Object Oriented Development

# TDD and Collections Walkthrough

# What does this walkthrough cover?

This walkthrough will introduce you to Test Driven Development in C# in addition to C# Collections.

# How long will the walkthrough take to complete?

3-4 hours

# What should you have already completed?

C# Day 1 Walkthrough and associated pre-requisites

# What do you need?

In order to complete this tutorial exercise you will need:

* Visual Studio 2013 Community Edition
* NUnit or MS Test
* Source Control

# What does this walkthrough cover?

* A further look at Visual Studio with NuGet and some of its additional functionality
* Recap of the theory behind Test Driven Development
* An introduction to Test Driven Development using NUnit
* An introduction to Collections in C#

# Test Driven Development (TDD)

The idea behind Test Driven Development is that, before we write any code, we create a series of tests that will use our code. This drives the development of our production code with tests to ensure that **every line of code we write adds value to our solution.**

We write each test based on the user specification and then write code to pass that test; doing this, we should never write speculative code that we *think* might be useful at a later date.

## The 5 Steps of TDD

1. Write the test
2. Do just enough to make the test compile
3. Watch the test fail
4. Do just enough to make the test pass
5. Refactor and generalise

Following these steps allows us to work up to our complete solution, ensuring that each piece we build does what we need it to do (but no more) before we move on.

## Qualities of a Good Test

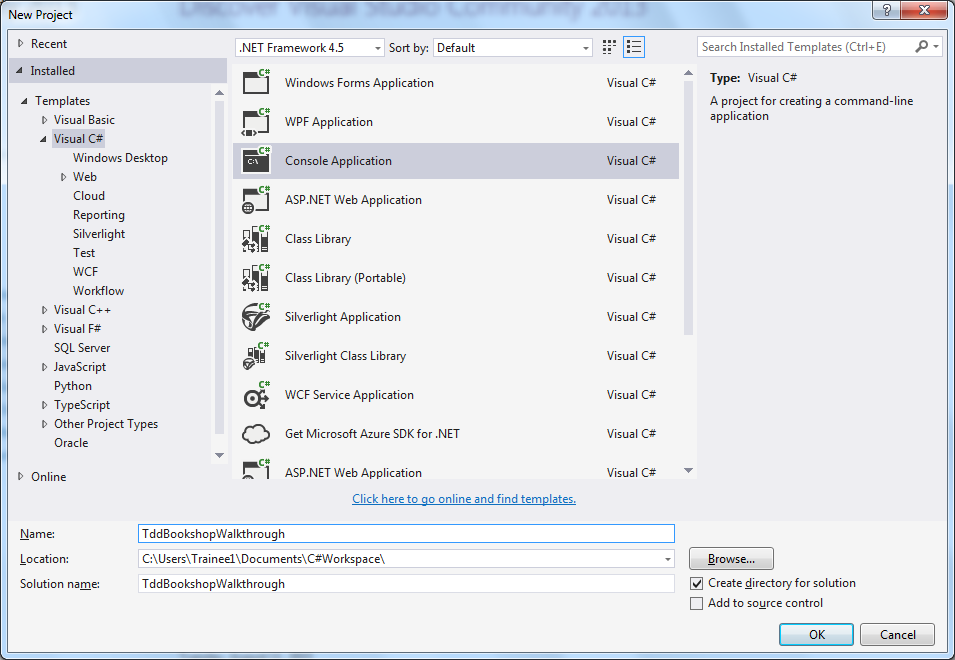
A good test should be:

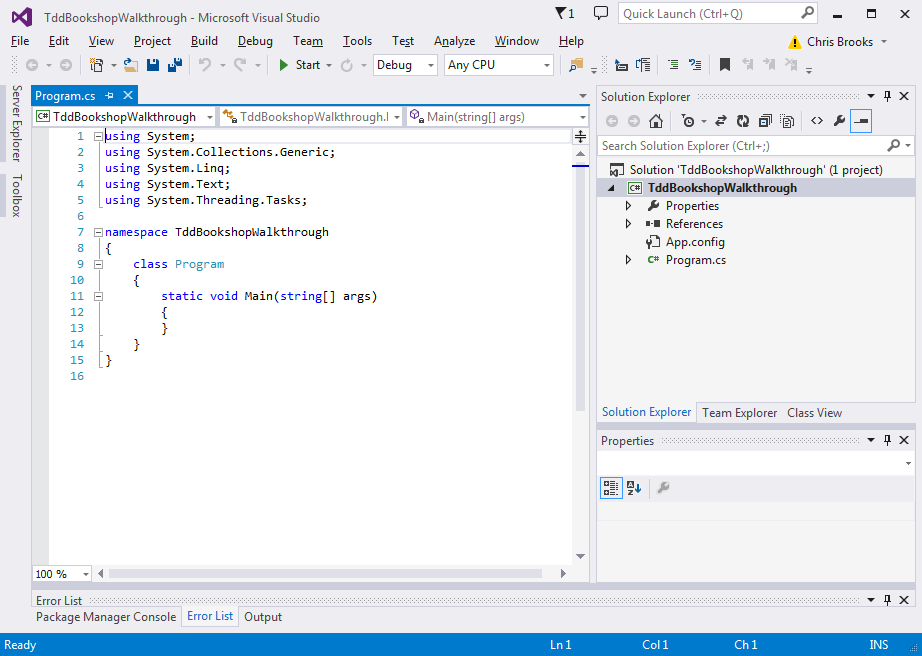
1. Focused – A test should test only one thing
2. Easy to read – A test name should be self-documenting and make it clear what the test is doing
3. Simple – A test should consist of a sequence only; it should not contain loops and decisions
4. Independent – A test should stand alone and not depend on any other test
5. Flexible – A test and the code it is testing should be able to be re-used in different projects without having to change anything

**Discussion Point: Why is each of the above qualities important? What advantages does following them bring?**

# TDD Example

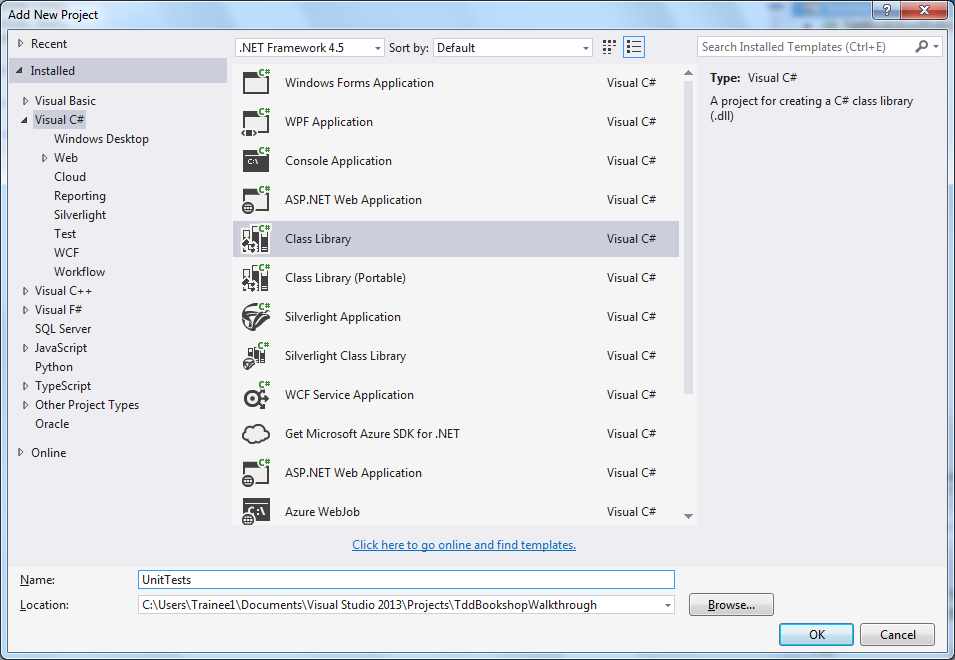
Create a new Console Application project named TddBookShopWalkthrough





Right-click on the solution and select Add -> New Project -> Class Library

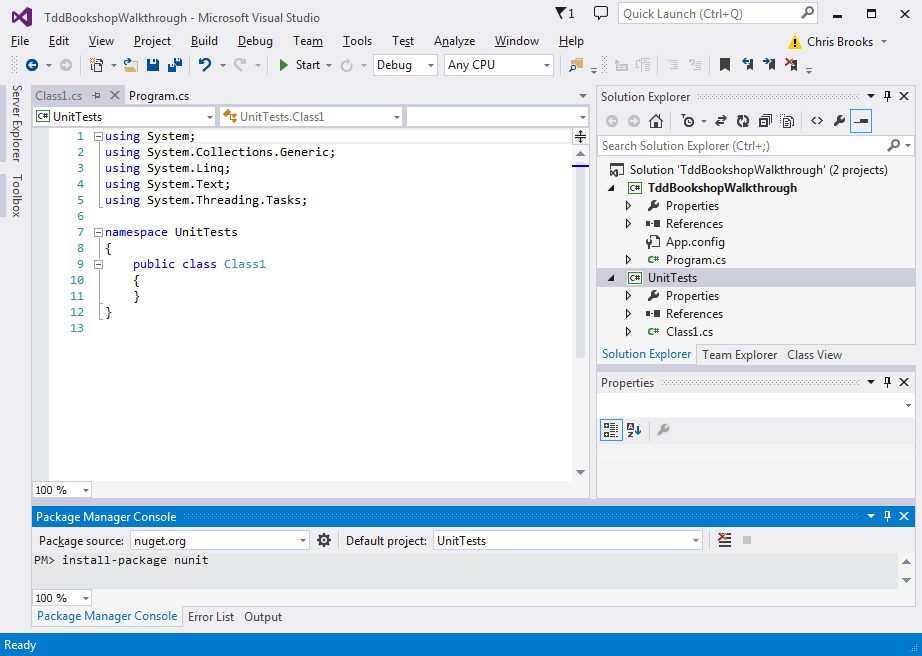
Call the class library UnitTests



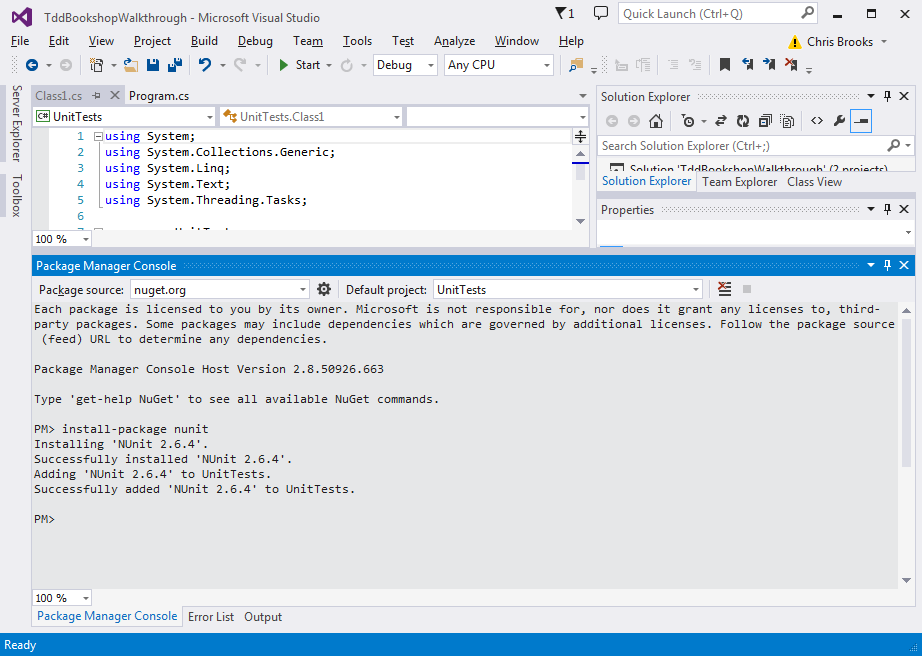
We now need to add *NUnit* to our test project (because it is our test project that will need to use NUnit to manage the tests).

Select Tools -> NuGet Package Manager -> Package Manager Console

When the package manager console appears, ensure that UnitTests is selected in Default Project

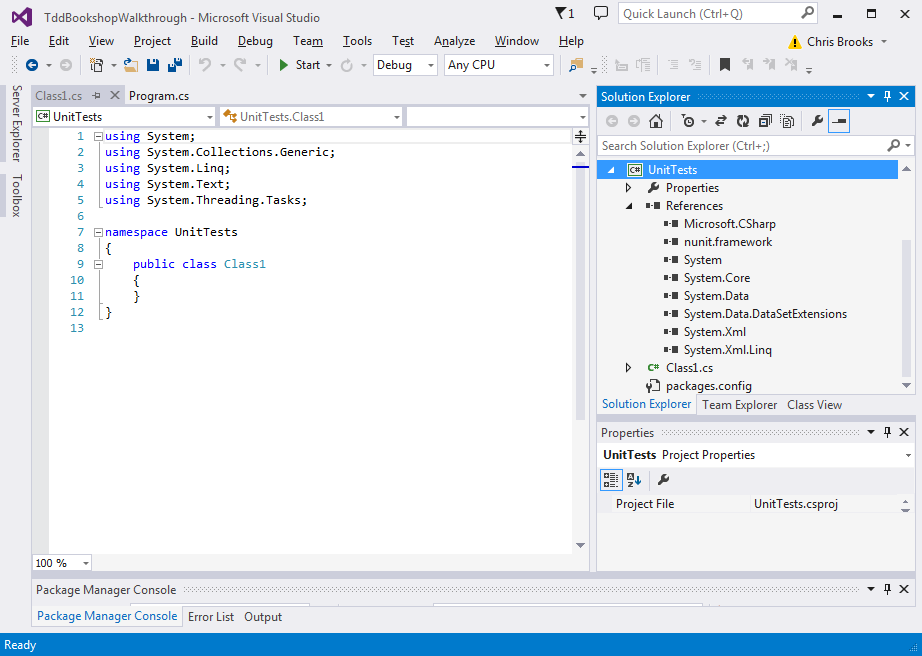
At the prompt, type install-package nunit 

and press *Enter*

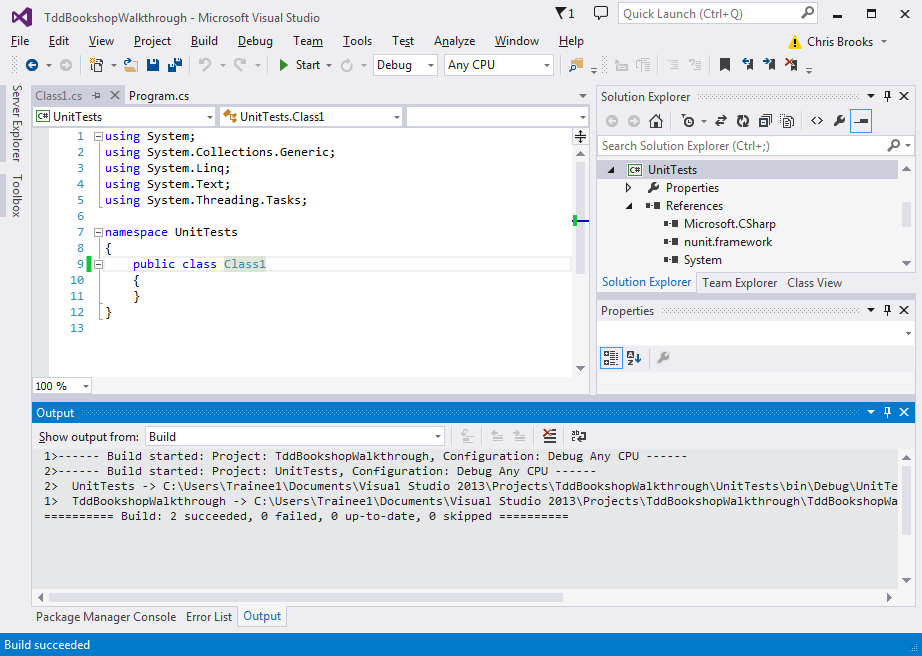


*NuGet* will download the *NUnit* framework assemblies.

Have a look at the *References* section of the *UnitTests* project to see the *nunit.framework* reference.

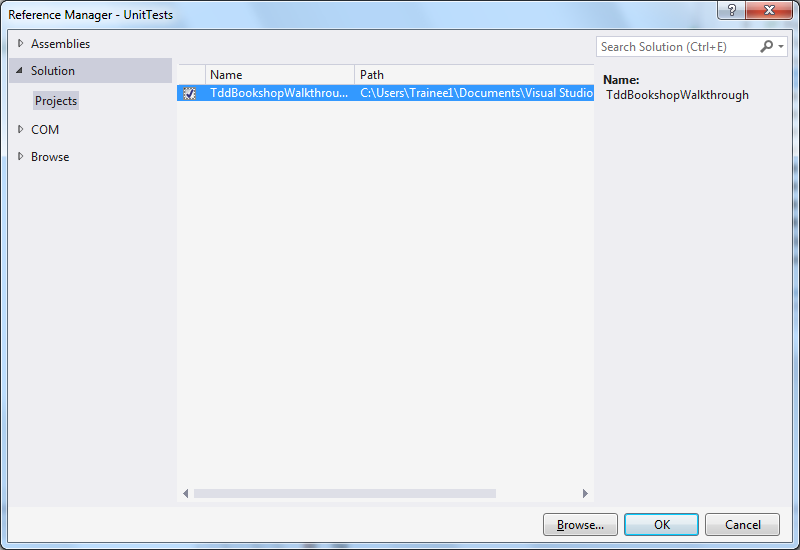


Build the solution, using either Ctrl-Shift-B or Build -> Build Solution



Right-click on the *References* section of *UnitTests*. Select Add References

Click Solution - > Projects and select *TddBookShopWalkthrough*

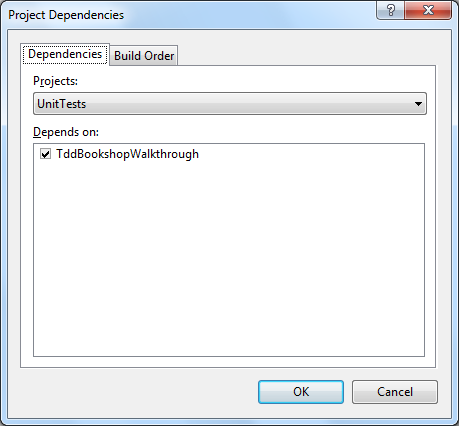


***Question: Why do we need to add a reference in this way?***

Right-click on the solution.

Select Project Build Order.

In Dependencies, select UnitTests. Make sure *TDDBookShopWalkthrough* is ticked.



Before we write any code, we want to make sure we are happy with what we want the code to do. For this, we turn to the Use Cases or User Requirements, which will indicate the application’s desired functionality.

Here are the User Requirements:

**The book shop application will consist of a catalogue of books. It should be possible to query the catalogue to obtain a list of all books it contains. Further, an administrative user should be able to add new book objects to the catalogue.**

**A book object should consist of a title, an author, a unique ISBN number, a price and the number of pages the book contains.**

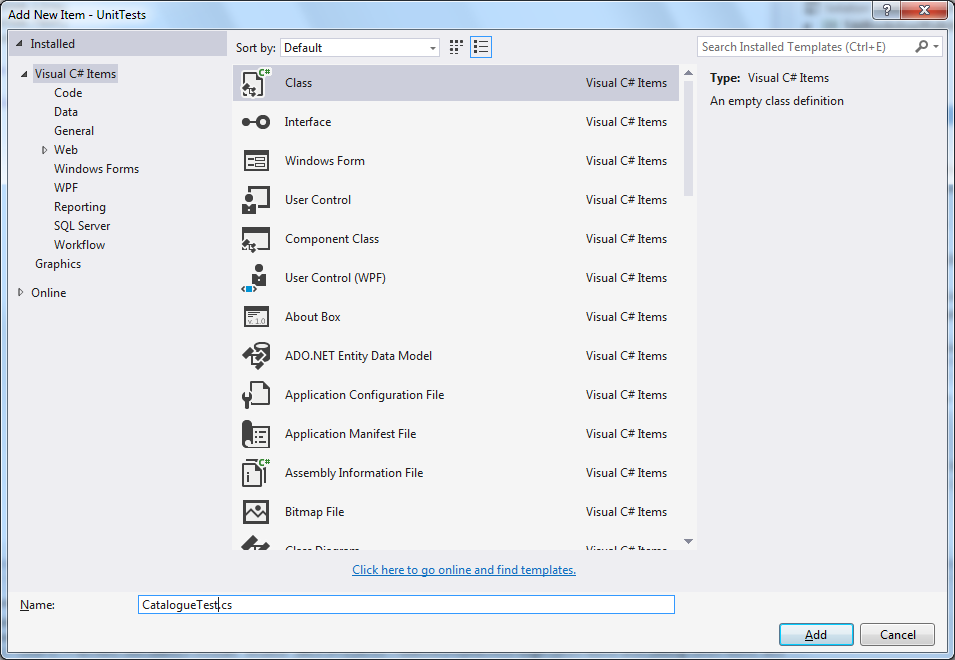
**The catalogue should be able to hold any number of books and it should be possible to remove books from the catalogue via their ISBN number.**

**A book’s price is subject to change, so functionality to update the price of a book in the catalogue should be included.**

The key object referenced here is the *Catalogue* object, so we will focus on that first.

## CatalogueTest

Right click on the *UnitTests* project, select Add -> Class. Name it *CatalogueTest*



This class will contain all of our tests for our *Catalogue* object. Note that we have not created the *Catalogue* class yet.

### [TestFixture]

In order for NUnit to recognise our tests, we use attributes defined by NUnit to ‘decorate’ the class and its methods.

Add

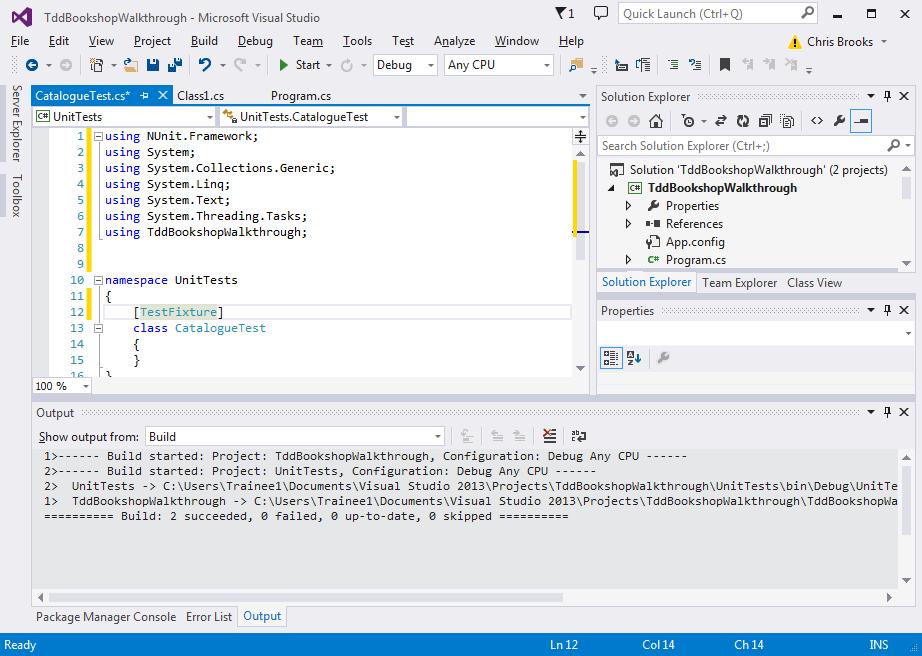
using TddBookShopWalkthrough;

using NUnit.Framework;

to the using statements.

Add [TestFixture] to the line above **class CatalogueTest**.

*Attributes* are typically used as flags in the C# language; this particular one is advertising to the NUnit framework that this class contains *Unit Tests*.



We use the attribute [Test] to tell NUnit that a method is a *Unit Test.*

### Test 1- Step 1: Write the Test

It can sometimes be difficult to make a *start* unit testing, so the first test is often the most difficult. Typically, we want to start with the *simplest possible behaviour*. We shall begin with requesting an empty list of books:

*When queried for the books it contains, an empty catalogue should return a list of length zero.*

Test names should be highly descriptive and usually consist of 3 sections:

1. The method that is going to be tested
2. The expected outcome
3. The initial conditions

[Test]

**public** **void** Test\_GetAllBooks\_ReturnsAListOfLengthZero\_WhenTheCatalogueHasHadNoBooksAdded(){

//test code

}

Once we have defined our test *name* we will build our test.

Test should be built whilst considering the three A’s:

1. **Arrange**
   * Set up our test and any initial conditions
2. **Act**
   * Call the method being tested, passing any required arguments
3. **Assert**
   * Look at the result of running the method being tested and make sure it does what we want. NUnit has a large number of pre-defined methods to help us perform these checks. Consult the API for more information:

[Test]

**public** **void** Test\_GetAllBooks\_ReturnsAListOfLengthZero\_WhenTheCatalogueHasHadNoBooksAdded(){

//Arrange

Catalogue catalogue = **new** Catalogue();

//Act

List<Book> booksInCatalogue = catalogue.GetAllBooks();

//Assert

*Assert.AreEqual(0, booksInCatalogue.Count);*

}

#### List

Within the test we are using an object of type List.

You can think of a List as an array that is able to both grow and shrink in size (something arrays cannot do) and which generally offers more options for data manipulation.

Here, the <Book> syntax simply means that this is a List of *Book* objects; we cannot put anything inside our List that is not a *Book* (or a subclass of *Book*). This gives us a degree of *Type Safety.*

A List is an example of a Collection in C#. We shall look at Collections in more detail later.

The angled brackets denote a ***Generic***. Generics will be covered in more detail later in the course.

### Test 1- Step 2: Make the Test Compile

We now need to write enough code to make our test compile. Remember we shall write *no* logic at this stage, simply empty classes and methods.

* Create a *Book* object in your TDDBookShopWalkthrough project. It should have no attributes at this stage:

**public** **class** Book {

}

* Create a *Catalogue* object in the same project:

**public** **class** Catalogue {

**public** List<Book> GetAllBooks(){

**return** **null**;

}

}

Catalogue has one method: GetAllBooks; it takes no arguments and returns a List of Book objects.

We just want our code to compile, so we return *null* (an empty piece of memory), which C# is happy to accept; we shall return something more useful later.

### Test 1- Step 3: Watch the Test Fail

Run the test.

Open the NUnit runner.

Select File -> New Project. Save a new project in your *UnitTests* folder.

Select Project -> Add Assembly. Browse to the *UnitTests.dll* file in the *bin* folder.

Your test should appear in the left hand pane. Click *Run*.

The NUnit runner should display a red bar, indicating the test has failed.

**Question: Why is it important to watch the test fail?**

### Test 1- Step 4: Do just enough to make the Test Pass

We will add the bare minimum code to force our test to pass. This is to ensure the logic is correct; we can come back and make the code better later.

**public** **class** Catalogue {

**public** List<Book> GetAllBooks(){

**return** **new** List<Book>();

}

}

### Test 1- Step 5: Refactor and Generalise

We have so little code at this point that there is very little refactoring we can do to either our production code or our test class. For now, we can move on to the next test.

# Next Steps

* Take a break to briefly reflect on what we have just covered
* Test 2: After one book has been added, the List returned should be of length 1
* Test 3: After adding one book, the catalogue should contain that same one book
* Test 4: After adding a Book we should be able to remove it using its ISBN number and get a List of length zero

# Collections

Within our TDD example, we used the List object on a number of occasions. Let’s take a more detailed look at Collections in C#.

## Collection Definition

In everyday English, a collection is a group of things. The same is true in C#.

A *thing* inside a Collection is referred to as an **Element**.

## Collection Hierarchy

There is an interface in C# called ICollection:

<https://msdn.microsoft.com/en-us/library/system.collections.icollection(v=vs.110).aspx>

ICollection sits at the top of the Collection hierarchy (hence the name) and has several child interfaces below it (as can be seen in the API). There is one in particular that we are interested in:

* IList

IList is an interface with no implementation of its own. We use implementing classes, some of which will be discussed below.

Consult the C# API for other concrete implementations of *IList*.

## List<T>

The main collection we will be dealing with is List<T>. When a list is defined it needs to be told what type it will be holding.

List<int> listOfNumbers = new List<int>();

In this example a new list is instantiated. The list will hold integers.

List<Book> listOfBooks;

In this example a new list is defined. The list will hold Book objects

**Question: What is the difference between instantiating and defining list?**

The list class has many useful methods we can use to manipulate the list and its contents. The one we will be using most often is the ‘Add’ method.

listOfNumbers.Add(2);

ListOfBooks.Add(new Book());

**Question: What would happen if we tried to add a new Book object to the listOfNumbers?**

## Collections

In addition to the Collection interface that sits at the top of the collection hierarchy, there is also a *class* called, confusingly, **System.Collections** (note the ‘s’ at the end).

Its API and further info can be found here:

<https://msdn.microsoft.com/en-us/library/ybcx56wz.aspx>

Collections consists of a number of helper methods that allow us to manipulate the various collections spoken about above, such as providing support for sorting, shuffling, reversing, copying and querying.

Take a look at the API and keep it in mind if you come across a situation where you need to manipulate a collection in some way. There may well be a built‑in way of achieving your goal.