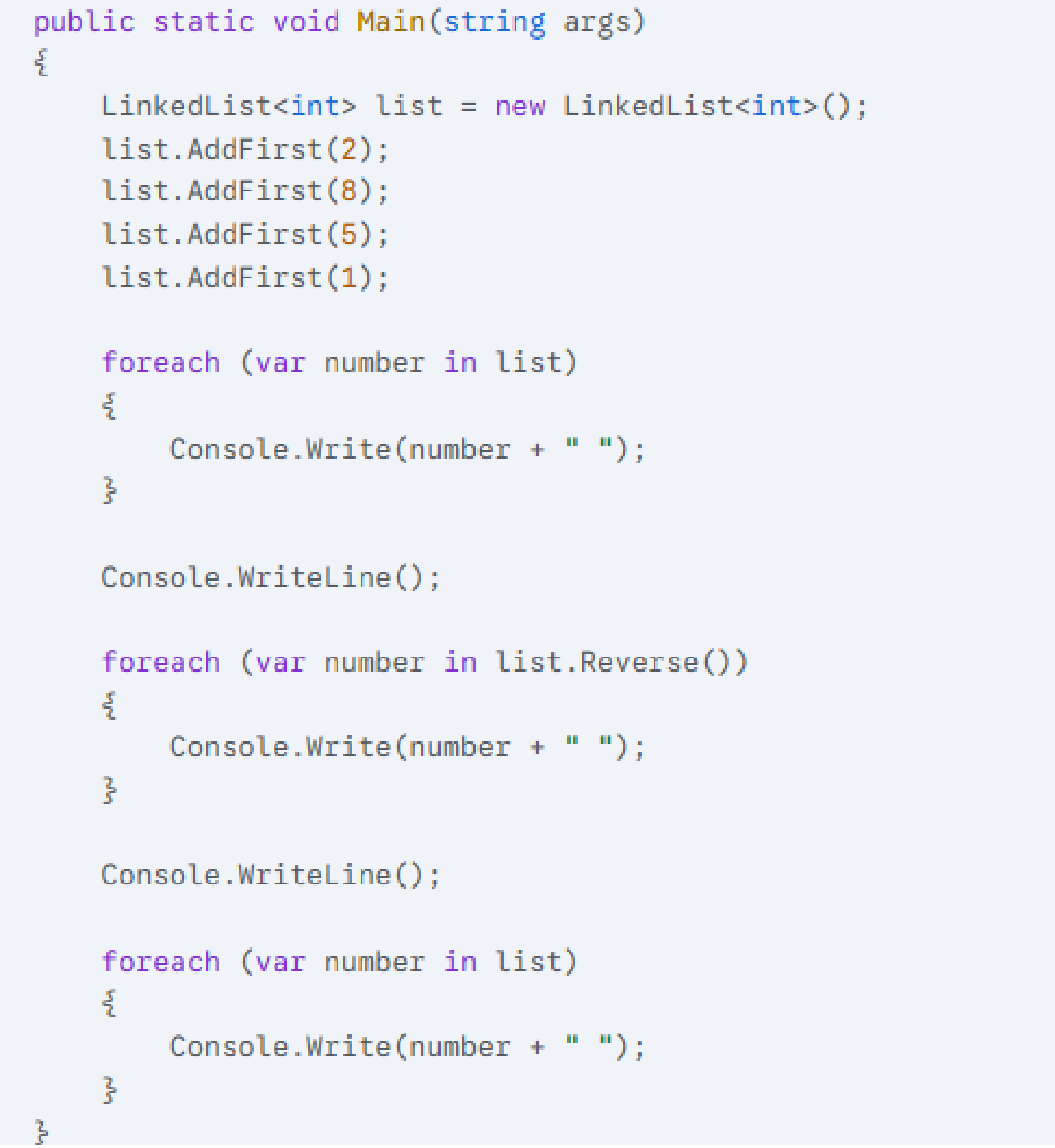
Exercise: List

Fill in the blanks so that the list is printed



```
List<String> list = new List<String>();

list.add("Andy");
list.add("Bart");
list.add("Carl");
list.add("Doug");
list.add("Elmo");

foreach (var ____ in ____) {
    Console.WriteLine(____);
}
```

Exercise: LinkedList- What is the output of 3 loops?



```
public static void Main(string args)
{
    LinkedList<int> list = new LinkedList<int>();
    list.AddFirst(2);
    list.AddFirst(8);
    list.AddFirst(5);
    list.AddFirst(1);

    foreach (var number in list)
    {
        Console.Write(number + " ");
    }

    Console.WriteLine();

    foreach (var number in list.Reverse())
    {
        Console.Write(number + " ");
    }

    Console.WriteLine();

    foreach (var number in list)
    {
        Console.Write(number + " ");
    }
}
```

a) 2 8 5 1

b) 1 5 8 2

c) 1 2 5 8

d) 2 1 8 5

Revision

IComparable & IComparer Interface

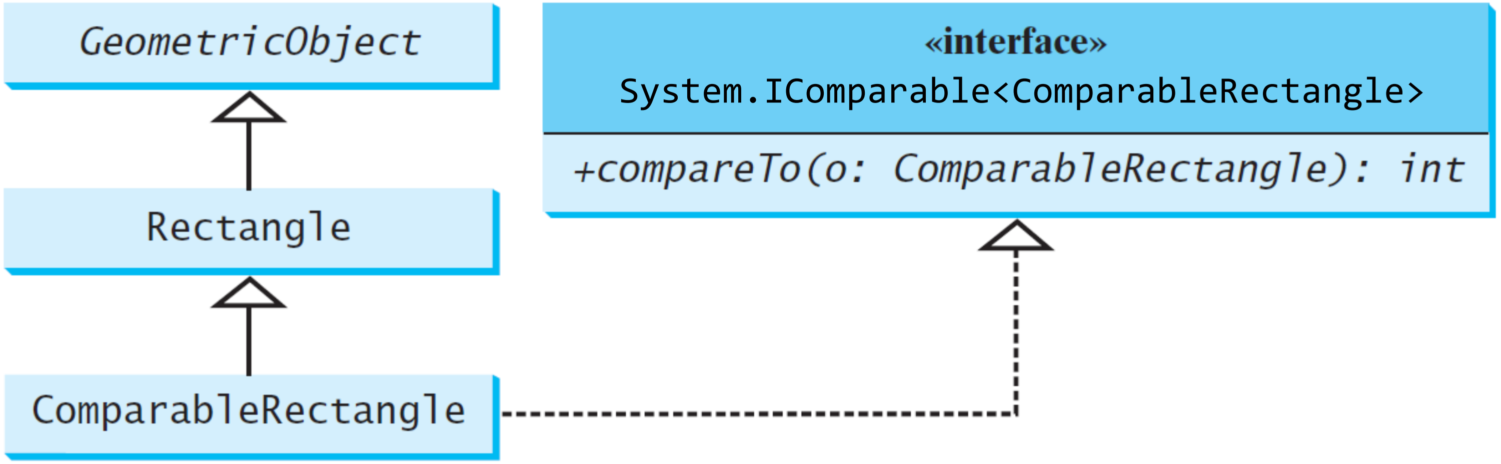
https://dev.to/digionix/icomparable-vs-icomparer-274f

The IComparable Interface

- The I**Comparable** interface defines the **CompareTo** method for comparing objects.
- The **CompareTo** method should return a **-ve** integer, **0** or **+ve** integer,
  if the compared object is less than, equal to, or greater than current object o.
- This objects should be of the same type e.g. two students, two geometric objects

```
public interface IComparable {
  public int CompareTo(E o);
}
```

Example: Using the Comparable Interface



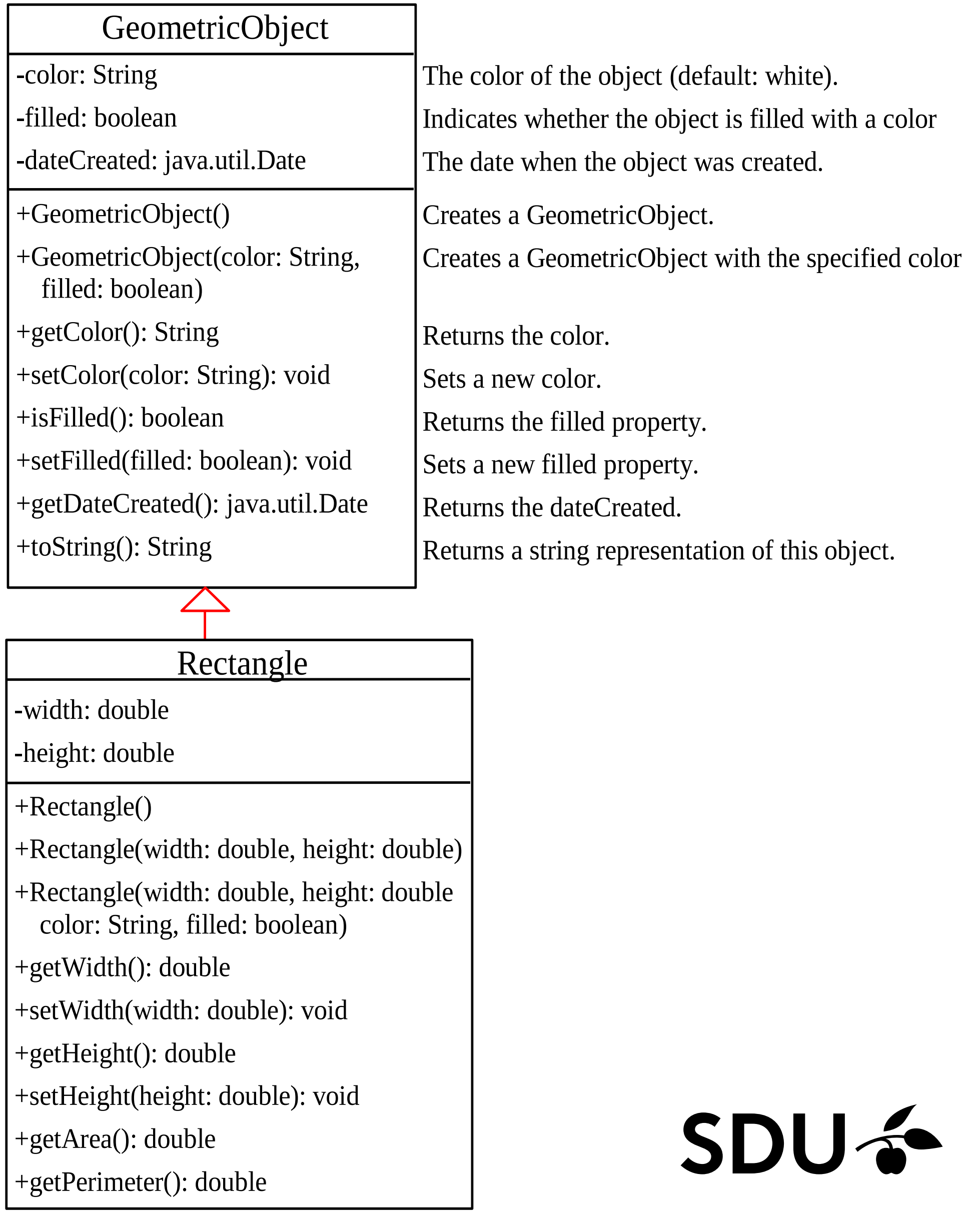
GeometricObject

Rectangle

ComparableRectangle

«interface»
System.IComparable<ComparableRectangle>

+compareTo(o: ComparableRectangle): int



GeometricObject

-color: String — The color of the object (default: white).

-filled: boolean — Indicates whether the object is filled with a color

-dateCreated: java.util.Date — The date when the object was created.

+GeometricObject() — Creates a GeometricObject.

+GeometricObject(color: String, filled: boolean) — Creates a GeometricObject with the specified color

+getColor(): String — Returns the color.

+setColor(color: String): void — Sets a new color.

+isFilled(): boolean — Returns the filled property.

+setFilled(filled: boolean): void — Sets a new filled property.

+getDateCreated(): java.util.Date — Returns the dateCreated.

+toString(): String — Returns a string representation of this object.

Rectangle

-width: double

-height: double

+Rectangle()

+Rectangle(width: double, height: double)

+Rectangle(width: double, height: double color: String, filled: boolean)

+getWidth(): double

+setWidth(width: double): void

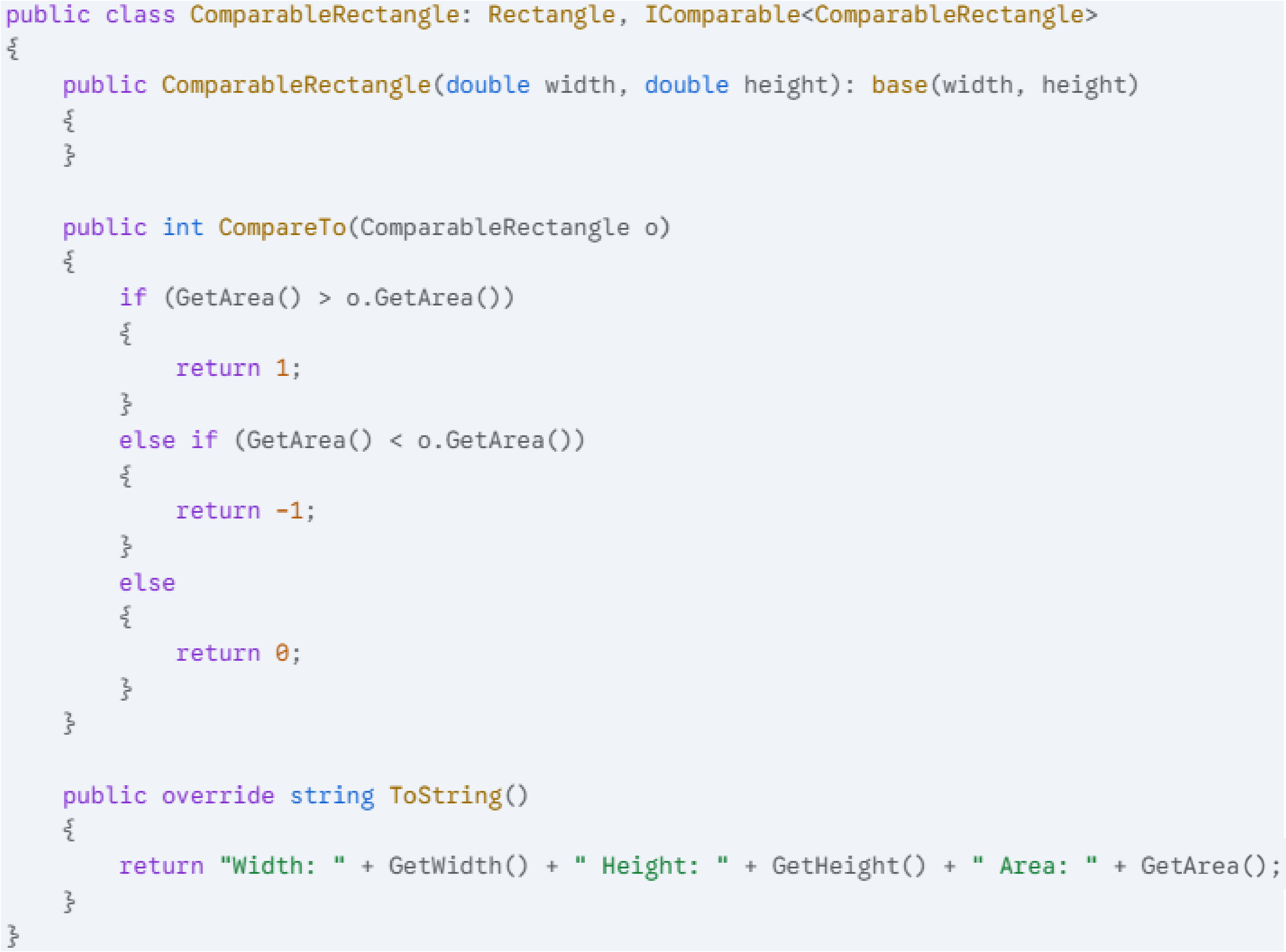
+getHeight(): double

+setHeight(height: double): void

+getArea(): double

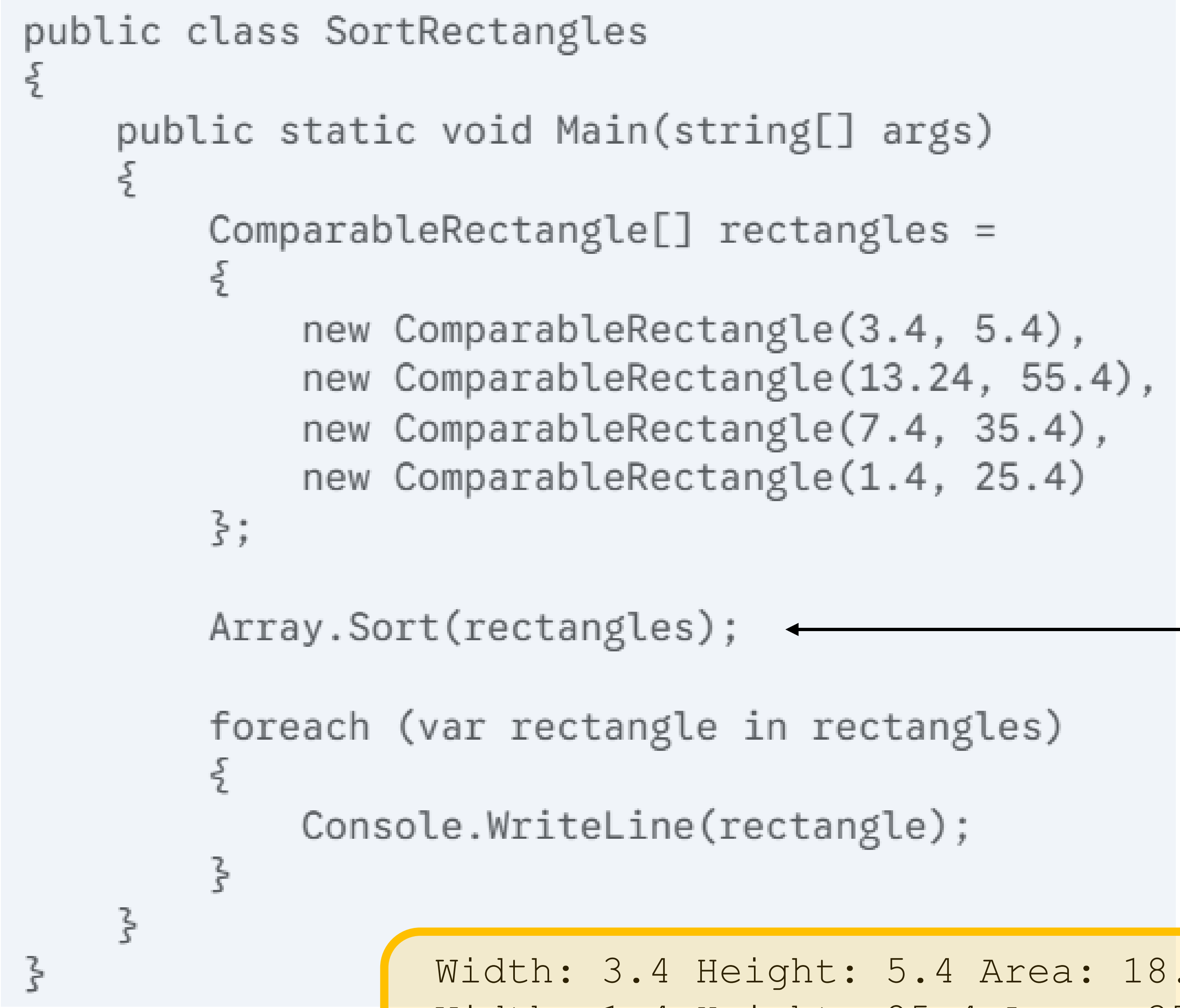
+getPerimeter(): double

Example: Using the IComparable Interface



```
public class ComparableRectangle: Rectangle, IComparable<ComparableRectangle>
{
    public ComparableRectangle(double width, double height): base(width, height)
    {
    }

    public int CompareTo(ComparableRectangle o)
    {
        if (GetArea() > o.GetArea())
        {
            return 1;
        }
        else if (GetArea() < o.GetArea())
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }

    public override string ToString()
    {
        return "Width: " + GetWidth() + " Height: " + GetHeight() + " Area: " + GetArea();
    }
}
```

Example: Using the Comparable Interface



```
public class SortRectangles
{
    public static void Main(string[] args)
    {
        ComparableRectangle[] rectangles =
        {
            new ComparableRectangle(3.4, 5.4),
            new ComparableRectangle(13.24, 55.4),
            new ComparableRectangle(7.4, 35.4),
            new ComparableRectangle(1.4, 25.4)
        };

        Array.Sort(rectangles);

        foreach (var rectangle in rectangles)
        {
            Console.WriteLine(rectangle);
        }
    }
}
```

How could we use the sort method to sort an array of rectangle objects?

```
Width: 3.4 Height: 5.4 Area: 18.36
Width: 1.4 Height: 25.4 Area: 35.559999999999995
Width: 7.4 Height: 35.4 Area: 261.96
Width: 13.24 Height: 55.4 Area: 733.496
```

The Comparer Interface

1. The **Comparer** interface defines a **Compare** method

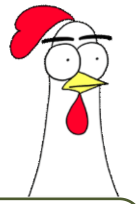
```
public int Compare(T element1, T element2)
```

   - The **Compare** method should return a **-ve** integer, **0** or **+ve** integer, if the element1 is less than, equal to, or greater than element2.

2. The **Comparer** interface can be used to compare
   - objects that doesn’t implement comparable
   - objects that define a new criteria for comparing objects.
   - or objects of different types.

Example: sorting

```
public class SortStringByLength {
 public static void Main(string[] args) {
  string[] cities = {"Atlanta", "Savannah", "New York", "Dallas"};
  Array.Sort(cities, new MyComparer());

  foreach (var city in cities) {
   Console.WriteLine(city + " ");
  }
 }
 public static class MyComparer: IComparer<String> {

   public int Compare(String s1, String s2) {
   return s1.length() - s2.length();
  }
 }
}
```



What is special about the sort method?

What will be the output, if I replace
Array.Sort(cities, new MyComparer());
with
Array.Sort(cities);

```
Dallas Atlanta Savannah New York
```

Example: Lambda Expression

```
public class SortStringByLength {
  public static void Main(string[] args) {
    string[] cities = {"Atlanta", "Savannah", "New York", "Dallas"};
    Array.Sort(cities, (s1,s2) => s1.Length - s2.Length);

    foreach (var city in cities) {
      Console.WriteLine(city + " ");
    }
  }
}
```

```
Dallas Atlanta Savannah New York
```

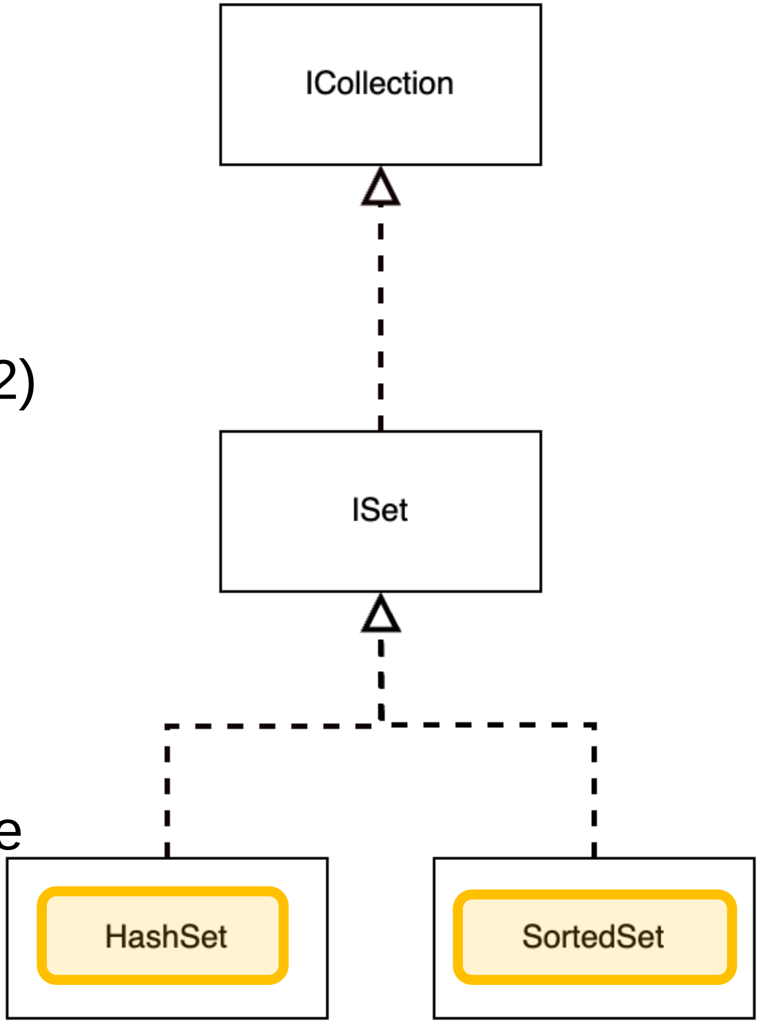
Break (10 min)

Sets and Dictionaries

The slides from this lecture are taken from:
Introduction to Java Programming and Data Structures, Comprehensive Version, Global Edition, 12/E Y. Daniel Liang, Georgia Southern University.

ISet Interface

- The ISet interface extends the ICollection interface.
- Sets contains no duplicate elements.
- So, no two elements e1 and e2 can be in the set and e1.equals(e2) is true.
- HashSet and SortedSet are concrete implementation of ISet.
- HashSets are unordered Sets.
- Hashing provides a super-efficient way to search, insert and delete elements.



Example: Using HashSet

```
public class TestHashSet {
  public static void Main(string[] args) {
    // Create a hash set
    ISet<string> set = new HashSet<string>();

    // Add strings to the set
    set.Add("London");
    set.Add("Paris");
    set.Add("New York");
    set.Add("San Francisco");
    Console.WriteLine(String.Join(”, ", set));

    // Display the elements in the hash set
    foreach (var city in set){
        Console.Write(city.ToUpper() + " ”)
     }
```

```
London, Paris, New York, San Francisco,
LONDON PARIS NEW YORK SAN FRANCISCO
```

SortedSet

- The SortedSet class guarantees that the elements in the set are sorted.
- Elements can also be ordered
  - using the Comparable interface, if the elements implements the comparable interface.
  - By specifying a comparer as an argument to the SortedSet constructor.

```
public class MySetWithCompr {

    public static void Main(String[] a){

SortedSet<string> ss = new SortedSet <string>();

        ss.Add("RED");
        ss.Add("ORANGE");
        ss.Add("BLUE");
        ss.Add("GREEN");
        Console.WriteLine(String.Join(" ", ss));
    }
}
```

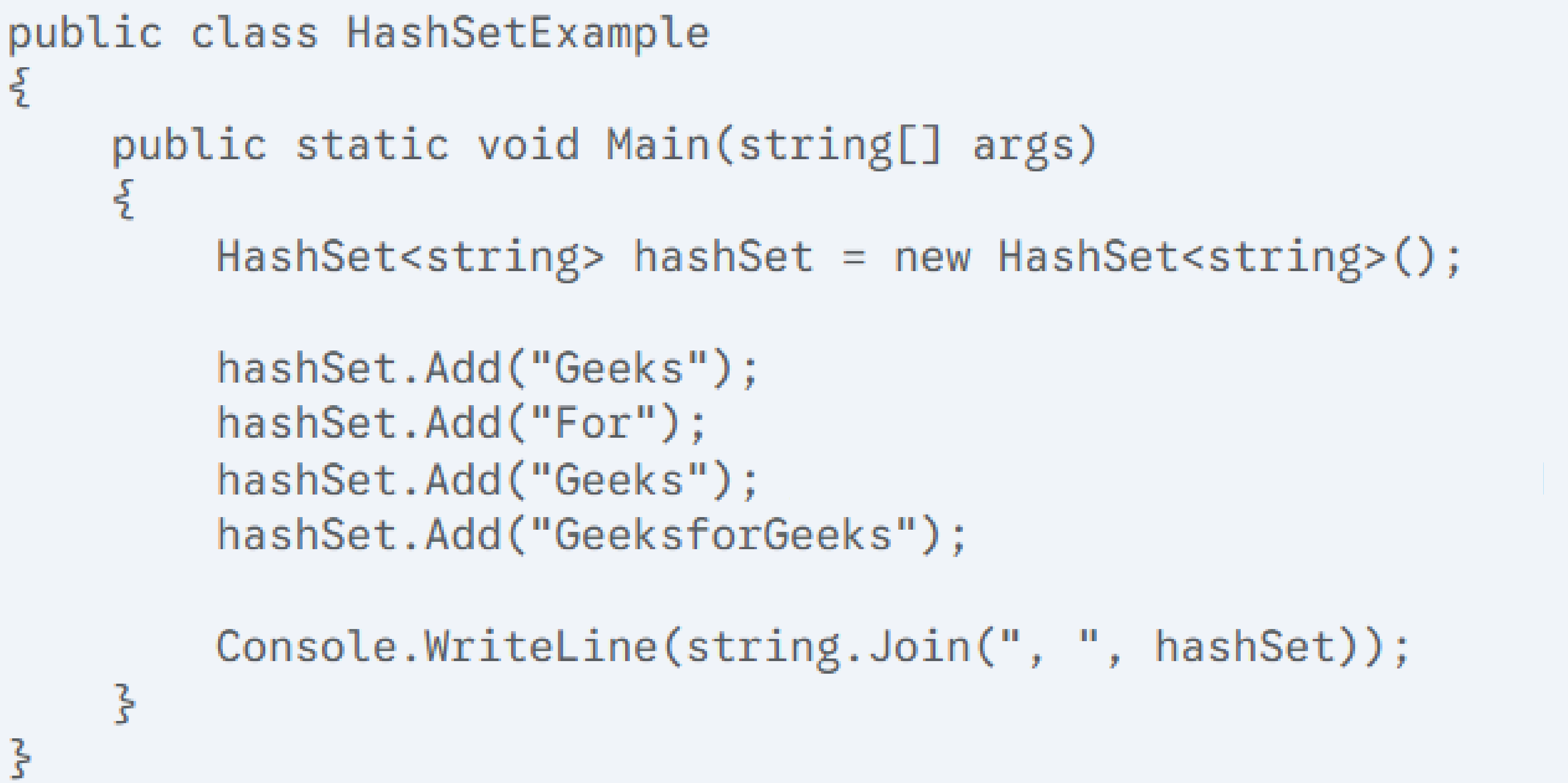
Output???

SortedSet: IComparer as an argument to the SortedSet constructor.

```
public class MySetWithCompr {
    public static void Main(String[] a){

        SortedSet<string> ss = new SortedSet<string>(new MyComp());

        ss.Add("RED");
        ss.Add("ORANGE");
        ss.Add("BLUE");
        ss.Add("GREEN");
        Console.WriteLine(String.Join(" ", ss));
    }
}
class MyComp: IComparer<String>{

    public int Compare(String str1, String str2) {
        return str1.length()-str2.length();
    }
}
```

Output???

Mini Quiz: What is the output????



```
public class HashSetExample
{
    public static void Main(string[] args)
    {
        HashSet<string> hashSet = new HashSet<string>();

        hashSet.Add("Geeks");
        hashSet.Add("For");
        hashSet.Add("Geeks");
        hashSet.Add("GeeksforGeeks");

        Console.WriteLine(string.Join(", ", hashSet));
    }
}
```

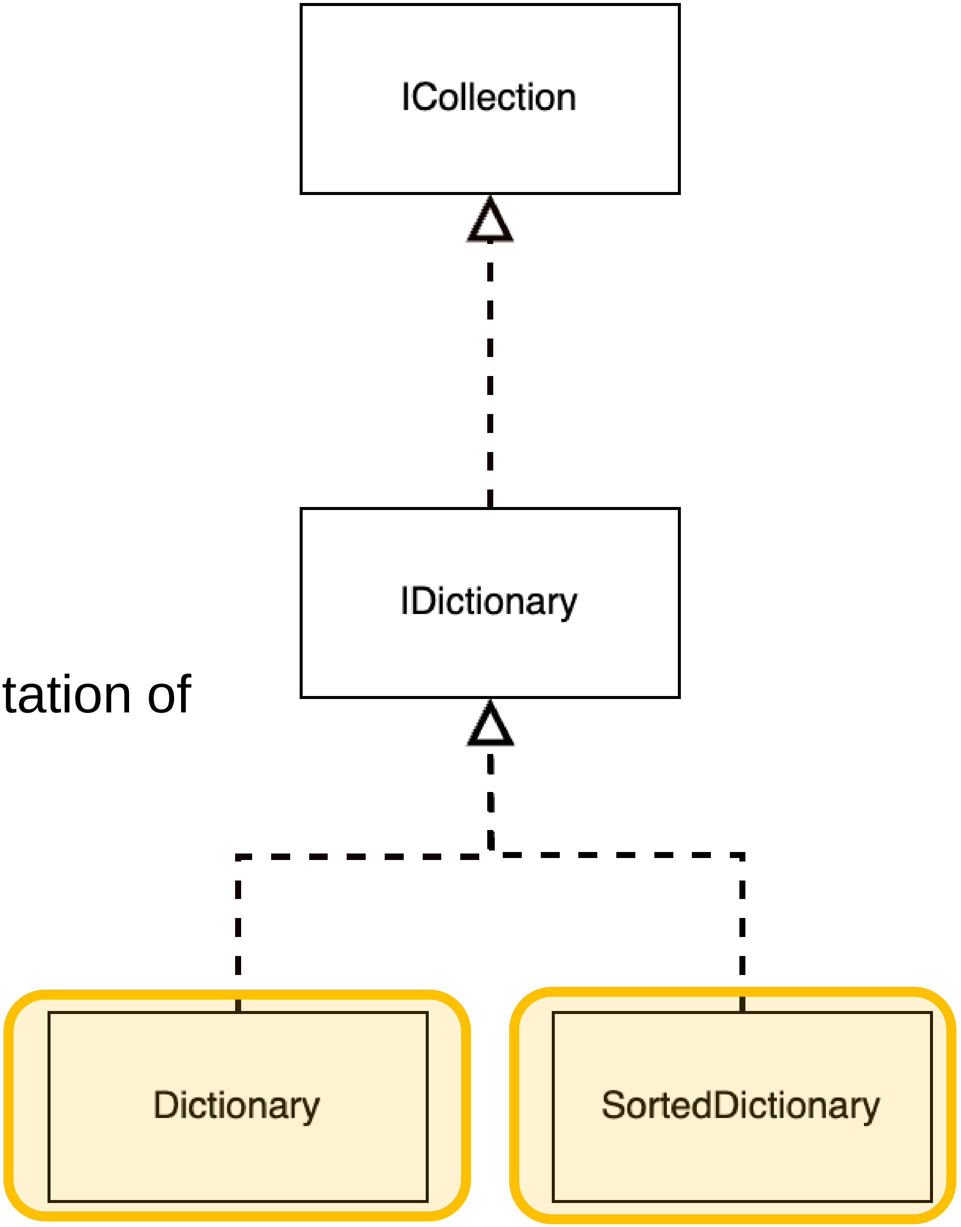
a Or b ???

a) [Geeks, For, Geeks, GeeksforGeeks]

b) [GeeksforGeeks, Geeks, For]

The IDictionary Interface

- Dictionaries are (key, value) pairs.
- The keys are like indexes. In List, the indexes are integer. In Dictionary, the keys can be any objects.
- You can get the object from a dictionary using a key,
- You have to use a key to put the object into the dictionary.
- The Dictionary and SortedDictionary are concrete implementation of the IDictionary interface.



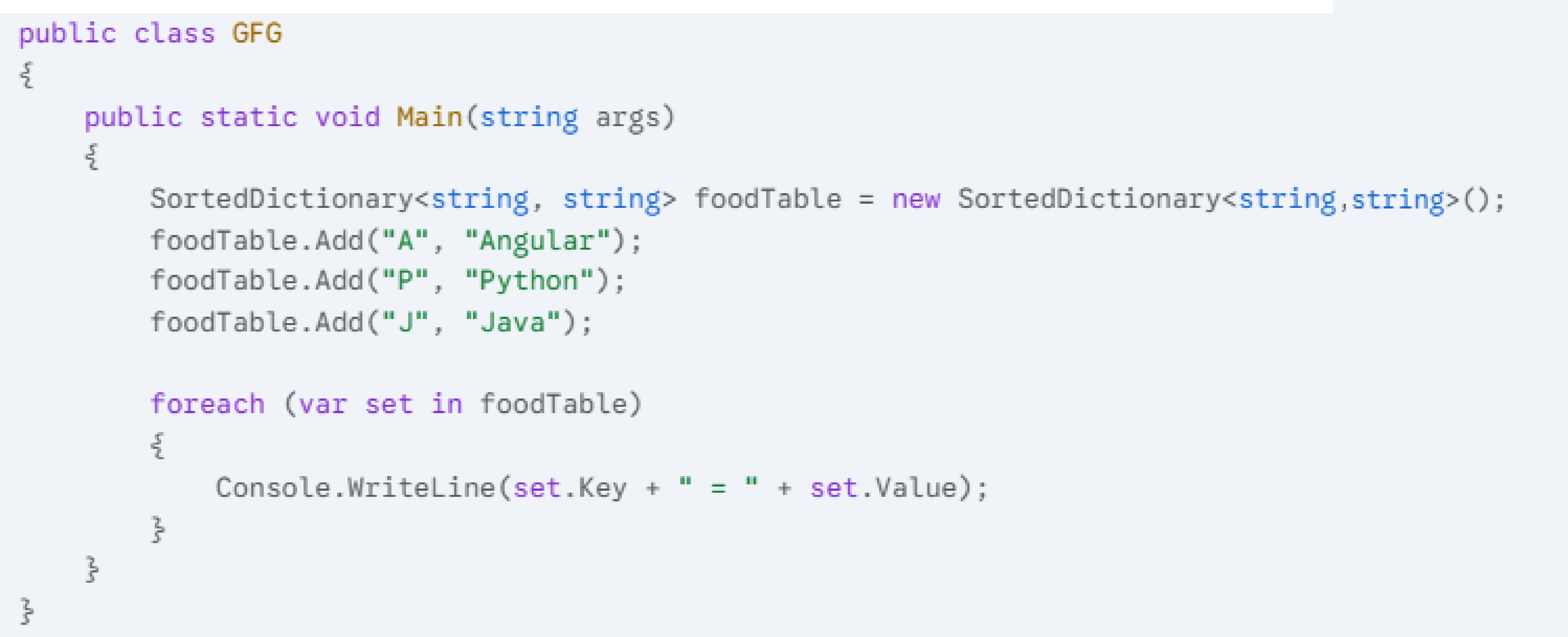
Dictionary and SortedDictionary

- The Dictionary class is efficient for locating a value, inserting an entry, and deleting an entry.
- The SortedDictionary class is efficient for traversing the keys in a sorted order.
- The keys can be sorted using the Comparable or the Comparer interface.
- To use the Comparer interface, we use the constructor

```
SortedDictionary(Comparer comparer)
```

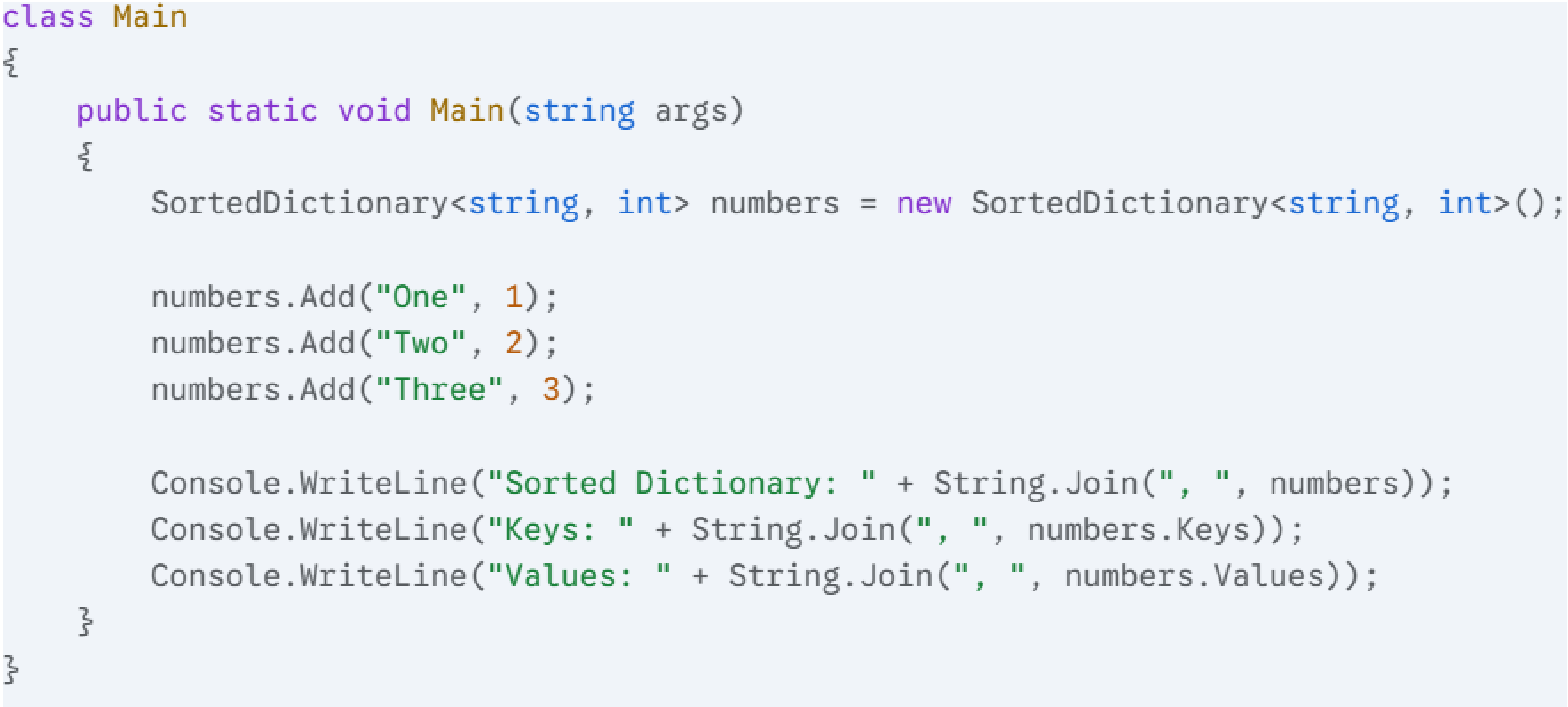
What is the output????

SortedDictionary Traversal



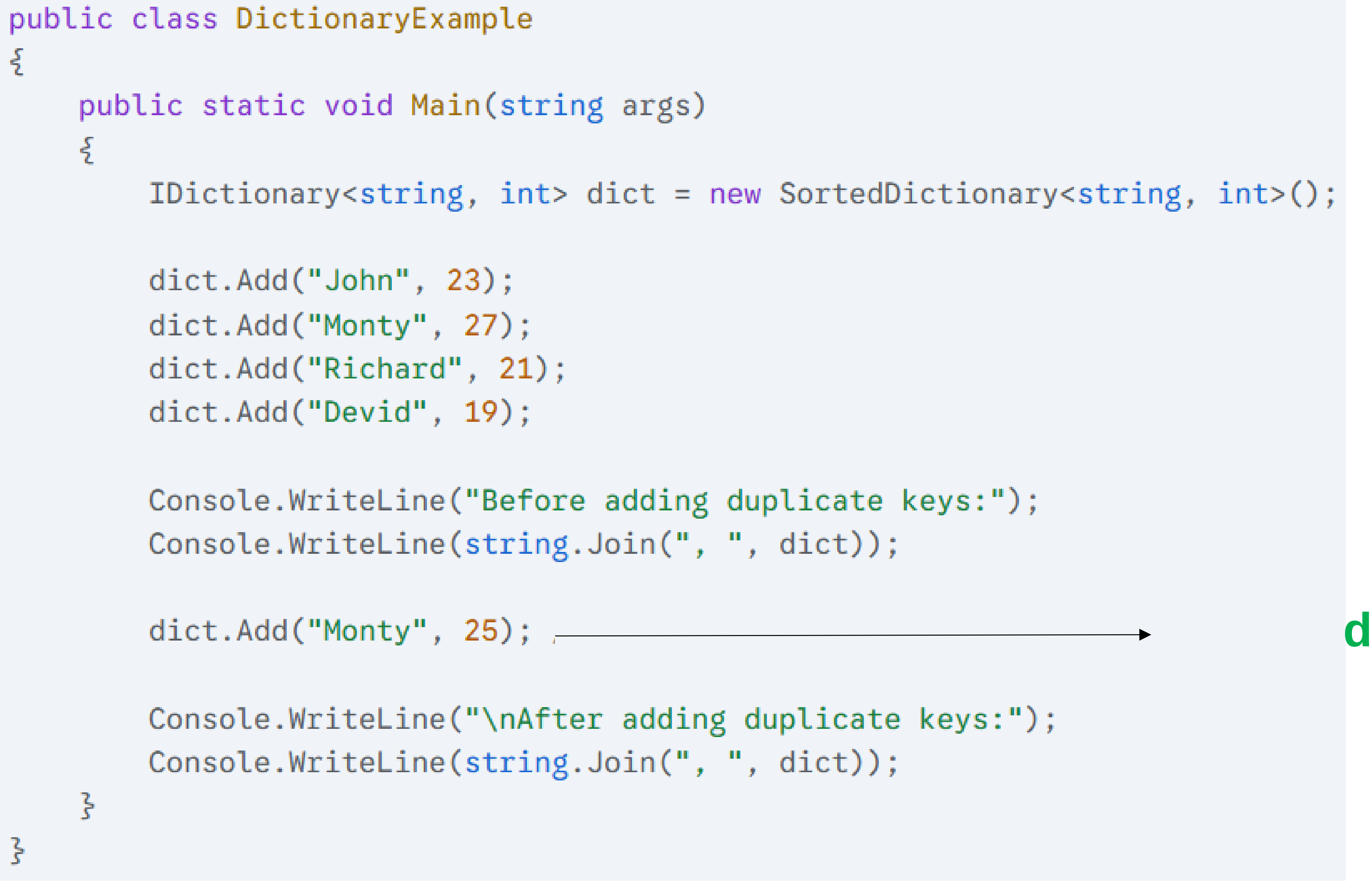
```
public class GFG
{
    public static void Main(string args)
    {
        SortedDictionary<string, string> foodTable = new SortedDictionary<string,string>();
        foodTable.Add("A", "Angular");
        foodTable.Add("P", "Python");
        foodTable.Add("J", "Java");

        foreach (var set in foodTable)
        {
            Console.WriteLine(set.Key + " = " + set.Value);
        }
    }
}
```

Mini Quiz: What is the output????



```
class Main
{
    public static void Main(string args)
    {
        SortedDictionary<string, int> numbers = new SortedDictionary<string, int>();

        numbers.Add("One", 1);
        numbers.Add("Two", 2);
        numbers.Add("Three", 3);

        Console.WriteLine("Sorted Dictionary: " + String.Join(", ", numbers));
        Console.WriteLine("Keys: " + String.Join(", ", numbers.Keys));
        Console.WriteLine("Values: " + String.Join(", ", numbers.Values));
    }
}
```

Mini Quiz: What is the output????



```
public class DictionaryExample
{
    public static void Main(string args)
    {
        IDictionary<string, int> dict = new SortedDictionary<string, int>();

        dict.Add("John", 23);
        dict.Add("Monty", 27);
        dict.Add("Richard", 21);
        dict.Add("Devid", 19);

        Console.WriteLine("Before adding duplicate keys:");
        Console.WriteLine(string.Join(", ", dict));

        dict.Add("Monty", 25);

        Console.WriteLine("\nAfter adding duplicate keys:");
        Console.WriteLine(string.Join(", ", dict));
    }
}
```

What if I replace **dict[“Monty”]=25;**

Take Aways

- List is good at random access
- LinkedList is good at adding or removing elements at the beginning of the list.
- Set are more efficient than lists for storing nonduplicate elements.
- Sets are more efficient than lists for testing whether an element is in a set or a list.
- SortedSet class guarantees that the elements in the set are sorted
- The SortedDictionary class is efficient for traversing the keys in a sorted order.

MCQs Quiz

Go to Plans -> VOP-4 -> VOP-4 (Lecture) -> C# Collection Test

Good Luck ☺

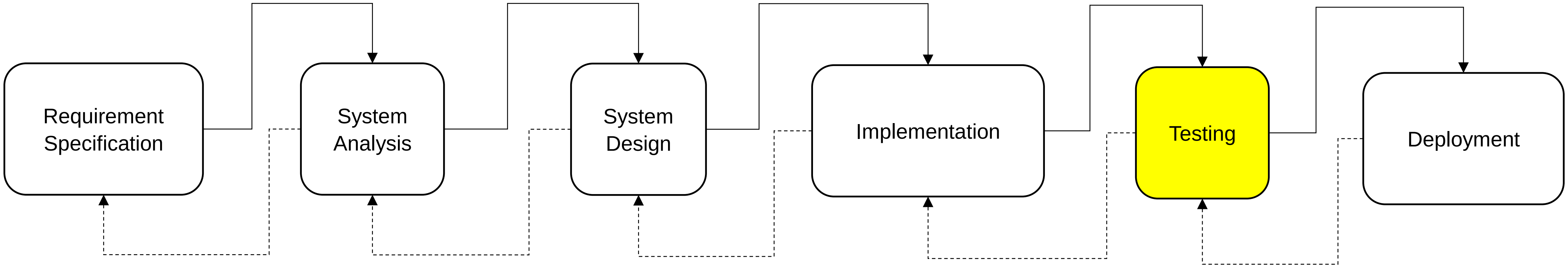
Unit Testing

- The slides provides a general introduction to unit testing concepts and techniques
- Consult additional resources and documentation for comprehensive guidance on unit testing

https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit

Testing and Unit Tests

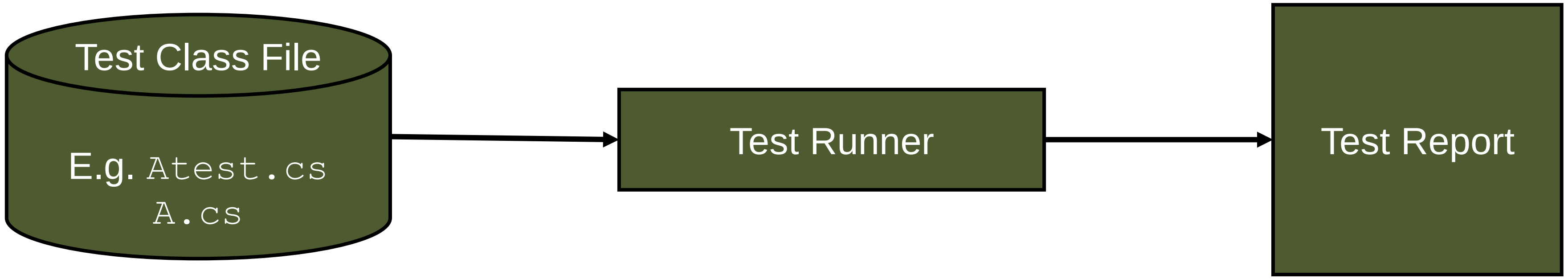
- Testing is part of the software development process.
- Software unit tests helps a developer to verify that the logic of a piece of code is correct.



- A unit test is characterized by a known input and an expected output.
- A unit test is a piece of software that validates that a code results in the expected state

NUnit: Introduction

- NUnit is a unit-testing framework for all .NET languages.
- NUnit is a direct port of JUnit, which was developed by Kent Beck and Erich Gamma. Its first version was released in 1997.
- The NUnit framework comes with the following features which:
  1. provides important classes for writing and running tests.
  2. uses attributes to mark classes and methods as tests
  3. provides a wide range of assertion methods to verify expected outcomes in your tests



Creating Test in Rider

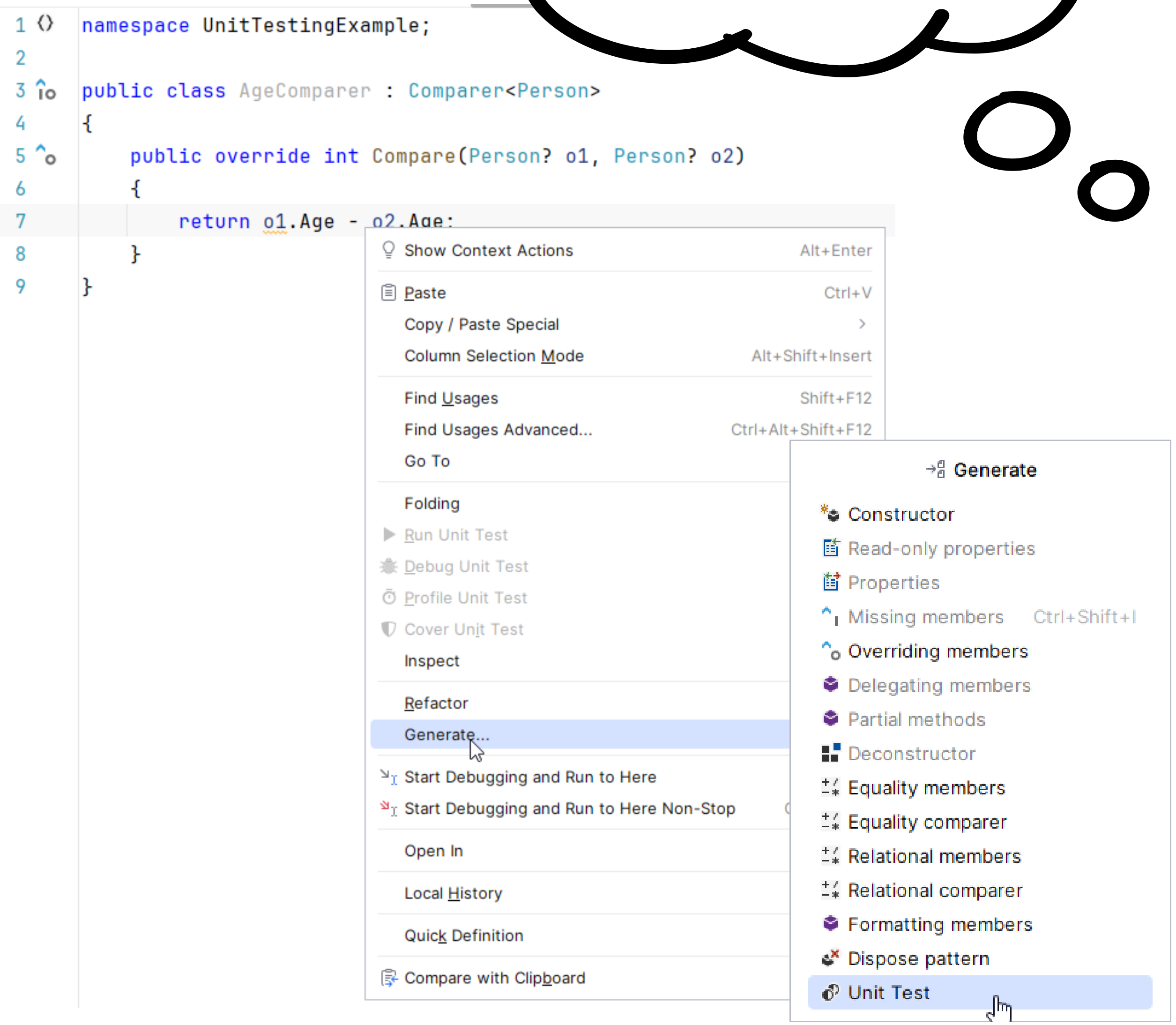
From Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> **UnitTestingApp.zip**

Creating Test in Rider: Method # 1

Assume, **UnitTestingExample** comprises:

- Person.cs
- AgeComparer.cs

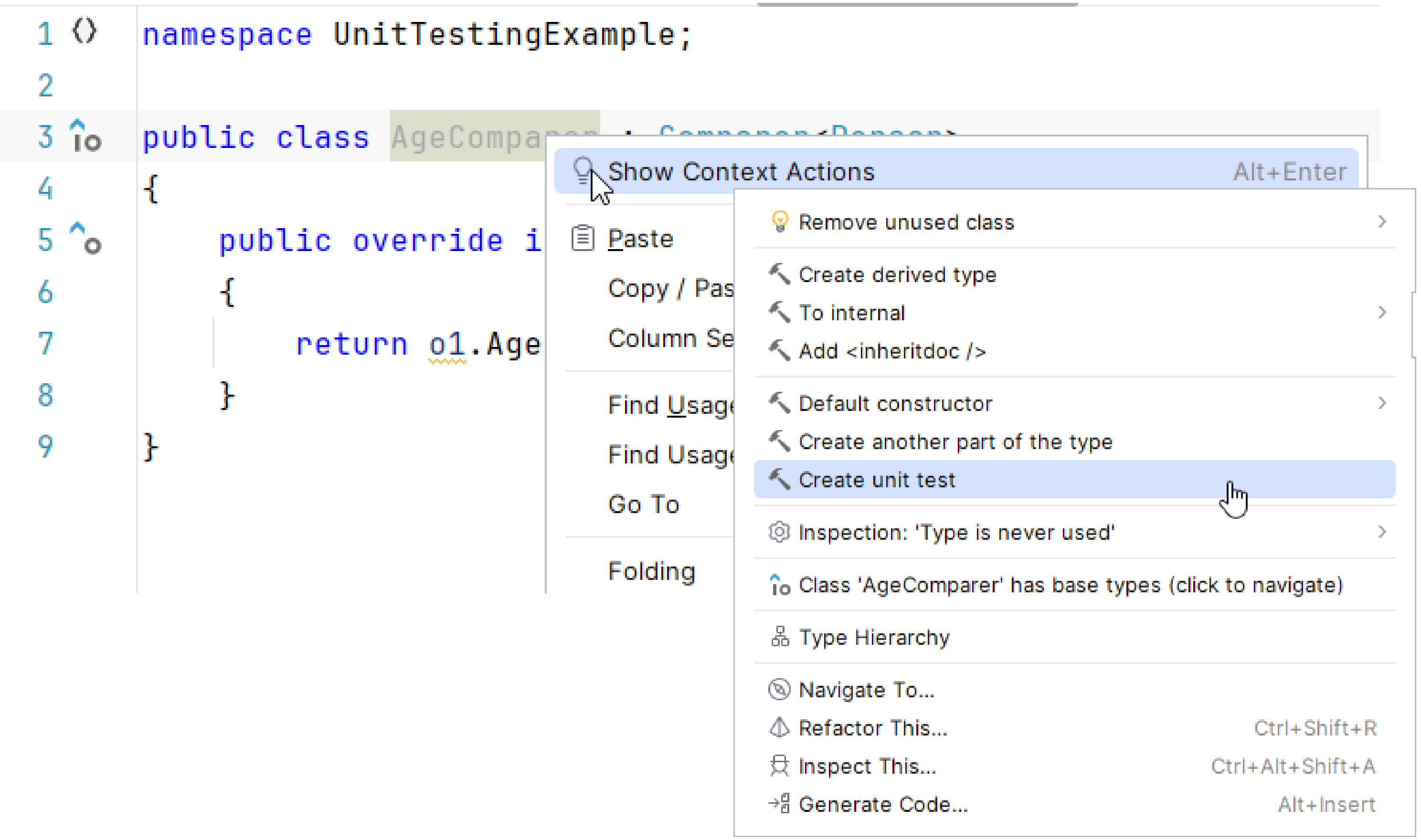
- After opening the project **“UnitTestingExample”**, open class **“AgeComparer.cs”.**
- Right-click on the name of the class **“AgeComparer”** in the file **AgeComparer.cs**
- Choose **“Generate”**
- Choose **“Unit Test”**



```
namespace UnitTestingExample;

public class AgeComparer : Comparer<Person>
{
    public override int Compare(Person? o1, Person? o2)
    {
        return o1.Age - o2.Age;
    }
}
```

Creating Test in Rider: Method # 2

- After opening the project **“UnitTestingExample”**, open class **“AgeComparer.cs”.**
- Right-click on the name of the class “**AgeComparer**” in the file **AgeComparer.cs**
- Choose “**Show Context Actions”**
- Choose **“Create unit test”**

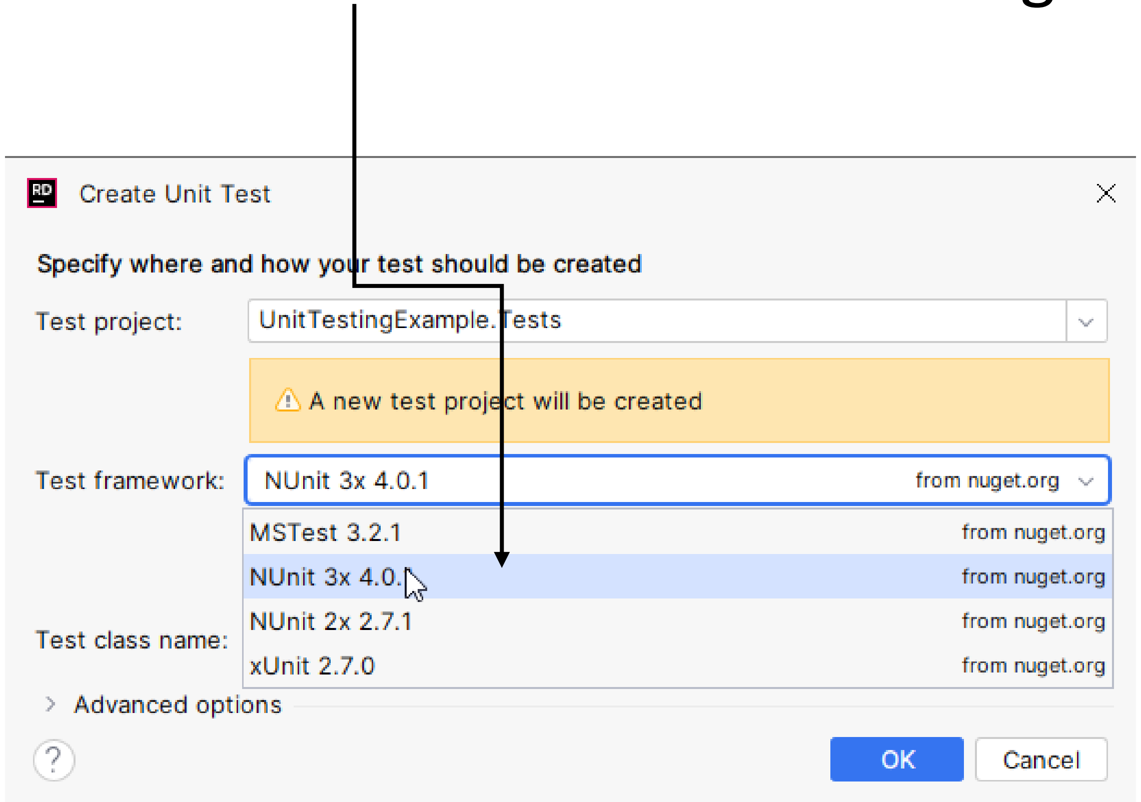


```
namespace UnitTestingExample;

public class AgeCompa
{
    public override i
    {
        return o1.Age
    }
}
```

Show Context Actions Alt+Enter
Paste
Copy / Pas
Column Se
Find Usag
Find Usag
Go To
Folding
Remove unused class
Create derived type
To internal
Add <inheritdoc />
Default constructor
Create another part of the type
Create unit test
Inspection: 'Type is never used'
Class 'AgeComparer' has base types (click to navigate)
Type Hierarchy
Navigate To...
Refactor This... Ctrl+Shift+R
Inspect This... Ctrl+Alt+Shift+A
Generate Code... Alt+Insert

Creating Test in Rider

- Select “**NUnit**” and click “**OK**” to generate the test file “**AgeComparerTest**”.



Create Unit Test

Specify where and how your test should be created

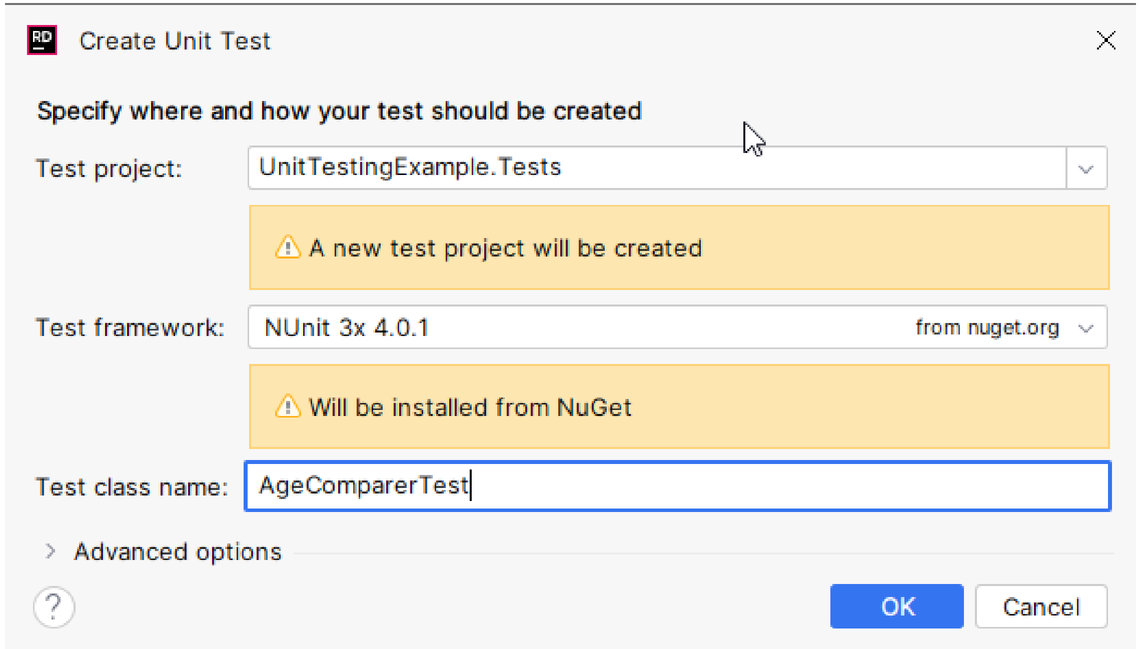
Test project: UnitTestingExample.Tests

⚠ A new test project will be created

Test framework: NUnit 3x 4.0.1 from nuget.org

Test class name:

MSTest 3.2.1 from nuget.org
NUnit 3x 4.0.1 from nuget.org
NUnit 2x 2.7.1 from nuget.org
xUnit 2.7.0 from nuget.org
Advanced options
OK
Cancel
?



Create Unit Test

Specify where and how your test should be created

Test project: UnitTestingExample.Tests

⚠ A new test project will be created

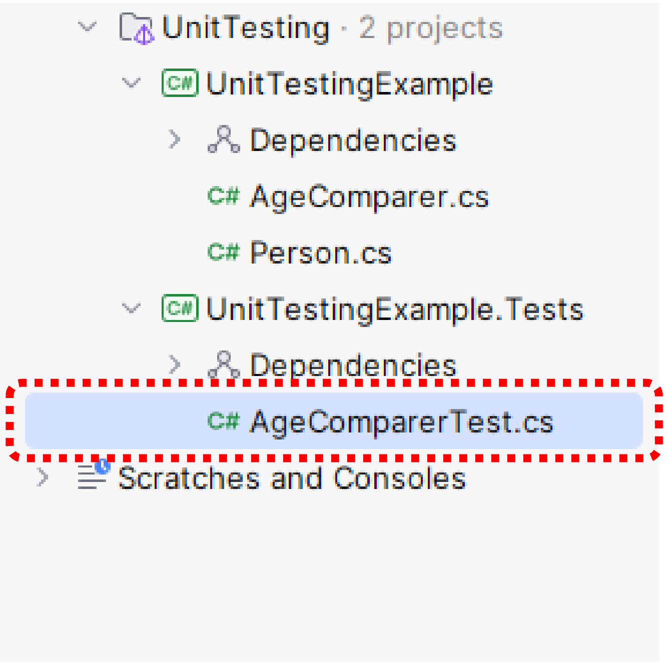
Test framework: NUnit 3x 4.0.1 from nuget.org

⚠ Will be installed from NuGet

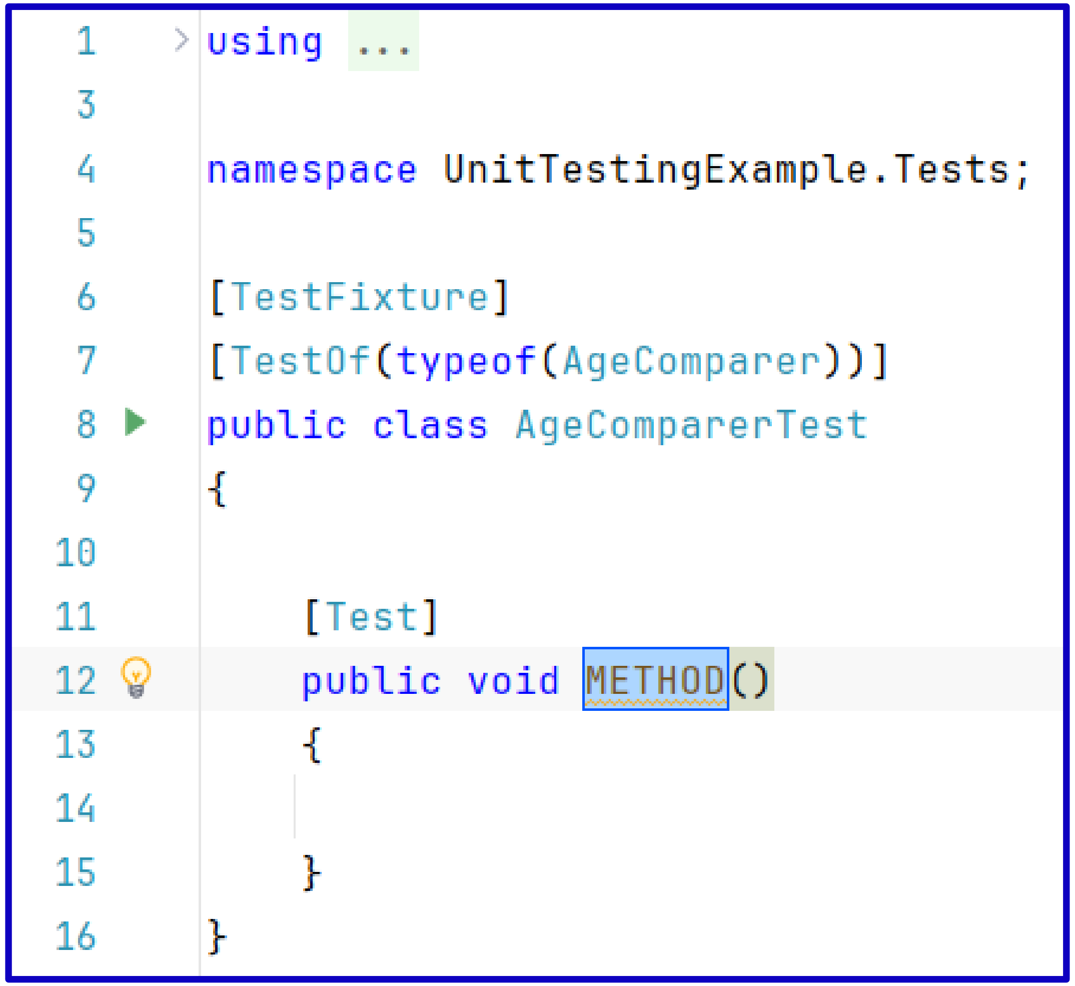
Test class name: AgeComparerTest

Creating Test in Rider

- On Clicking OK, a test file “**AgeComparerTest**” will be created.



```
UnitTesting · 2 projects
  UnitTestingExample
    Dependencies
    AgeComparer.cs
    Person.cs
  UnitTestingExample.Tests
    Dependencies
    AgeComparerTest.cs
Scratches and Consoles
```

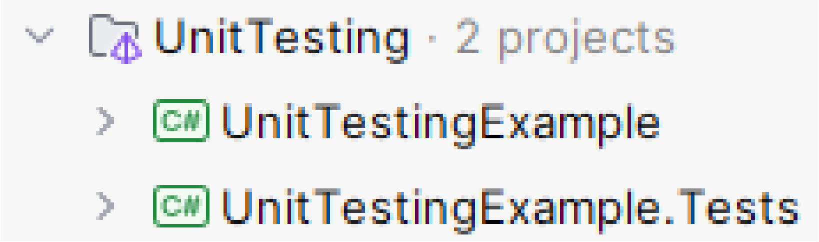


```
using ...

namespace UnitTestingExample.Tests;

[TestFixture]
[TestOf(typeof(AgeComparer))]
public class AgeComparerTest
{

    [Test]
    public void METHOD()
    {

    }
}
```

Where should the test be located?

Typically, unit tests are created in a separate project to keep the test code separate from the real code. The standard convention is to use two different projects in the same solution:

- <name> – for source code
- <name>.Tests – for test classes



UnitTesting · 2 projects

UnitTestingExample

UnitTestingExample.Tests

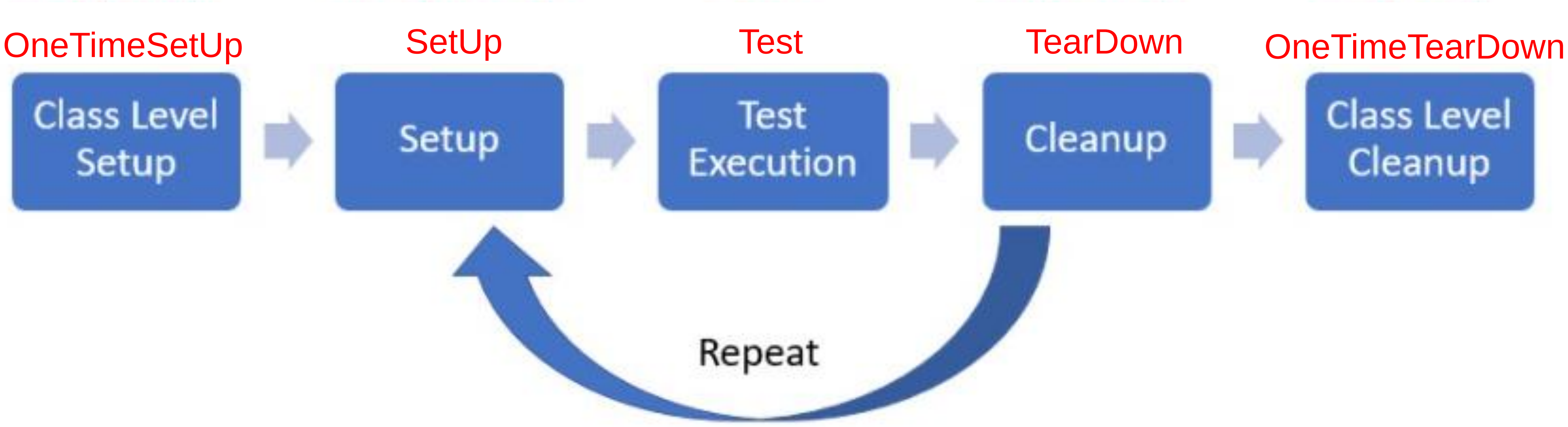
NUnit Attributes

https://docs.nunit.org/articles/nunit/writing-tests/attributes.html

NUnit Attributes

- Attributes are meta-tags that you can add to methods.
- NUnit uses attributes to mark methods as test methods and to configure them.
  - Test
  - SetUp
  - TearDown
  - OneTimeSetUp
  - OneTimeTearDown
  - Description
  - Ignore
  - Category
  - Repeat

NUnit Test Lifecyle Phases



OneTimeSetUp

SetUp

Test

TearDown

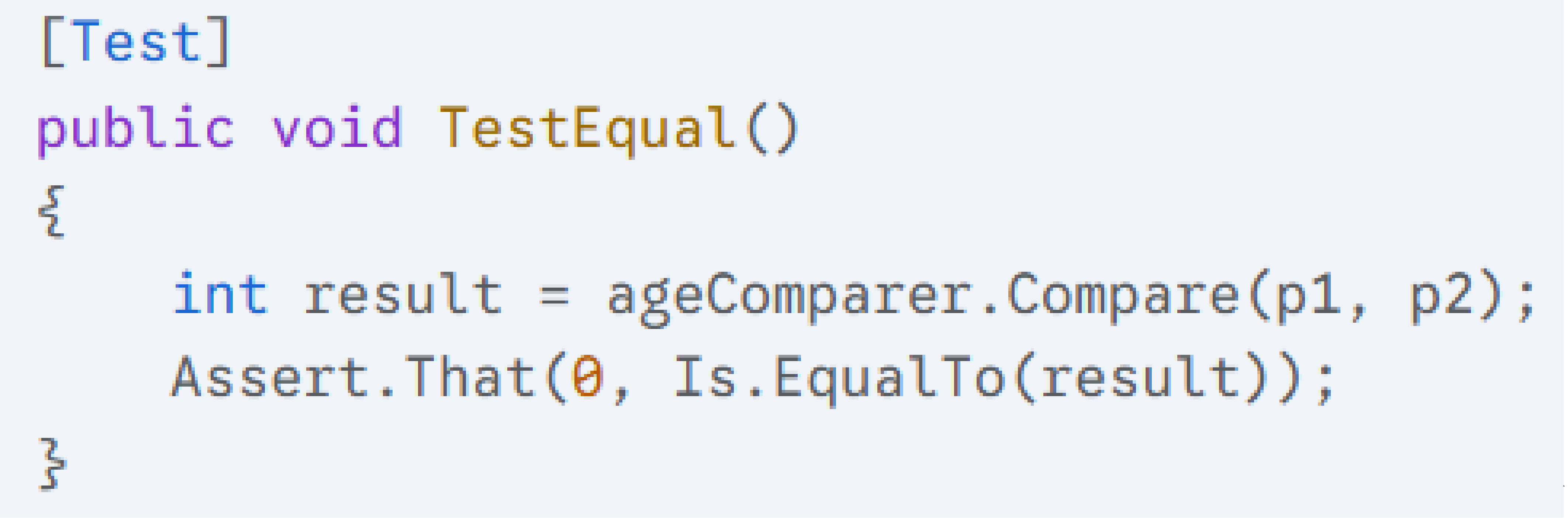
OneTimeTearDown

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->person_details.Tests-> **AgeComparatorTest.cs**

NUnit Annotations: Test

It is used to mark a method as a NUnit test.

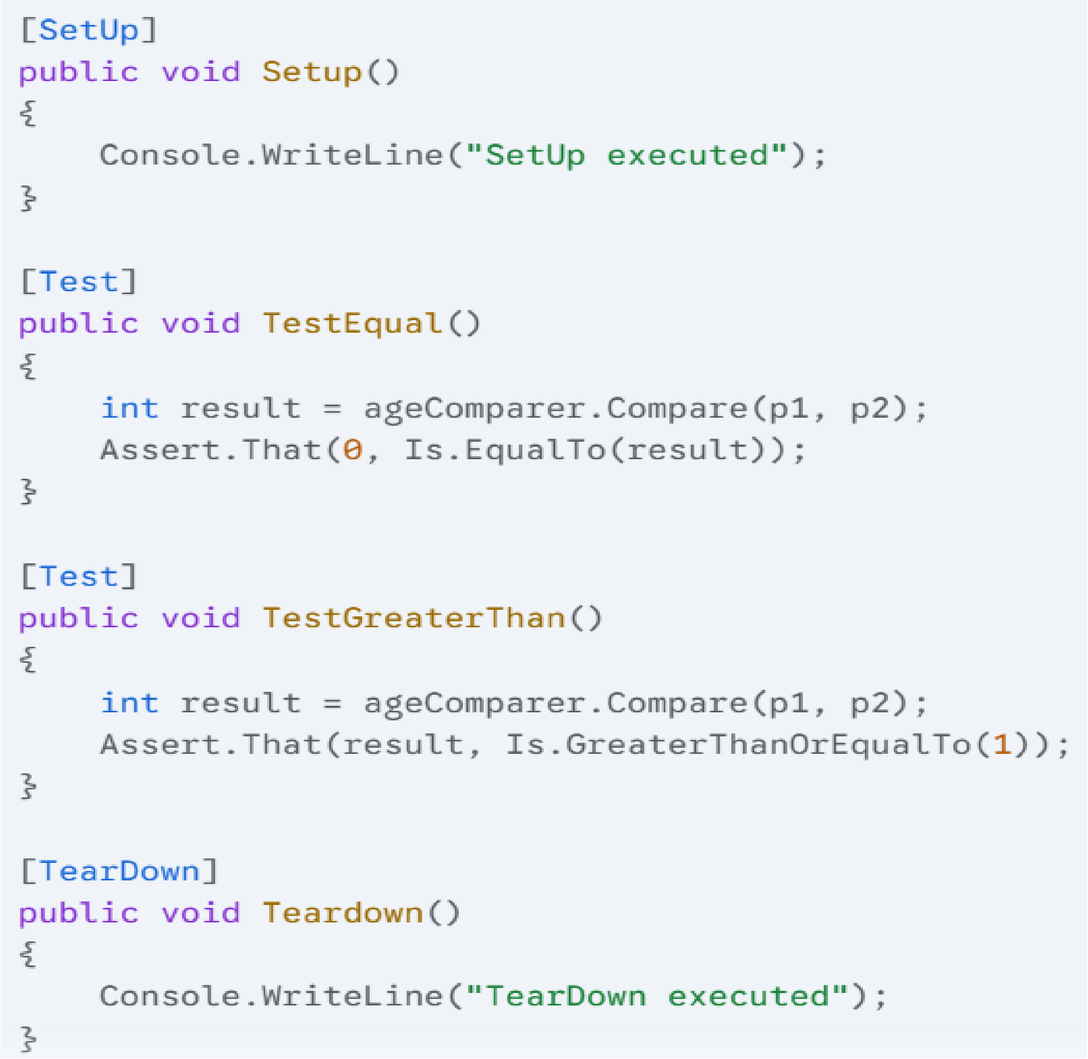
```
Person p1 = new Person("Ali", 112233, 29);
Person p2 = new Person("Joy", 112234, 29);
AgeComparator ageComparer = new AgeComparator();
```



```
[Test]
public void TestEqual()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(0, Is.EqualTo(result));
}
```

NUnit Attributes: Setup/Teardown

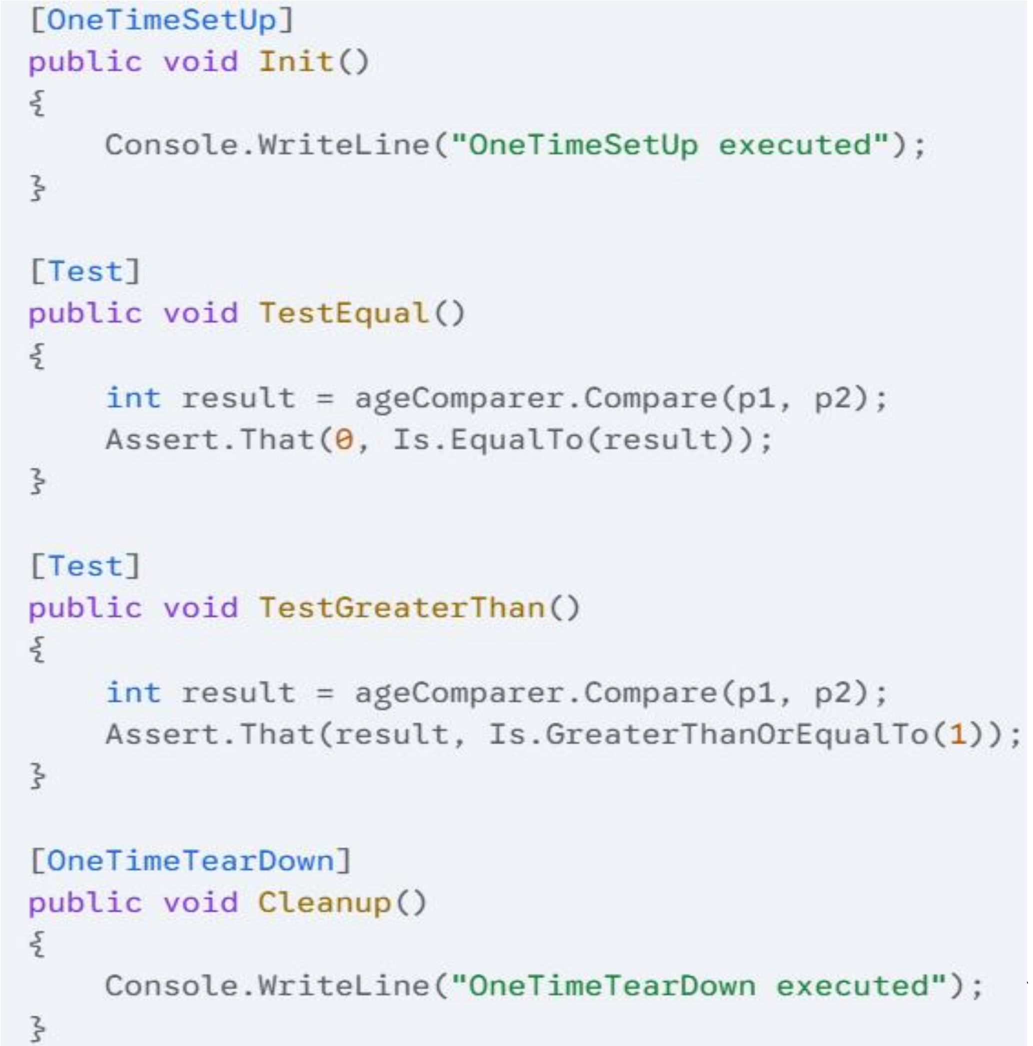
Setup and Teardown attributes cause the method to be run before and after each Test method respectively



```
[SetUp]
public void Setup()
{
    Console.WriteLine("SetUp executed");
}

[Test]
public void TestEqual()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(0, Is.EqualTo(result));
}

[Test]
public void TestGreaterThan()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(result, Is.GreaterThanOrEqualTo(1));
}

[TearDown]
public void Teardown()
{
    Console.WriteLine("TearDown executed");
}
```

NUnit Attributes: OneTimeSetUp / OneTimeTearDown

- OneTimeSetUp and OneTimeTearDown attributes cause the method to be run once in the entire execution cycle.



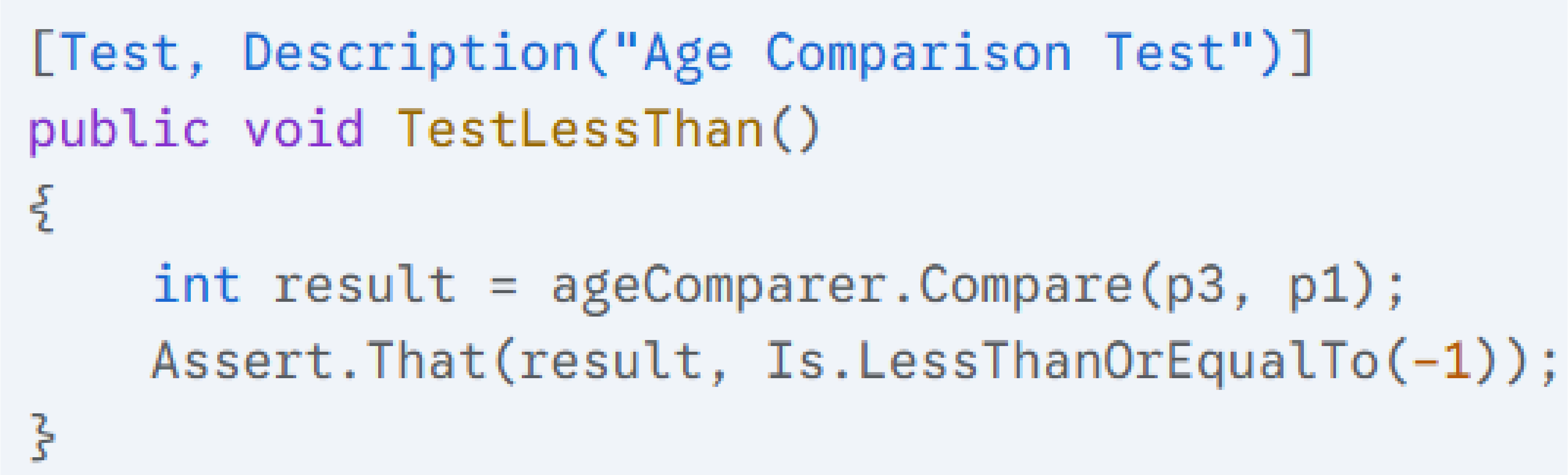
```
[OneTimeSetUp]
public void Init()
{
    Console.WriteLine("OneTimeSetUp executed");
}

[Test]
public void TestEqual()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(0, Is.EqualTo(result));
}

[Test]
public void TestGreaterThan()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(result, Is.GreaterThanOrEqualTo(1));
}

[OneTimeTearDown]
public void Cleanup()
{
    Console.WriteLine("OneTimeTearDown executed");
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->person_details.Tests-> **AgeComparatorTest.cs**

NUnit Attributes: Description

It is used to provide any custom display name for a test class or test method

```
Person p1 = new Person("Ali", 112233, 29);
Person p3 = new Person(“Peter", 112235, 10);
AgeComparator ageComparer = new AgeComparator();
```

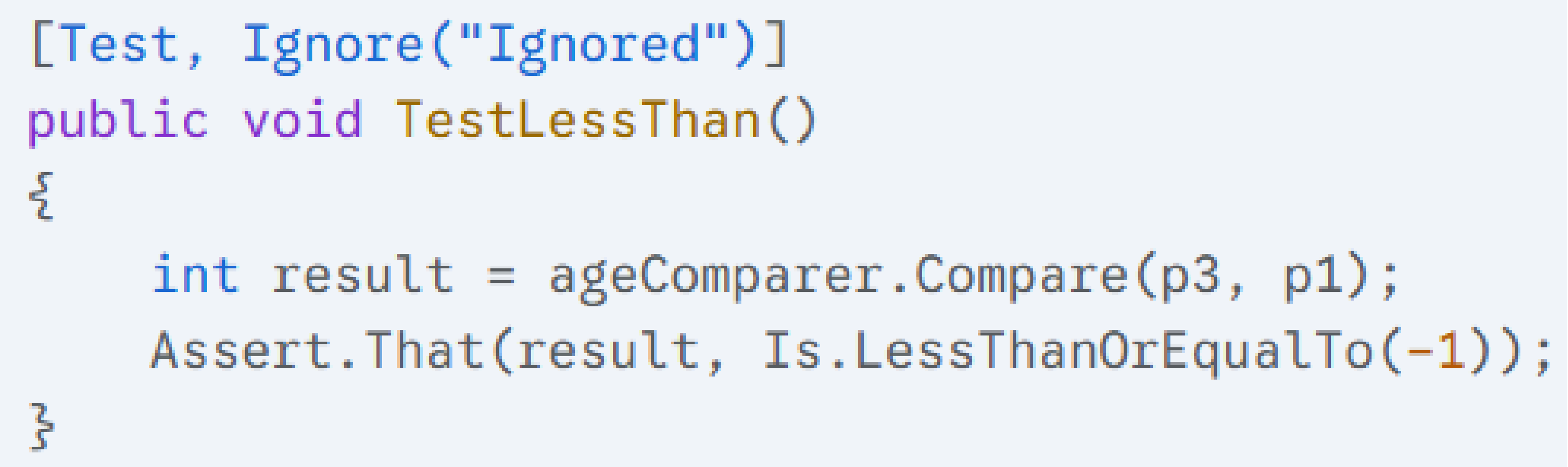


```
[Test, Description("Age Comparison Test")]
public void TestLessThan()
{
    int result = ageComparer.Compare(p3, p1);
    Assert.That(result, Is.LessThanOrEqualTo(-1));
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->person_details.Tests-> **AgeComparatorTest.cs**

NUnit Attributes: Ignore

It is used to ignore/disable test class (disables all test methods in that class) or individual test methods.

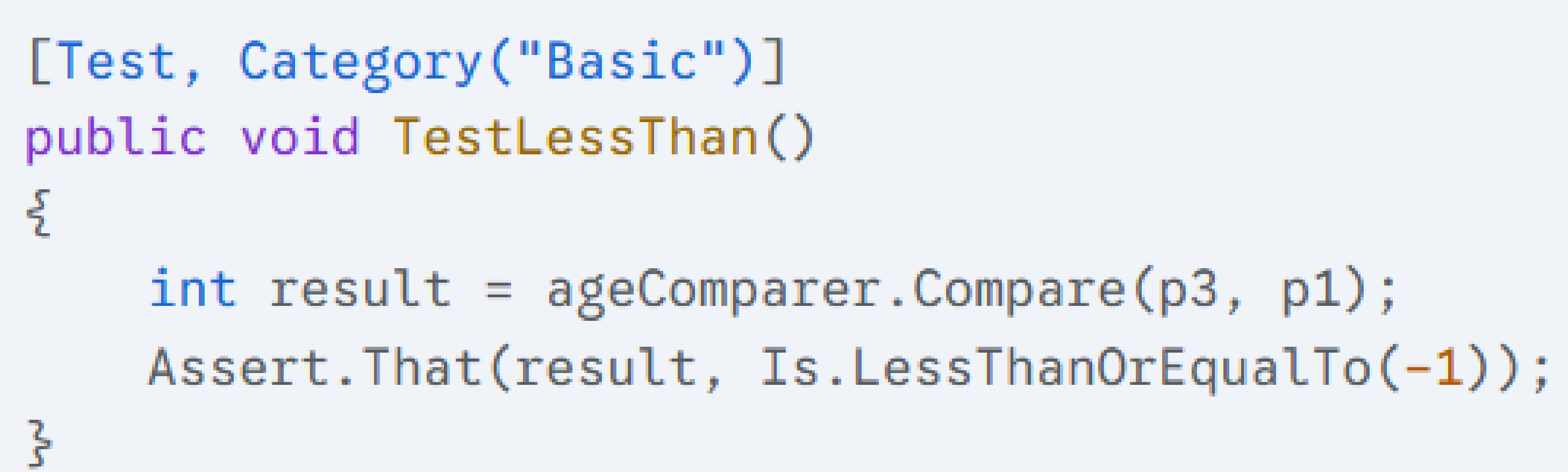


```
[Test, Ignore("Ignored")]
public void TestLessThan()
{
    int result = ageComparer.Compare(p3, p1);
    Assert.That(result, Is.LessThanOrEqualTo(-1));
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->person_details.Tests-> **AgeComparatorTest.cs**

NUnit Attributes: Category

- Category can be used to filter testcases and gives you fine-grained control over test execution.
- When categories are used, only the tests in the selected categories will be run.

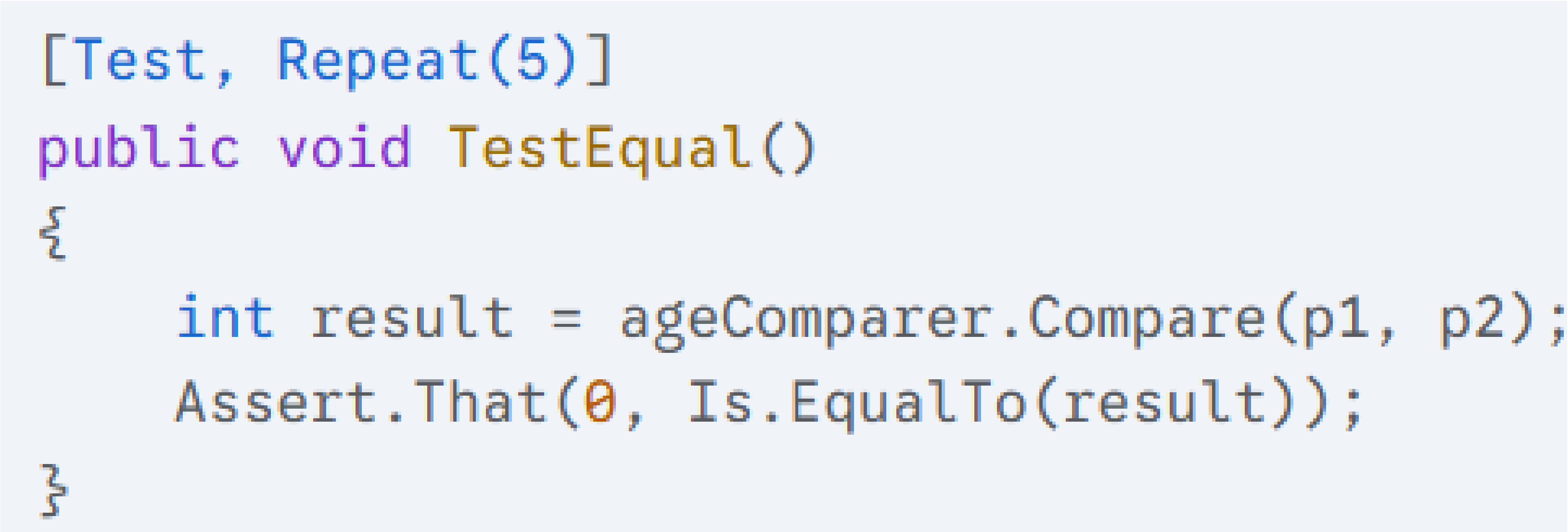


```
[Test, Category("Basic")]
public void TestLessThan()
{
    int result = ageComparer.Compare(p3, p1);
    Assert.That(result, Is.LessThanOrEqualTo(-1));
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->person_details.Tests-> **AgeComparatorTest.cs**

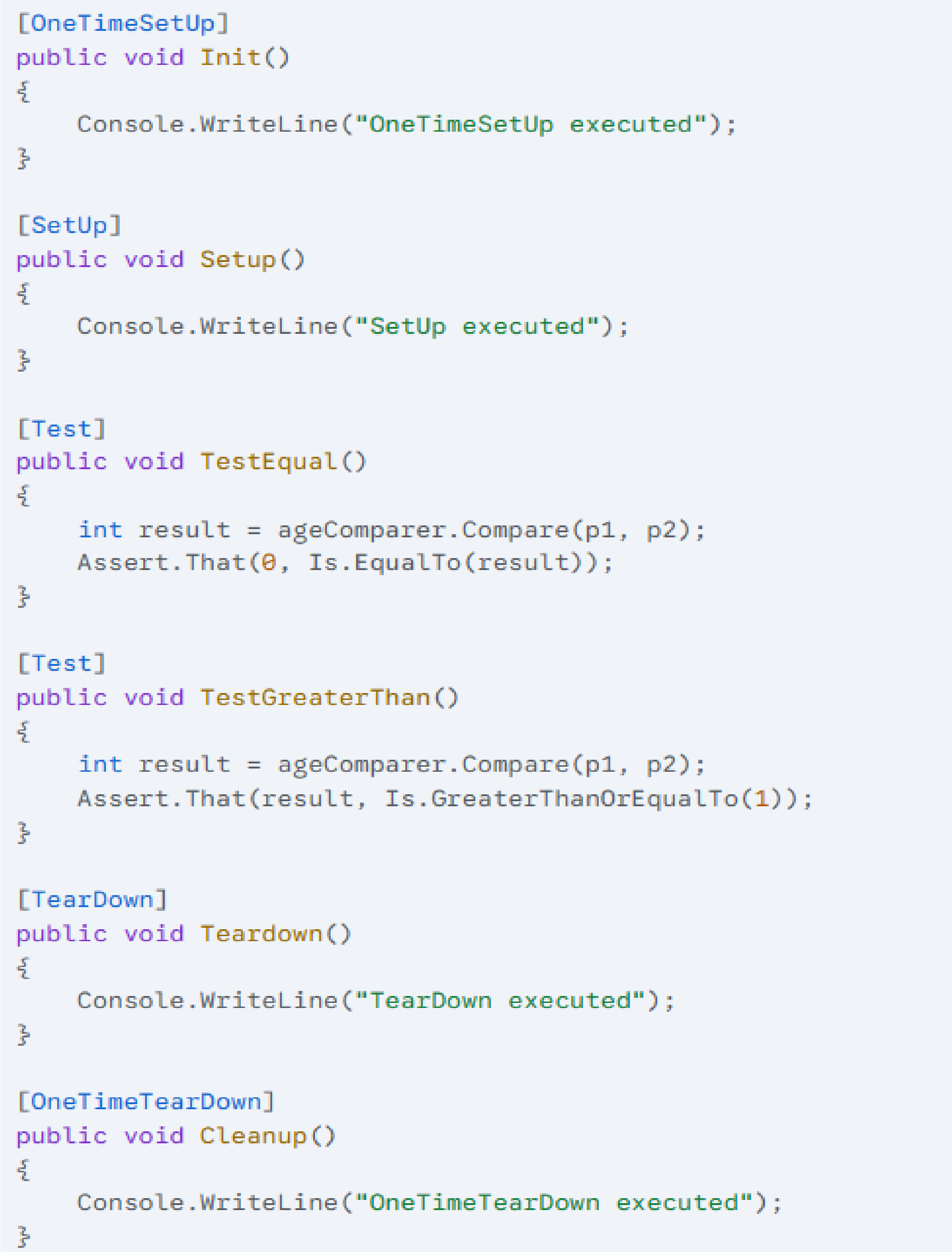
NUnit Attributes: Repeat

- It is used to repeat the Test N number of time by passing the number as an argument to the annotation



```
[Test, Repeat(5)]
public void TestEqual()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(0, Is.EqualTo(result));
}
```

NUnit Attributes: Example



```
[OneTimeSetUp]
public void Init()
{
    Console.WriteLine("OneTimeSetUp executed");
}

[SetUp]
public void Setup()
{
    Console.WriteLine("SetUp executed");
}

[Test]
public void TestEqual()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(0, Is.EqualTo(result));
}

[Test]
public void TestGreaterThan()
{
    int result = ageComparer.Compare(p1, p2);
    Assert.That(result, Is.GreaterThanOrEqualTo(1));
}

[TearDown]
public void Teardown()
{
    Console.WriteLine("TearDown executed");
}

[OneTimeTearDown]
public void Cleanup()
{
    Console.WriteLine("OneTimeTearDown executed");
}
```

```
OneTimeSetUp executed
SetUp executed
====== Equal TEST EXECUTED =======
TearDown executed
SetUp executed
======GreaterThan TEST EXECUTED =======
TearDown executed
OneTimeTearDown executed
```

What will be the execution flow, If Ignore is added to **GreaterThan** method????

NUnit Assertions

https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertions.html

NUnit – Assert Class

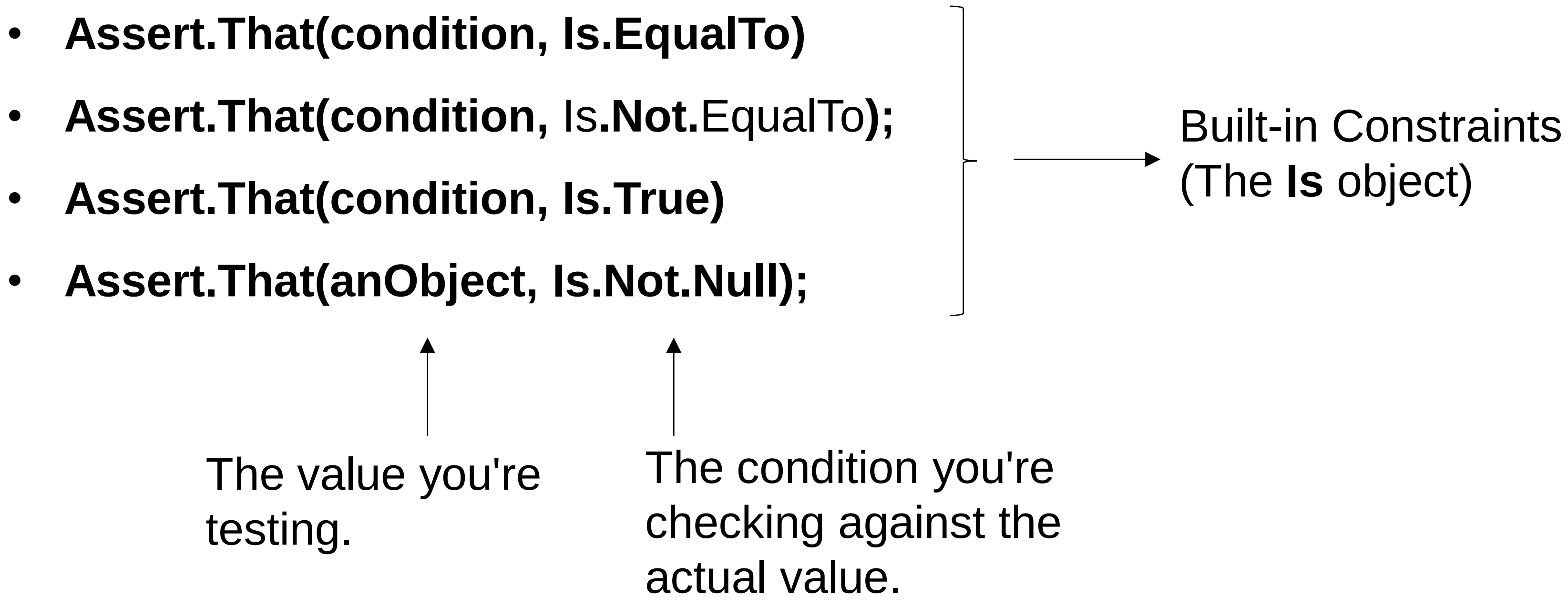
- In the old version of NUnit, the Assert class provided a set of assertion methods useful for testing certain conditions.
- An assert method compares the actual value returned by a test to the expected value.
  - **Assert.Equals**
  - **Assert.True**
  - **Etc.**

https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertion-models/classic.html

NUnit – Constraint Model

- In the newer versions of NUnit, it has moved over to the constraint model. This takes constraint objects as a argument

- Everything has been moved in to **Assert.That**(), wherein constraint objects are used:



  - **Assert.That(condition, Is.EqualTo)**
  - **Assert.That(condition,** Is**.Not.**EqualTo**);**
  - **Assert.That(condition, Is.True)**
  - **Assert.That(anObject, Is.Not.Null);**

Built-in Constraints
(The **Is** object)

The value you're testing.

The condition you're checking against the actual value.

https://docs.nunit.org/articles/nunit/writing-tests/constraints/Constraints.html

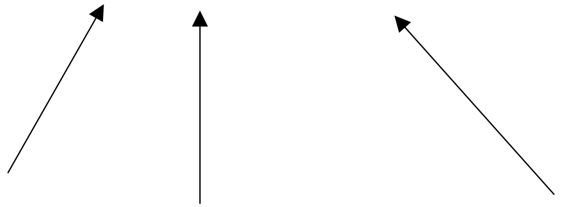
NUnit Constraints – Equal / Not Equal

- Asserts that actual value and expected value are **equal**.

```
Assert.That(int actual, Is.EqualTo(int expected));
```

- Asserts that actual value and expected value are **NOT** equal.

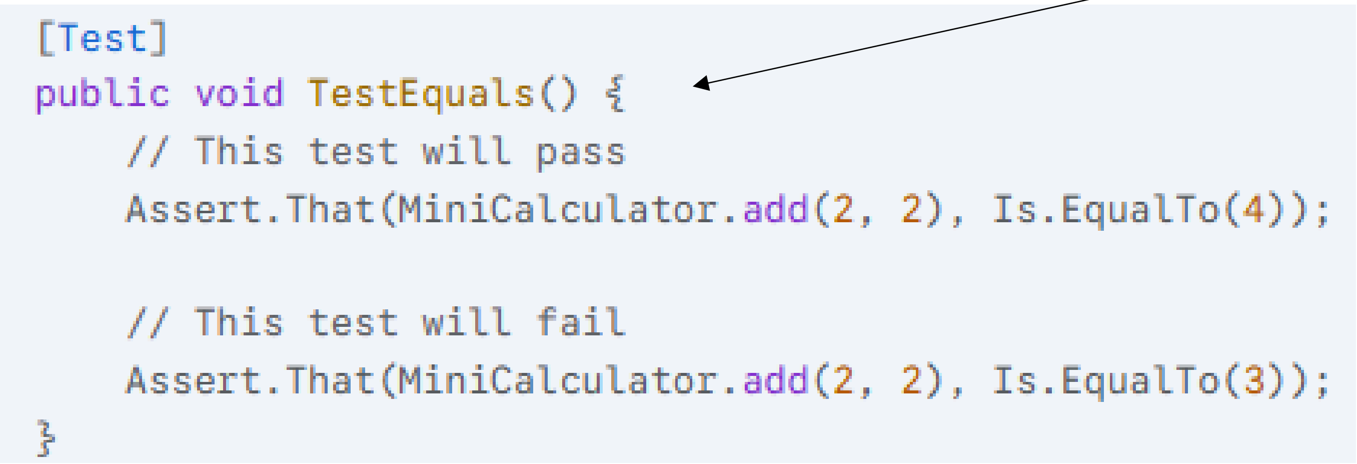
```
Assert.That(int actual, Is.Not.EqualTo(int expected));
```



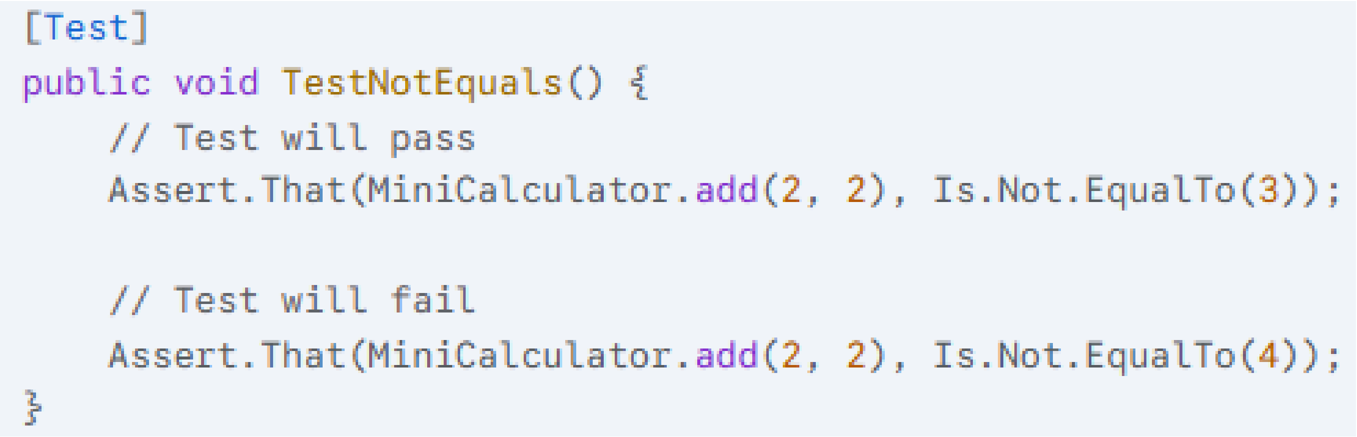
Is Object Negation Operator EqualTo Constraint

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->calculator.Tests-> **MiniCalculatorTest.cs**

Example



```
[Test]
public void TestEquals() {
    // This test will pass
    Assert.That(MiniCalculator.add(2, 2), Is.EqualTo(4));

    // This test will fail
    Assert.That(MiniCalculator.add(2, 2), Is.EqualTo(3));
}
```



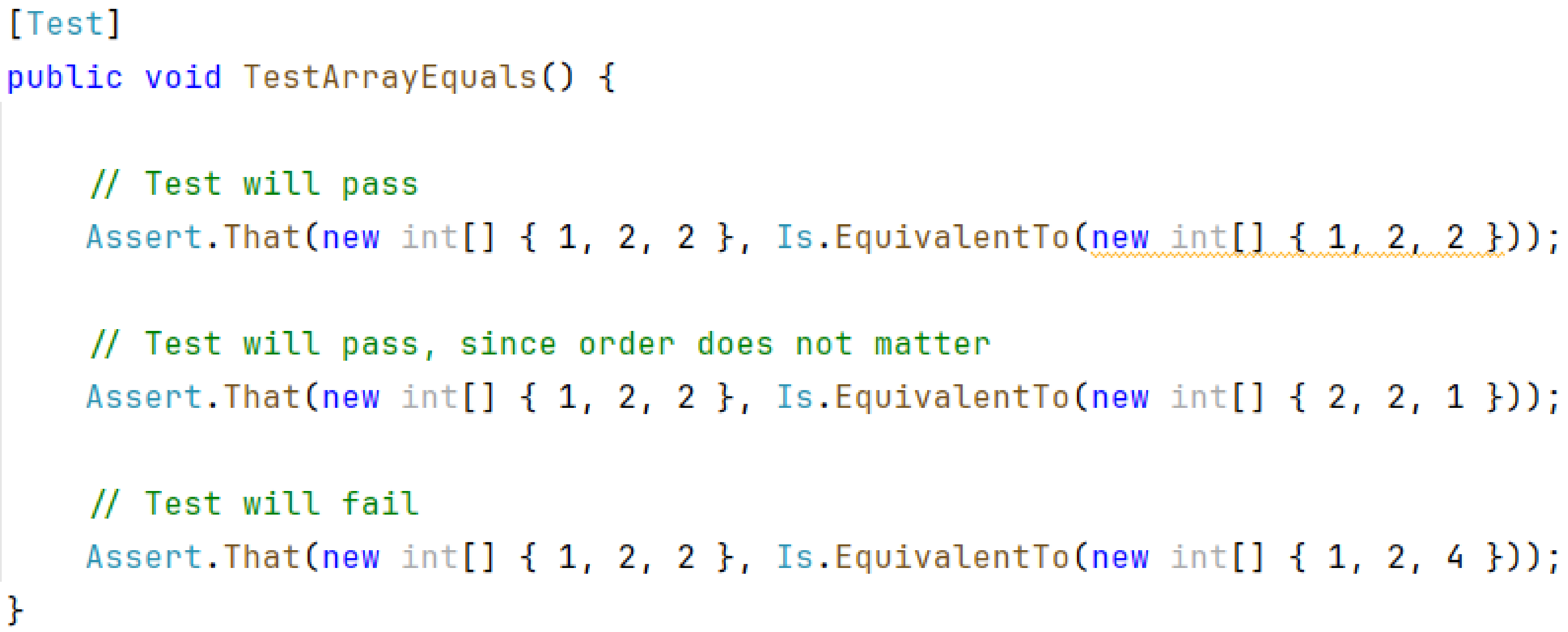
```
[Test]
public void TestNotEquals() {
    // Test will pass
    Assert.That(MiniCalculator.add(2, 2), Is.Not.EqualTo(3));

    // Test will fail
    Assert.That(MiniCalculator.add(2, 2), Is.Not.EqualTo(4));
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

NUnit Constraints – CollectionEquivalent

- Asserts that actual and expected arrays are **equivalent.**

- Only checks if items are the same, not order



```
[Test]
public void TestArrayEquals() {

    // Test will pass
    Assert.That(new int[] { 1, 2, 2 }, Is.EquivalentTo(new int[] { 1, 2, 2 }));

    // Test will pass, since order does not matter
    Assert.That(new int[] { 1, 2, 2 }, Is.EquivalentTo(new int[] { 2, 2, 1 }));

    // Test will fail
    Assert.That(new int[] { 1, 2, 2 }, Is.EquivalentTo(new int[] { 1, 2, 4 }));
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

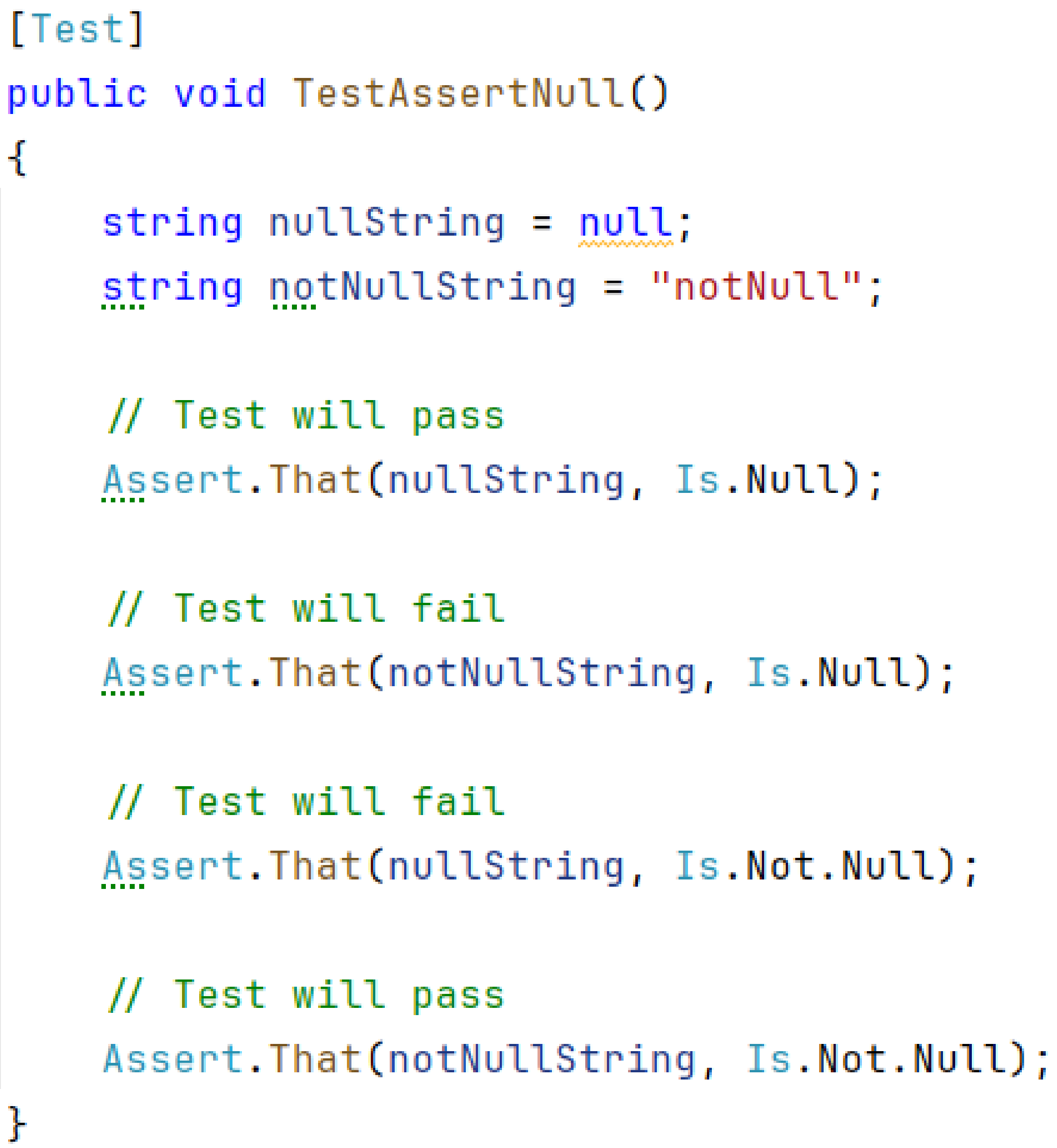
NUnit Constraints –Null/Not Null

- Asserts that anObject is **null.**
  - Assert.That(anObject, **Is.Null**);

- Asserts that anObject is **NOT null.**
  - Assert.That(anObject, **Is.Not.Null**);

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

Example



```
[Test]
public void TestAssertNull()
{
    string nullString = null;
    string notNullString = "notNull";

    // Test will pass
    Assert.That(nullString, Is.Null);

    // Test will fail
    Assert.That(notNullString, Is.Null);

    // Test will fail
    Assert.That(nullString, Is.Not.Null);

    // Test will pass
    Assert.That(notNullString, Is.Not.Null);
}
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

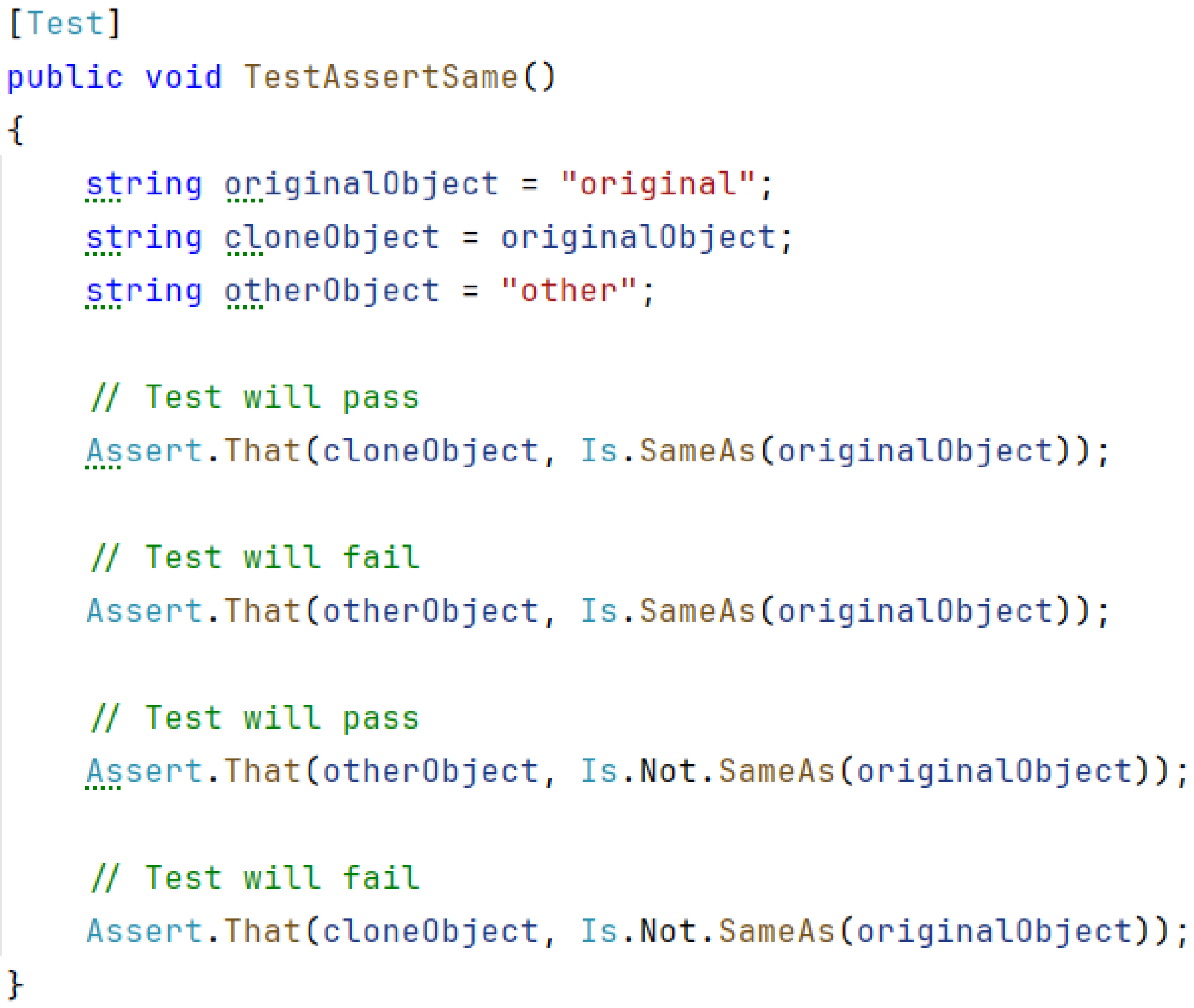
NUnit Constraints – SameAs/Not SameAs

- Asserts that actual **and** expected refer to the same object.
  - Assert.That(object actual, **Is.SameAs**(object expected))

- Asserts that actual **and** expected **DO NOT** refer to the same object.
  - Assert.That(object actual, **Is.Not.SameAs**(object expected))

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

Example



```
[Test]
public void TestAssertSame()
{
    string originalObject = "original";
    string cloneObject = originalObject;
    string otherObject = "other";

    // Test will pass
    Assert.That(cloneObject, Is.SameAs(originalObject));

    // Test will fail
    Assert.That(otherObject, Is.SameAs(originalObject));

    // Test will pass
    Assert.That(otherObject, Is.Not.SameAs(originalObject));

    // Test will fail
    Assert.That(cloneObject, Is.Not.SameAs(originalObject));
}
```

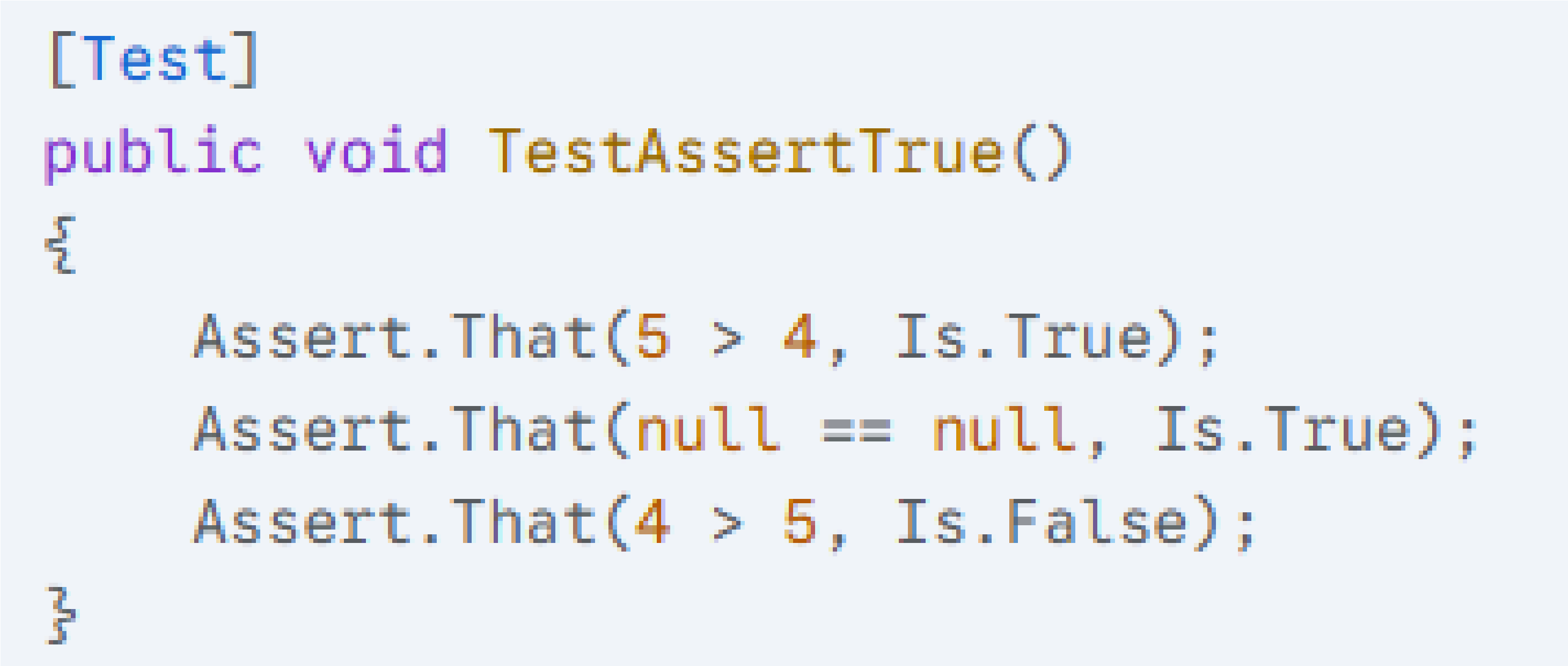
Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

NUnit Constraints–True/False

- Asserts that the supplied condition is **true**
  - Assert.That(condition, **Is.True**);
- Asserts that the supplied condition is **false**
  - Assert.That(condition, **Is.False**);

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->Misc.Test-> **MiscTest.cs**

Example



```
[Test]
public void TestAssertTrue()
{
    Assert.That(5 > 4, Is.True);
    Assert.That(null == null, Is.True);
    Assert.That(4 > 5, Is.False);
}
```

NUnit Constraints – Throws

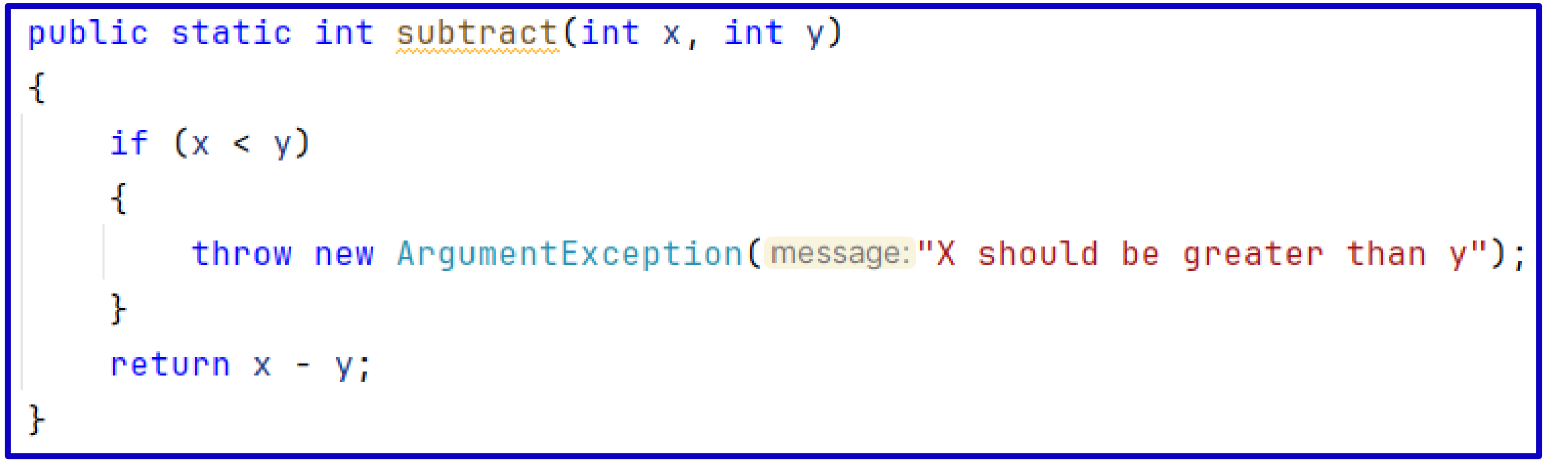
- It is used to write the test code that is expected to throw an exception
- Asserts that execution of the supplied executable block or lambda expression throws an exception of the expectedType.

```
ThrowsConstraint(Type expectedType)
```

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->calculator.Tests-> **MiniCalculatorTest.cs**

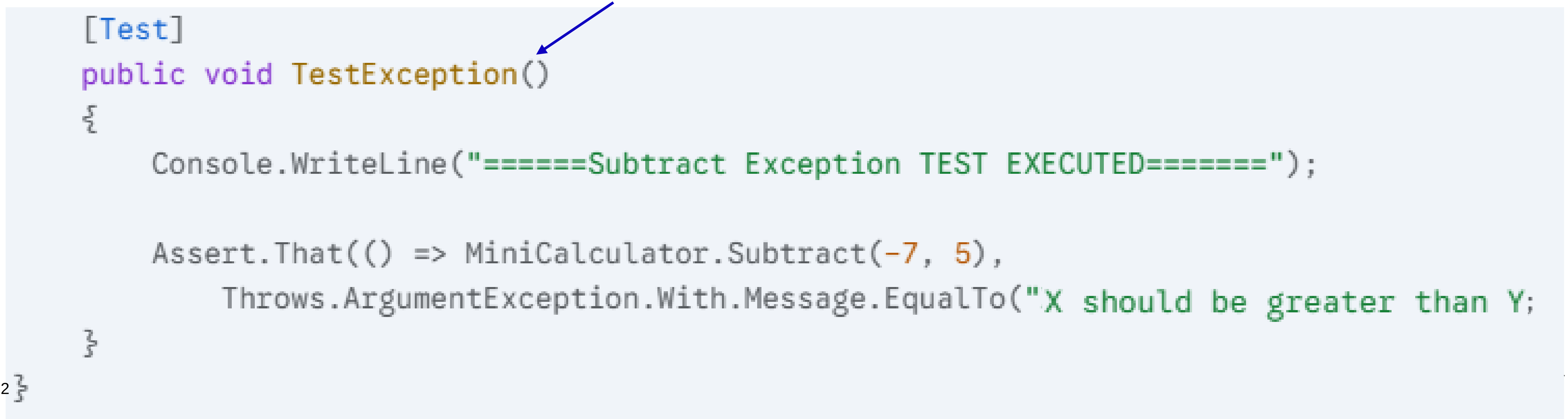
Example

Method in MiniCalculator Class



```
public static int subtract(int x, int y)
{
    if (x < y)
    {
        throw new ArgumentException(message:"X should be greater than y");
    }
    return x - y;
}
```

Test Method in MiniCalculatorTest Class



```
    [Test]
    public void TestException()
    {
        Console.WriteLine("======Subtract Exception TEST EXECUTED=======");

        Assert.That(() => MiniCalculator.Subtract(-7, 5),
            Throws.ArgumentException.With.Message.EqualTo("X should be greater than Y;
    }
}
```

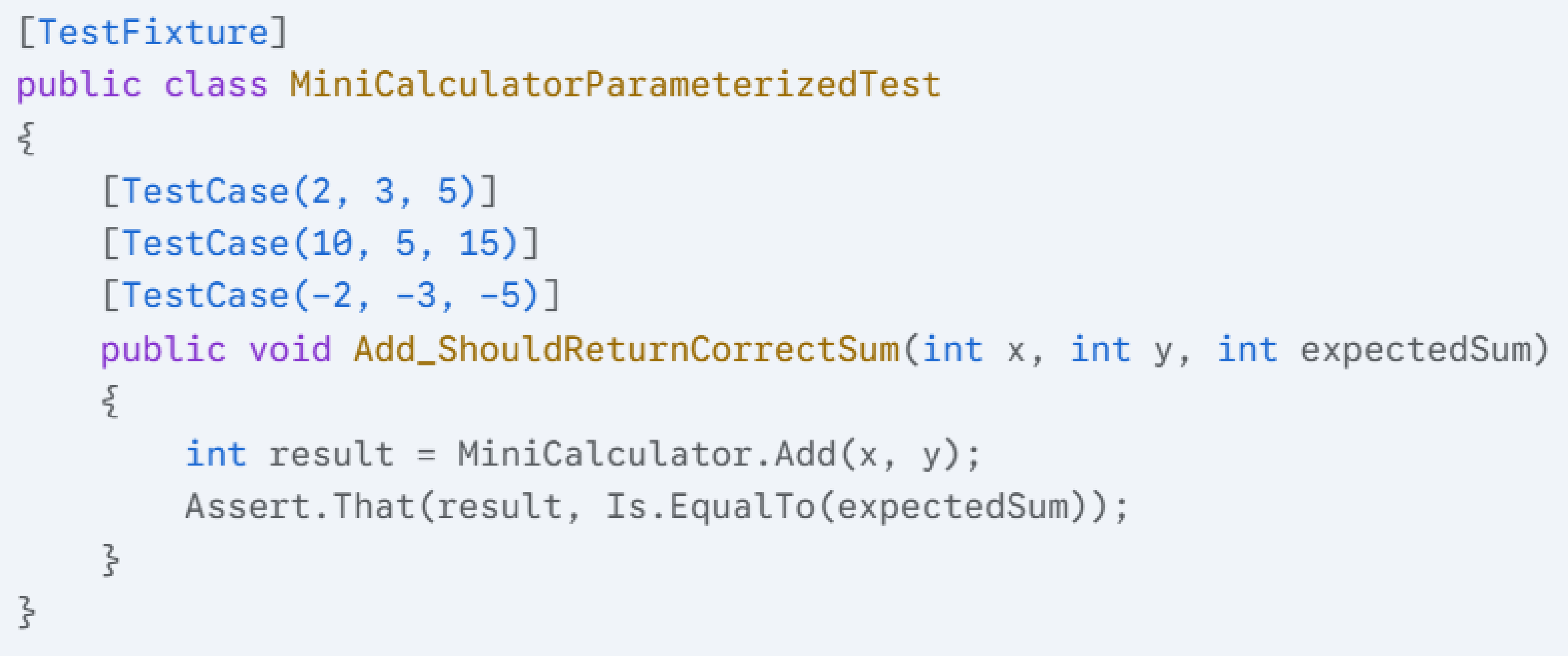
Break 10 min

Writing Parameterized Tests

Allows running a single test with multiple sets of input data using the [TestCase] attribute.

Example

Plans-> Topics -> VOP-4->VOP-4 (Lecture)-> Resources and Activities-> UnitTestingApp->calculator.Tests-> **MiniCalculatorParameterizedTest.cs**



```
[TestFixture]
public class MiniCalculatorParameterizedTest
{
    [TestCase(2, 3, 5)]
    [TestCase(10, 5, 15)]
    [TestCase(-2, -3, -5)]
    public void Add_ShouldReturnCorrectSum(int x, int y, int expectedSum)
    {
        int result = MiniCalculator.Add(x, y);
        Assert.That(result, Is.EqualTo(expectedSum));
    }
}
```

MCQs Quiz

Go to Plans -> VOP-4 -> VOP-4 (Lecture) -> Unit Testing Test

Good Luck ☺

HomeWork Exercise

Go to Plans -> VOP-4 -> VOP-4 (Lecture) -> HomeWork.zip

Good Luck ☺

HomeWork.zip (Solution is also provided)

- Unzip HomeWork.zip and locate ReadMe.md. The ReadMe.md comprises the following instruction:
- Locate/Find a class *“ColorBag.cs”* in Exercise folder.
- Create a test class *“ColorBagTest”* in exercise.Test directory.
- Add SetUp/TearDown methods in the test class
- Add 5 test methods in the test class to test all the 5 instance methods in the class *“ColorBag.cs”.*
- *Declare a variable of type ColorBag as ColorBag colorBag in the class “ColorBagTest”*
- Create an instance of ColorBag and add the following 6 colors in ColorBag in the method annotated with *SetUp* in the class “*ColorBagTest”*

red, green, yellow, blue, magenta, brown

HomeWork.zip

- Use the appropriate assert methods and annotations to create the tests
  - Add “pink” color in the *ColorBag* in the *AddColor() test* method and check if the pink color is added in the *ColorBag*
  - Remove “brown” color from the *ColorBag* in the RemoveColor() test method and check if the brown color is removed from the *ColorBag*
  - In the *ContainsColor()* test method, check if **“red”** color is in the *ColorBag*
  - In the *IsBagEmpty() test* method, check if the bag is not empty
  - In the *Size() test* method, check if the size of the *ColorBag* is correct, i.e., 6