C# Jumpstart training

Version 1.0

Contents

Training outline

The goal of the training is to learn a developer with experience in a different programming language and basic knowledge of Object Oriented Programming (OOP) the fundamentals of the C# language. The scope of this training should give you a head start for developing a C#-based program semi-independently.

The training will consist out of self-study part that takes about 1 to 3 workdays, followed by a final test of what you have learned by implementing a 1-Player Battleship game as a console app. Implementing this game is likely to take between 2 to 4 workdays.

Prerequisites

- The PC used during the training has at least .NET Framework 4.6.1 installed;
- Has experience with OOP and/or followed the ALTEN OO-1 training, and is therefore familiar with concepts like objects, instances, inheritance, polymorphism, encapsulation, composition, delegation, abstraction, methods and fields/members/attributes/properties;
- Has prior programming experience in imperative languages, and is therefore familiar with concepts like variables, functions, control flow statements and arithmetic operations.

Topics covered by this training

This training will cover the following topics:

- How to implement the ‘traditional’ “Hello World” application using C# and the .NET Framework;
- Methods;
- Data types, like value- and reference-types and how to properly compare them;
- Enumerations (a.k.a. enums);
- Properties;
- Classes;
- Inheritance (abstract classes and interfaces) and applying OOP in C#;
- Generics (how to make use of generic classes. Implementing generic classes yourself is out of scope);
- Arrays and Collections;
- Exceptions;
- Garbage Collection, *IDisposable* and the *using* statement;
- Events;
- Multithreading basics;
- LINQ basics;
- NuGet and platforms;
- Unit testing basics.

Course materials

Before you set off learning the C# language, let's first cover a bit of the .NET Framework ecosphere. When developing a C# application, you'll very likely going to make use of the .NET Framework. Then you start learning about the C# language and it's features from an online resource. It's recommended that you 'type along' while reading through the resources, as this will help you retain the knowledge better than just reading pieces of text. Then you'll proceed with some final information covered by this manual, about the topics not covered by the online resource. And the conclude the training with layout the details on implementing the 1-Player Battleship game and you putting your new skills to the test!

C# and the .NET Framework

The .NET Framework has been developed by Microsoft and consists out of various components and libraries. One major component you should be aware of, is the Common Language Runtime (CLR). This is an application virtual machine (similar to the Java virtual machine in concept), providing features like memory management, exception handling and security. It enables to develop a C# application that is practically OS-agnostic and hardware-agnostic (although you could still introduce those dependencies, such as through calling into OS-specific or platform-specific library).

When you compile a C# application, the compiler will not generate machine-code instructions. Instead it will generate Common Intermediate Language (CIL) instructions. This product is known as an *assembly*, and other manifests itself as either a *dynamic-link library* or an *executable*. When you run an assembly, the CIL instructions will be compiled into machine-code at run-time. This process is typically referred to a 'Just-In-Time' (JIT) compilation.

This means that code optimization also happens at run-time and that there is a (typically very minor) performance overhead when executing the code. Typically, this detail won't affect your day-to-day job when developing C# applications, but it's good to keep this detail in mind when developing a performance critical application.

Learning C# - Self-study from RB Whitaker's Wiki

RB Whitaker has made an excellent online resource for learning C# available online. You can start out with chapter 2 "Installing Visual Studio" if your PC's doesn't have Visual Studio installed yet. Otherwise, go ahead and start with chapter 3 "Hello World: Your First C# Program".

While the examples and screenshots are from Visual Studio 2013, the examples translate without a problem to Visual Studio 2017. So, go ahead and type along with the code examples presented, as it will greatly improve the retention of the knowledge you gain! Proceed to reading up to and including chapter 32 "Extension Methods". Once you've finished that chapter, continue with the remainder of the chapters from this manual.

Learning C# - Self-study remainder

All caught up on what RB Whitaker has to say about C#? Good, then let's cover the final pieces of knowledge here before you proceed onto implementing the final exercise of this training.

Chapter 33 – Garbage Collection, IDisposable and the using statement

As mentioned in Chapter C# and the .NET Framework, a program built using C# targeting the .NET Framework uses memory management. Any application will have a process running in the background, known as the *Garbage Collector*, that checks for pieces of allocated memory that are no longer being referenced. When it finds any of these sections of memory, it will free those and makes them available to reallocation.

The main advantage of this is that it removed a cognitive burden from the developer and therefore reducing risks that are present in languages that require explicit manual memory management, such as risks like dangling pointer or freeing a region of memory too many times. Off course this doesn't come for free: the garbage collection process itself requires resources like memory and CPU to work and it can cause possible stalls in application execution while the garbage collector reclaims memory.

Managed and unmanaged resources

Does the existence of the Garbage Collector mean you don't have to worry about 'cleaning up after yourself'? No, this is not the case. The Garbage Collector is only aware of *managed resources*: memory allocated in the heap.

Which begs the question: if managed resources are thing, what are *unmanaged resources*? These are things like operating system resources such as files and file handles, Windows (GUI), network connections or database connections. Typically, there is some class that wraps such a resource and as you might have guessed, that class itself is a managed resource and thus the Garbage Collector is aware then such a resource can be reclaimed. However, the Garbage Collector doesn't know how to reclaim or cleanup the unmanaged resources used by that class.

Cleaning up unmanaged resources – IDisposable and the using statement

To free up unmanaged resources, the interface *IDisposable* has been defined. The interface defines only 1 method:

```
public void Dispose();
```

When you create and use an object that implements *IDisposable*, you have the responsibility of ensuring that this Dispose method is called when you are done using that object. The Garbage Collector will not do this for you! Luckily there is the *using* statement that does all of the heavy lifting for you. It guarantees that the Dispose method is called when the instruction pointer leaves the scope defined by the using statement, even if this happens because of an exception that has been thrown from inside its scope. In practice, it would look like this:

```
using(var socket = new ClientWebSocket())
{
      // Magic happens here using the socket
}
// The instance in variable 'socket' is out of scope and Dispose has been
// called.
// Because there are no references to 'socket' any more, it's eligible for
// garbage collection.
```

The Dispose pattern

An implementation pattern has emerged and promoted by Microsoft as the recommended way of implementing the *IDisposable* interface: The Dispose pattern. A detailed article on this subject is available online at https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/dispose-pattern.

This article describes how to deal with:

- Owning other *IDisposable* instances;
- Inheritance;
- Owning unmanaged resources;
- Under what conditions you should or shouldn't implement a finalizer.

Go ahead and read the article and make sure you've understood it thoroughly. Ask your trainer for clarifications if there are any parts you didn't fully grasp.

Chapter 34 – LINQ

The Language Integrated Query (LINQ) is the name for the technologies used to integrate query capabilities into the C# language. It offers a declarative query expression similar to what you might have seen before when working with database queries like when using SQL, allowing for filtering, ordering and grouping operations on various datasources. Out of the box, the following datasources are supported:

- SQL Databases;
- ADO.NET datasets;
- XML documents/streams;
- .NET collections.

As part of this training, we'll be focusing on the .NET collection usecases.

IEnumerable and IEnumerable<T>

To use LINQ queries, the data source should implement either the *IEnumerable* or the generic *IEnumerable<T>* interface. Both interfaces define a method named *GetEnumerator()*, which should return an iterator for that data source. An iterator is an object responsible for making data elements from some container available in some sequence. The order in which those data elements are made available is typically not guaranteed, meaning that consecutive iterating over all elements might result in different sequences every time.

All .NET collections already implement these interfaces and some non-collection containers might as well. We're not going into how to implement an iterator yourself, as needing to do this in practice rarely happens. You typically can reuse the iterators made available by the .NET collections.

One important property of iterators when one is created, it won't actually bind to the data source yet until it actually gets used to generate data elements. I.e., you can construct an iterator at time T0 and at some later time T1 start to get elements from the data source. Any changes to the data source between T0 and T1 will therefore be available when the actual enumeration of elements happens.

Using LINQ extension methods

All LINQ methods are defined as extension methods for either *IEnumerable* and generic *IEnumerable<T>*. The following sections will describe some of the most commonly used methods, but there are even more out there. To make use of LINQ, you'll first need to use the *System.Linq* namespace.

Filtering data – Where

Let's say we have a collection with names:

```
var names = new string[] { "Jack", "John", "Pete", "Barry" };
```

And let's say we want only the names that start with a J and assume that we don't know the actual contents of the 'names' collection. By now, you'd probably implement either a *foreach* or *for-loop* and in each iteration check the name if it starts with a J and add it to an end result collection if it does.

Probably something like this:

```
var namesStartingWithJ = new List<string>();
foreach(string name in names)
{
    if (name.StartsWith("J"))
    {
        namesStartingWithJ.Add(name);
    }
}
```

How would you achieve the same result using LINQ?

```
var namesStartingWithJUsingLinq = names.Where(name => name.StartsWith("J"))
    .ToList();
```

So, what's happening here exactly? Remember chapter 26 Delegates? The extension method *Where* takes 1 argument of type *Func<string, bool>*, which is a generic delegate defined as:

```
public delegate TResult Func<in T,out TResult>(T arg)
```

The *Where* method binds the generic parameter *TResult* to bool always, and binds *T* based on its own generic parameter *T*. Because 'names' is a generic collection of type string, Where's generic parameter *T* binds to string.

Okey, so what's the story about this `name => name.StartsWith("J")` ? This is a so-called *lambda expression*, a dynamically defined method. The terms left the => define the method arguments and the types are inferred from the context they are used in. In this case, the context being a *Func<string, bool>* so the 'name' argument is of type string.
The terms right of the => define the method body (the expression to be really exact) and it should return *bool* (because this is what *Func<string, bool>* demands). The method *StartWith("J")* returns a *bool* when the argument 'name' starts with a capital J.

And what does the *Where* method do? It will take the *IEnumerable<string>* 'names' and outputs a new *IEnumerable<string>* that only contains elements where the *Func<string,bool>* has returned *true* when executing that delegate on that element.

Finally, there is the *ToList()* call. Remember one of the key properties of iterators only binding to the data when it actually needs to iterate over the result? To list will iterate of the full result and create an *List<T>* instance with the elements in the final resulting *IEnumerable<string>*.

Go ahead and run this piece of code:

```
var names = new List<string> { "Jack", "John", "Pete", "Barry" };
var namesStartingWithJ = names.Where(name => name.StartsWith("J"));
foreach(string name in namesStartingWithJ)
{
    System.Console.WriteLine(name);
}
names.Clear();
System.Console.WriteLine();
System.Console.WriteLine("Reusing the query after clearing 'names':");
foreach (string name in namesStartingWithJ)
{
    System.Console.WriteLine(name);
}
```

Now use *ToList()* at the end of the *Where* method and run it again. See the difference?

Projecting data – Select

What if you want to transform a dataset into a different format? Maybe you have a collection of names and you need to create a collection of Person, where Person is:

```
public class Person
{
    public string Name { get; set; }
}
```

This is where the *Select* method can help out:

```
var names = new string[] { "Jack", "John", "Pete", "Barry" };
Person[] people = names.Select(name =>
{
    var person = new Person();
    person.Name = name;
    return person;
}).ToArray();
```

Select takes 1 argument and you might have guessed it, it's indeed a delegate: *Func<string, Person>*. The binding to string is the same as explained for *Where*. The binding to Person is inferred from both the return type (Person[] demands *TResult* to be of type Person), but's also inferred from the delegate that produced an object of type Person.
This example also demonstrates a *multiline expression*, but you can off course reduce it to a single line expression if you want:

```
var people = names.Select(name => new Person { Name = name }).ToArray();
```

Notice the use of 'var' and this code still compiling? The compiler still can infer that it's returning elements of type Person because the expression returns Person instances. What do you think if you'd cast the new Person to object?

An alternative of *ToList()* is the *ToArray()* extension method, which creates an array instead of a *List<Person>*.

Projecting sequences of data – SelectMany

Now you know how to project a sequence, but what if you need to project a sequence of sequences? Let's define the following:

```
public class Child
{
    public string Name { get; set; }
}

public class Parent
{
    public string Name { get; set; }
    public ICollection<Child> Children { get; } = new List<Child>();
}
```

Now we have a collection of parents, and we need a collection of children. Let's use *SelectMany*:

```
var parents = new[]
{
    new Parent
    {
        Name = "Margret",
        Children =
        {
            new Child{Name = "Maddy"},
            new Child{Name = "Madison"},
            new Child{Name = "Mark"}
        }
    },
    new Parent
    {
        Name = "Harry",
        Children =
        {
            new Child{Name = "Hank"},
            new Child{Name = "Hagrid"}
        }
    }
};
IEnumerable<Child> allChildren = parents.SelectMany(parent =>
    parent.Children);
```

SelectMany requires an argument of type *Func<TSource, IEnumerable<TResult>>*, and we provide this using a lambda expression that returns the collection of children. *SelectMany* will process all those elements into a single sequence.

But what if we wanted an Array of the names of the children? Can you think of a way to achieve this using LINQ?
Hint1: Select and *ToArray()* are useful here.
Hint2: There are actually two ways of solving this. Can you find both of them?

Ordering data – OrderBy and OrderByDescending, ThenBy and ThenByDescending

We've talked about that the iterator produced by an *IEnumerable* or its generic variant do not guarantee anything about the order of elements in the sequence. So, what if you need particular ordering?

You can use *OrderBy* (or *OrderByDescending*) to sort the result of a LINQ query. You can sort using the default comparison logic for that type (for types that implement *IComparable* or the generic *IComparable<T>* interfaces), but there is also an overload available where you can pass an implementation of the *IComparer* interface (or its generic alternative). This way you can implement whatever custom ordering logic you'd like.
If you need secondary (or tertiary etc) ordering, you can follow up with calls to *ThenBy* and *ThenByDescending*.

To sort the collection of children based on the length of their name and then by first letter of their name in descending order:

```
allChildren.OrderBy(child => child.Name.Length)
           .ThenByDescending(child => child.Name[0]);
```

Experiment with sorting some more. Define some more parents with various children. Then sort the parents by the number of children they have in descending order, followed by the last character of their name.

Finding data – First(OrDefault), Last(OrDefault) and Single(OrDefault)

Sometimes you are looking for (the existence of) some object, like an instance with a particular value for a property. This is where the following extension methods can be useful:

- First();
- FirstOrDefault();
- Last();
- LastOrDefault();
- Single();
- SingleOrDefault().

Each of these methods also have an overload that takes 1 argument of *Func<TResult, bool>*. Looks familiar? It should, because it's the same kind of delegate that the *Where* method needs, and that is because those overloads actually combine a *Where* with a the parameterless overload.
Another way of looking at it, is that the parameterless methods actually use a lambda expression of x => true, meaning they many and all elements in the sequence.

The names of *First* and *Last* do as the name suggest: they return the first or last element in the sequence. So getting the first or last character of a name, could also be written like `child.Name.First()` and `parent.Name.Last()`.

Then there is *Single*, which turns the matching element and verifies that there is exactly 1 occurrence of elements that match. If this requirement isn't met, it will throw an *InvalidOperationException* instead.

But what actually happens when executing any of these methods on an empty collection or a sequence that doesn't match the given expression? Then these methods will also throw an *InvalidOperationException*.

Therefore, there are also the '…*OrDefault'* variants, which do not throw an *InvalidOperationException* when the collections are empty or not matches can be found. Note that *SingleOrDefault* will still throw an *InvalidOperationException* when multiple matches are found.

Subsections of data – Skip and Take

Sometimes you only need to work with subsections of the data. Maybe you know you can skip the first N elements and then needs to process only the next 2 elements. This is where the *Skip* and *Take* methods come in. Both extension methods take an *int* as argument, which represents the number of elements to skip or keep. So, if we're only interested in the 3rd and 4th child after having sorted them, we can do this with the result:

```
var theInterestingChildren = sortedChildren.Skip(2).Take(2);
```

Experiment a little bit with these 2 methods. What happens when you do Skip(2).Skip(2)? What happens when you Take(2).Take(2)? What happen when you alternate Skip(1).Take(1).Skip(1).Take(1)? Do you understand why you are getting these results? If you are having trouble understanding what's going on, make separate variables for each call and debug the application.

Using LINQ query language

Besides LINQ introducing various extension methods, it also introduces an alternative notation. While this alternative notation doesn't support all LINQ extension methods, it does allow for some other logic that doesn't express itself as easily using only extension methods. The alternative notation looks very similar to many existing query languages, like SQL. An example:

```
var names = new string[] { "Jack", "John", "Pete", "Barry" };
var result = names.Where(n => n.StartsWith("J"));
var equalResult = from n in names
                  where n.StartsWith("J")
                  select n;
```

'result' and 'equalResult' produce the same sequences with the same behavioral properties.

So where does this query language notation really shine? Let's show this by an example and why don't you try to implement it without using the query language. There are two arrays of numbers:

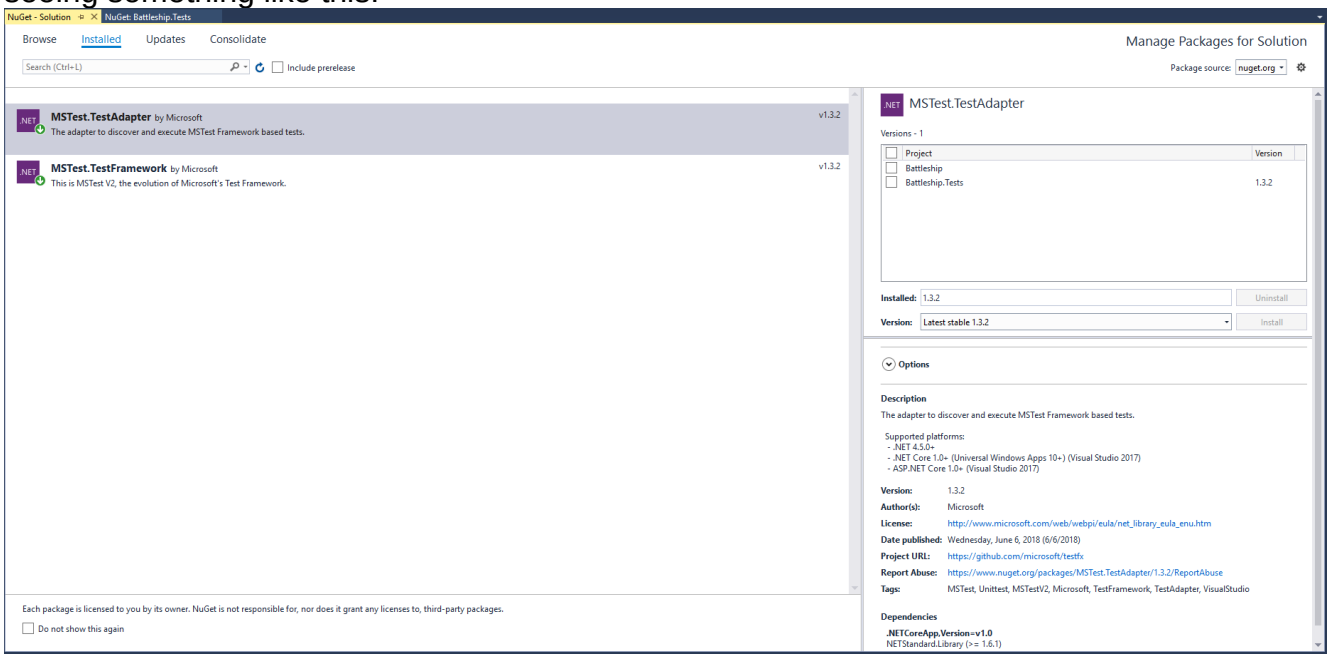
```
int[] firstArray = { 1, 2, 3, 4 };
int[] secondArray = { 5, 6, 7, 8 };
```
Now, produce the resulting collection of all single numbers from ‘firstArray’ multiplied by any single number from ‘secondArray’ that is greater than 15. So, the result of 1*5 should not be in the end result, but the value of 4*8 should. Give it a try being continuing on.

You probably needed 2 loops to calculate the products of all the permutations. Perhaps you used a *Where* call on the intermediate result, or maybe you just used an if-statement? Wonder how this can be expressed using the query language?
```
IEnumerable<int> result =
    from firstElement in firstArray
    from secondElement in secondArray
    let product = firstElement * secondElement
    where product > 15
    select product;
```
Elegant isn’t it? Especially when working with multiple different collections can the query notation be easier to read and therefore easier to maintain.

Chapter 35 – NuGet

One of the appealing aspects in software development, is being able to reuse some component or library that someone else already developed and made available to the general public. One source installed by default in Visual Studio is the NuGet package manager.

You can access the NuGet Package manager for your solution or project by right clicking on the corresponding node in the Solution Explorer panel. Go ahead and open the solution for the Battleship exercise and then the NuGet package manager for the solution. You should be seeing something like this:



Tour of the NuGet package manager GUI

In the top left you find the tab named “Browse”, which allows you to search for packages using keywords. Below you find a list of packages, each with a title and short description. It’s possible to get access to the cutting edge latest (and possibly slightly unstable) prereleases by checking the checkbox. To the top right you can see the selected ‘package source’, which is likely nuget.org. It’s possible to add additional sources, like a company hosted feed or even a feed based on your current machine. The “Microsoft Visual Studio Offline Package” is a preinstalled example of this, with NuGet packages that were shipped along with Visual Studio and made available in case you develop in a disconnected environment.

In order to install a package, you click on the respective package and then in the right panel which projects should have this package installed and referenced. You have to option of picking a specific version and this is important when doing configuration management and preventing ‘dll conflicts’. It’s typically advised to use only 1 particular version across the whole solution.
Below is an options panel that can be opened, which hides some advanced settings that we’re not going to cover within this training. Just be aware that it exists. Below that, you find the full description of the package, followed by some meta-data. One of the important ones among them, is the license under which the package is made available.

When downloading and using a package in your application, you should first confirm that the licensing conditions are compatible for the product you are working with. For example, packages released under the GNU Public License can only be used within a product that itself is also released under the GNU Public License, whereas packages released under the Microsoft .NET Library license can be installed without problem. When in doubt if a particular License can pose a problem, read the license thoroughly yourself, ask colleagues or the legal department of the client you’re working for. Another resource you could consult is https://tldrlegal.com/.
Using a package without meeting the licensing requirements typically results in copyright infringement and risks associated penalties for such violations.

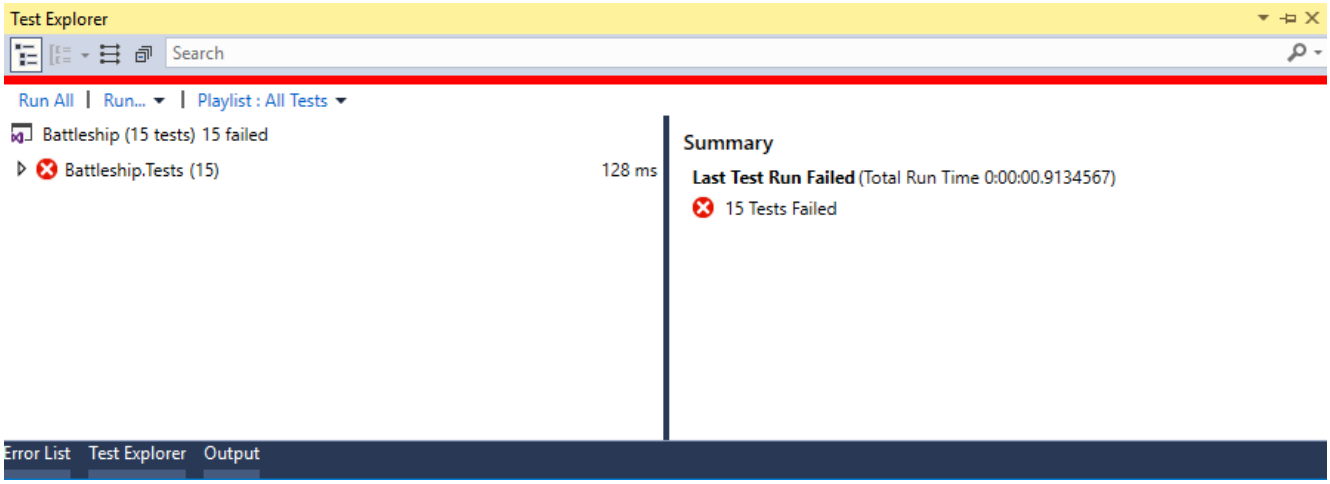
The “Installed” tab shows all currently installed packages and their versions.

The “Updates” tab shows all packages for which there is an update available. These could be new features, general improvement and bugfixes and security patches. Off course, an update also risks compatibility issues.

Finally, there is the “Consolidate” tab, and this one is only available when managing packages at solution scope. This panel helps you identify if you have any packages in your solutions for which multiple different versions are being used. It facilitates an easy way to ensuring only 1 version within the solution of a particular package is being used.

Chapter 36 – Unit Testing

Visual Studio supports out of the box the execution of tests. The solution of the Battleship exercise contains various unit tests implemented using the *MS Test framework*. In order to run all the tests, click in the menu “Test > Run > All Tests”. This will trigger a compilation of all the projects, followed by the execution of unit tests. If there are no compilation errors, the Text Explorer window will open after all the unit tests have executed. It will look similar to this:



You can navigate the available unit tests by expanding the tree. This tree is populated based on the name of the project, namespace, class name and then individual test methods.

By clicking on a test, the panel to the right shows the details of that test:

- Name
- Location of the method
- The error message, if any
- The stack trace of the error, if any

By double clicking on a test, the corresponding code window will open showing that particular test.

Implementing tests using MS Test

Each class that defines tests, should be decorated with the *[TextClass]* attribute. This enables the test execution engine to determine which classes in a test project are relevant.

Each test method is decorated with a *[TestMethod]*, also to enable the test execution engine to differentiate methods that should be considered tests and those that are not (e.g., because you define a method to reduce duplication happening in the tests). The body of such a method typically contains:

- Setup code that initialized the environment and create a particular state for the relevant objected
- The code that is actually being tested (could be a single method or property, or a sequence of actions)
- Assertion code that verifies the expected state or end result of the tested code.

For example, take a look at the method named **GivenNoCommandlineArguments_WhenCallingLoadLevel_PrintsErrorMessageAndTerminatesGame**. The name of the method is just for semantic meaning to what is happening and follows one of naming conventions that exists for test specifications. You can see that the ‘game’ variable loads a level using an empty string array. The methods of the Assert class all test for particular conditions and throw an exception if those conditions aren’t met. You can see the following assertions:

- The property *HasClearedConsole* of the variable ‘console’ should be *false*.

- The method *GetAllTextInConsole()* of the variable ‘console’ should return a collection of text matching a single line of text.
- The property *IsGameFinished* of the variable ‘game’ should be *true*.

Then, there are a couple of attributes that you can use to define methods that should be called:

- Once before the test are executed. (Decorate it with *[ClassInitialize]*)
- Every time before any test is executed. (Decorate it with *[TestInitialize]*)
- Every time after any test has been executed. (Decorate it with *[TestCleanup]*)
- Once after the tests have been executed. (Decorate it with *[ClassCleanup]*)

These methods allow for reducing code duplication when a particular code pattern happens for every single test and that code is related to setting of the test environment or tearing it down again (like removing files that might be generated will the test executed).

You can see that currently the class *BattleshipTests* only defines a method for *TestInitialize*, where it creates the needed objects to perform the test.

Final Exercise: Implementing 1-Player Battleship

Now that you have learned about the C# syntax, OOP using C# and some handy libraries and tools that are available, it's time for you for one final exercise. And that is implementing a 1-Player Battleship game to be run from the command line.

Visual Studio Solution

We've prepared an Visual Studio solution for you to get started, which contains two projects. The project named "Battleship" exists to host the 'domain model', the classes that make the executable of the 1-Player Battleship game. The "Battleship.Tests" project contain functional requirements that you are going to meet when implementing the 1-Player Battleship game, expressed as tests.

In the "Battleship" project, you'll find the following code-files:

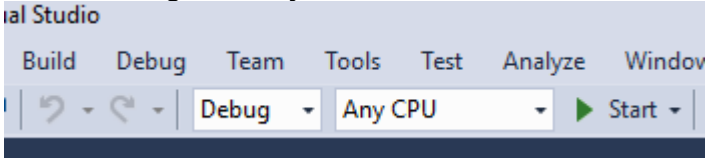
- *IConsole.cs*, which is the interface of the *System.Console* class that we want you to use as part of this exercise. *System.Console* is normally the class you'd use when interfacing with the command line terminal, but directly using that class also means it's impossible to write tests based on that interaction. As such, an interface is defined that you can use instead.
  We've also limited the available API to ensure the functional tests are relevant.
- *Console.cs*, which is the implementation of *IConsole* and directly routes all calls to *System.Console*.
- *BattleshipGame.cs*, which is the class that represents the entry point of the game. Currently, it's an empty skeleton and it's going to be your responsibility to 'fill in the blanks'.
- *Program.cs*, which defines the program entry point when it gets executed and the core game loop.

In the "Battleship.Tests" project, you'll find the following code-files:

- *ConsoleMock.cs*, which is an alternative implementation of the *IConsole* interface and exists to test interaction with the *IConsole* instance presented to the *BattleshipGame* instance.
- *BattleshipTests.cs*, which contains the functional unit tests for *BattleshipGame*.

Building the solution

You can build the solution by clicking in the menu "Build > Build Solution". A build will generate a folder named 'bin' in the project's directory (which is <repo root>\src\Battleship\ for the "Battleship" project), and in there will be two folders "Debug" and "Release". The active configuration you can find below the menu:



With the drop down, you can switch between debug and release builds. Based on this configuration, you'll find the binaries in the respective folder on your machine.

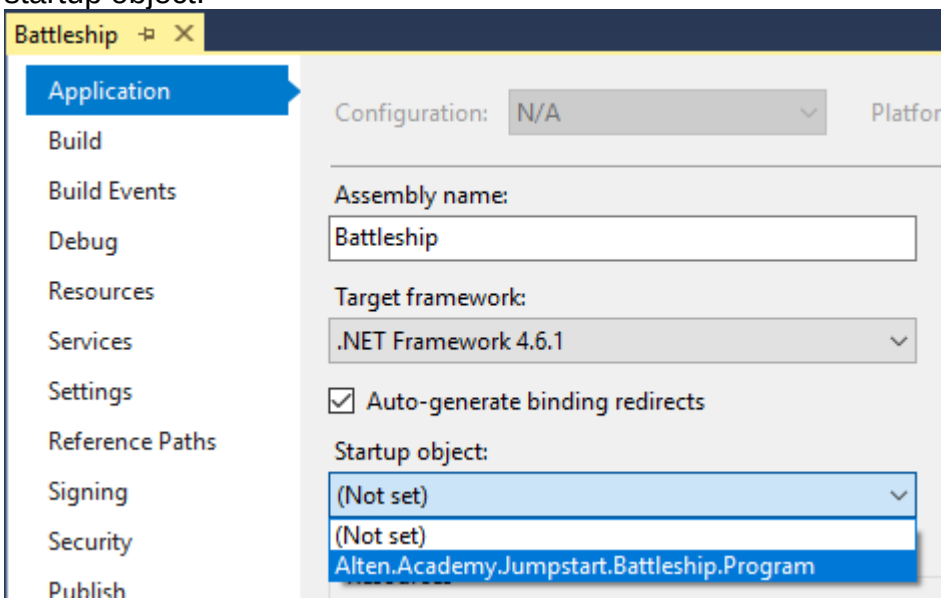
Running Battleship

There are two ways of running Battleship. Chose the method that you prefer: running it from Visual Studio (especially useful if you want to debug your application!) or from command line. Regardless of the chosen method you'll need to provide the path to a level file. There is a valid level file available in <repo root>\src\Battleship.Tests\testdata\validLevel.csv.

When running the application, you'll get a black command line terminal and nothing happens. You can close the terminal when you are done.

From Visual Studio

First configure Battleship to be startup program. Right click on Battleship project in the Solution Explorer of Visual Studio, and click "Properties". In the 'Application' tab, set the startup object:



Next, in the Debug tab, fill in the following command line arguments:
..\..\..\Battleship.Tests\testdata\validLevel.csv

Then, to run (and start debugging) click in the menu "Debug > Start Debugging".

From commandline

In Windows search, type "cmd" and chose the item "Command Prompt".
You can change the current working directory using the command 'cd <path>'. For the debug build, change the working directory to <repo root>\src\Battleship\bin\Debug.

Next, run the following command:
Battleship "..\..\..\Battleship.Tests\testdata\validLevel.csv"

Goal

Your goal is to implement the Battleship 1-Player game to the degree that it becomes playable from the console. You are going to parse a level file and use that at the initial game state. You are also going to implement the game state management and user input handling.

Rules of the game

The world consists out of a square grid of 9x9. Horizontal coordinates are indicated with [1,…,9] and vertical coordinates are indicated with [A,…,I]. The game is going to look like this:

In this grid, a couple of ships are hidden and you win by guessing the locations of these ships and sinking them. You sink a ship by shooting and hitting all the coordinates they take up. The game defines the following ships:

- Aircraft Carrier, which is a line of 5 squares long and 1 square wide
- Battleship, which is a 1x4 line.
- Submarine, which is a 1x3 line.
- Destroyer, which is also a 1x3 line.
- Patrol boat, which is a 1x2 line.
- Cruise ship, which is a 1x2 line.

Each ship is oriented either horizontally or vertically.

Every turn you get to guess a coordinate, such as A1 for the top-left coordinate. The game will tell you if you missed or when you hit a particular type of ship.

But there is a twist: if you sunk a cruise ship, you lose the game immediately!

Level file

When starting a game of Battleship, you must provide a file path as command line argument: a level file. The file format is the ‘comma separated values’ format, a plain text format readable using your favorite text editor or Excel.

The file defines 5 columns:

1. Name of the ship, which can be any string not containing a ‘,’ character.
2. Type of the ship, whose value must be in the set [Carrier, Battleship, Submarine, Destroyer, PatrolBoat, Cruise].
3. Horizontal coordinate, whose value must be in the set [1,…,9].

4. Vertical coordinate, whose value must be in the set [A,…,I].
5. Orientation of the ship, whose value must be in the set [Horizontal, Vertical].

Each line defines the location of single ship. As an example, take the line
SS Skyrake,Carrier,9,E,Vertical
This defines an Aircraft Carrier with the name “SS Skyrake” which takes up coordinates 9E, 9F, 9G, 9H and 9I.

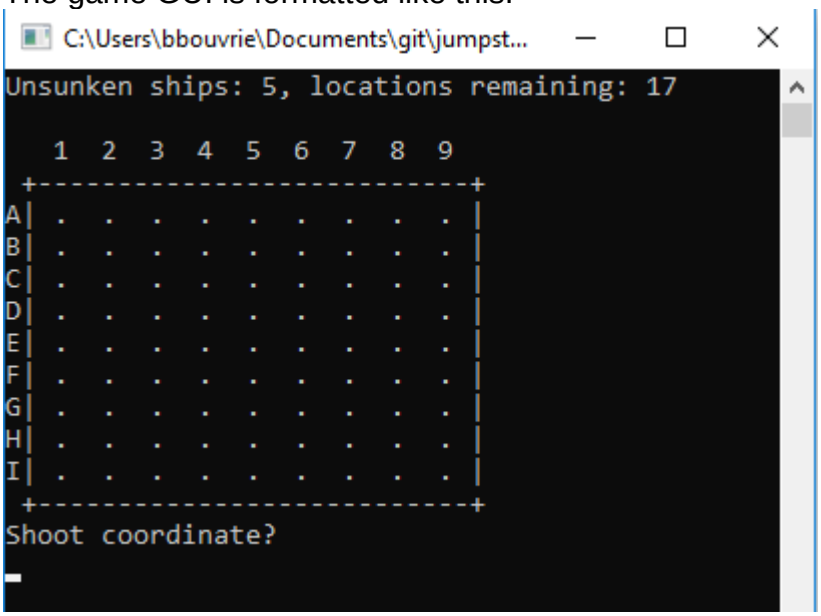
Another example:
SS Bombard,Battleship,5,F,Horizontal
This defines a Battleship with the name “SS Bombard” which takes up coordinates 5F, 6F, 7F and 8F.

The file can contain any number of lines and any combination of ships.

Functional Requirements

You are required to implement the following set of requirements in order to complete this exercise:

1. The game GUI is formatted like this:



   a. The first line keeps track of the current number of unsunken ships and the number of locations that still need to be guessed.
   b. The second line displays any messages.
   c. The gameboard is formatted in the next 12 lines.
   d. The question to guess which coordinate to shoot at
   e. The console input locations, where the player can fill in the coordinate to shoot at.
2. The program must allow only 1 command line argument. This file path should refer to a level file that exists and should be valid.
3. If an invalid level file is read, print a message explaining the error and terminate.
4. If a valid level file is read, clear the console and present the game GUI and start a player turn.

5. A valid level file is defined as:
    a. A file of any name and any extension (but might be easier to constrain yourself to the .csv extension).
    b. A file that exists on the file system.
    c. Is formatted as explained in section Level file.
    d. Does not contain any ship that goes out of bounds.
    e. Does not contain any ship that overlaps another.
6. When shooting at a coordinate that doesn't contain a part of a ship, this is considered a miss. This should result in a game message.
    a. The game message should state the coordinate shot at.
7. When shooting at a coordinate that contains a part of a ship, this is considered a hit. This should result in a game message.
    a. The game message should state the type of the ship.
    b. The game message should warn the user to not shoot at cruise ships if it hit one.
    c. The game message should state the coordinate shot at.
8. Hitting a ship should decrement the number of locations that should be shot at.
9. Coordinates in the game GUI should be marked as follows:
    a. Unshot: .
    b. Miss: ~
    c. Hit an Aircraft Carrier: A
    d. Hit a Battleship: B
    e. Hit a Submarine: S
    f. Hit a Destroyer: D
    g. Hit a Patrol Boat: P
    h. Hit a Cruise Ship: C
10. Sinking a ship should result in a game message.
    a. The game message should state the type of the ship.
    b. The game message should state the name of the ship
11. Sinking a Cruise Ship should result in a game message.
    a. The game message should state that the player lost the game due to sinking a Cruise Ship.
    b. This game message will completely replace the message that normally should be displayed for sinking a ship.
12. Sinking a Cruise Ship should terminate the game.
13. Sinking a ship that is not a Cruise Ship, should decrement in number of unsunken ships.
14. Sinking the last unsunken ship should result in a game message.
    a. The game message should state that the player won the game.
    b. This game message will completely replace the message that normally should be displayed for sinking a ship.
15. Sinking the last unsunken ship should should terminate the game.
16. Games that terminated in a win or a loss should not ask the player to shoot at a coordinate.
17. All tests should pass
    a. Most of the functional requirements stated above are covered by these tests.

Useful resources

- Use the tests to guide your progress.
- You can access the documentation of .NET Framework code by selecting the class/property/method and pressing F1. Alternatively, you can look it up online at https://docs.microsoft.com/en-us/dotnet/api/index?view=netframework-4.6.1
- The online C# reference guide you can find here: https://docs.microsoft.com/en-us/dotnet/csharp/index

Conclusion

You've learned how to use C# to create programs using OOP principles. You've been introduced to concepts like events, exceptions, managed and unmanaged resources and the risks they can pose to (long term) proper execution of your application. We've tipped how to implement multithreaded code and how to use LINQ. You've learned to make use of generics and experienced the added value that is presents due to code-reuse. We've covered NuGet and how to benefit from more reuse of existing libraries. And how to use existing unit tests.

But there are still topics within the C# and .NET ecosphere that we haven't covered yet. But those topics you probably do not need immediately, so you can learn about them along the way.
In case you are looking into topics to dive into next, here is a list of suggestions:

- Developing graphical user interfaces for desktop applications using Windows Presentation Foundation (WPF), Windows Forms (WinForms, which is being replaced by WPF)
- How to develop generic classes yourself
- Dive deeper into multi-threading and asynchronous programming, using 'async/await', Tasks and Threads
- Developing graphical user interfaces for multiple platforms using Universal Windows Platform (UWP, note that this is less feature then WPF at the time of writing)
- Data persistence using Entity Framework
- Developing Client/Server applications using Windows Communication Foundation (WCF)
- Develop server-side web applications using ASP.NET
- Develop cross-platform libraries using .NET Core