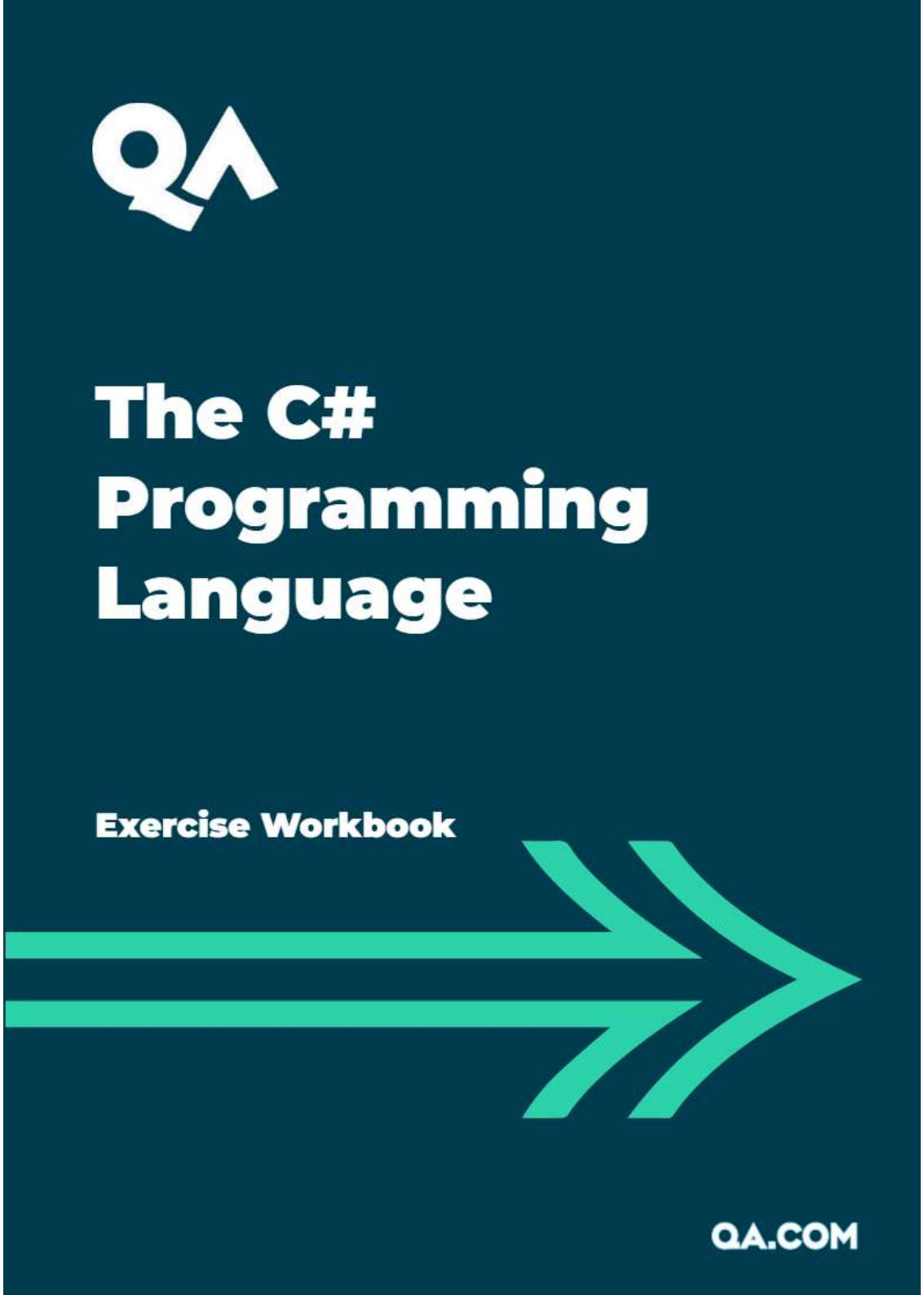
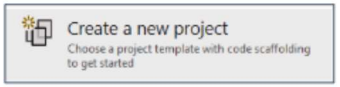
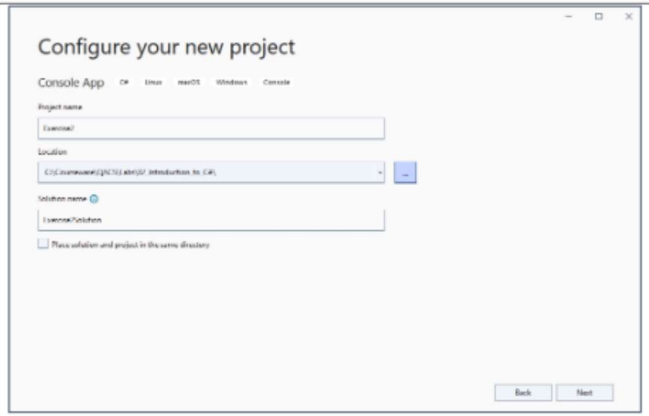
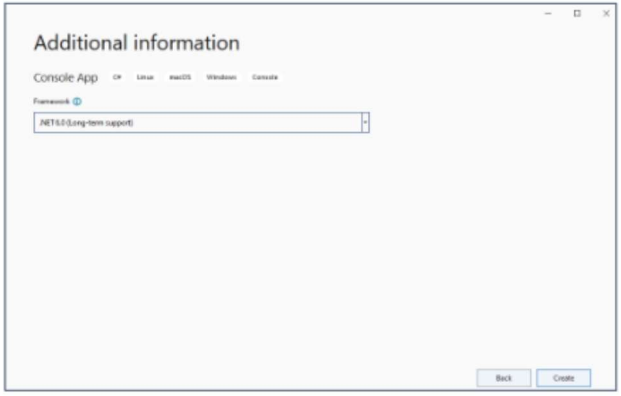
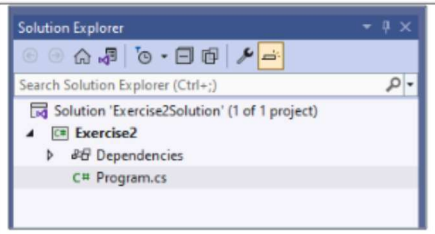

The C#
Programming
Language
Exercise Workbook
QA.COM

Introduction to C#

The objective of this exercise is to get you started using C# and the Visual Studio IDE and introduce you to simple debugging. You will also work with a test project and write some simple tests.

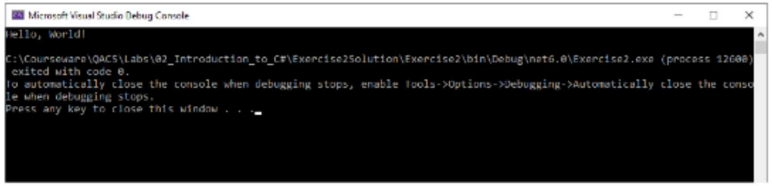
1	Start **Microsoft Visual Studio 2022**
2	Choose '**Create a new project'** 
3	In the Search box, type **Console** 
4	Select **Console App** and choose **Next** 
5	Name the *Project* **Exercise2** and the *Solution* **Exercise2Solution**. Save the files in **C:\Courseware\QACS\Labs\02_Introduction_to_C#**

	
6	Ensure **.NET 6.0 (Long-term support)** is selected in Additional information and click **Create** 
7	You will see a code editor window for a file called **Program.cs** containing the following code: 
8	You can see the Solution Explorer window with **Program.cs** being tracked as the active file.



9 From the toolbar, click the green arrow to **Start with Debugging**:

10 Observe the output of the program and press any key to close the console window.

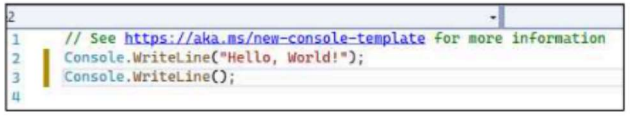


11 Add a line of code at line 3 as follows. Type:

CW

Notice it is a recognised code snippet for **Console.WriteLine**:

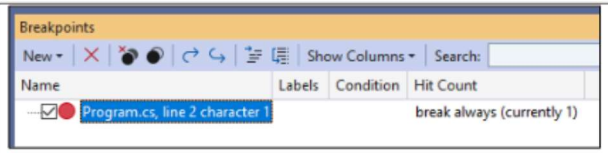
Tab twice to insert the code:



Page 4

Author: Pete Behague Subject: Highlight Date: 11/28/2022 10:38:12 AM

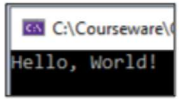
12	Add the text string **"Hello from C#"** as a parameter to the WriteLine method. `Console.WriteLine("Hello, World!");` `Console.WriteLine("Hello from C#");`
13	Run the program again, this time using the keyboard shortcut **F5** Microsoft Visual Studio Debug Console Hello, World! Hello from C# Press any key to quit the program.
14	You will now do some very simple debugging. Click in the margin well to the left of **line 2** to set a breakpoint: Program.cs Exercise2 `// See https://aka.ms/new-console-template` `Console.WriteLine("Hello, World!");` `Console.WriteLine("Hello from C#");`
15	Debug the program with **F5** and notice how the application is now paused on your breakpoint: `// See https://aka.ms/new-console-templ` `Console.WriteLine("Hello, World!");` `Console.WriteLine("Hello from C#");` Observe the changes to the Visual Studio layout. Numerous debug windows are now open including Autos, Locals, Watch 1, Call Stack, and Breakpoints.
16	Open the **Breakpoints** window and observe the **Hit Count** value is set to **"break always (currently 1)"**



17 You will Step Into the next line of code which will run the line that is currently highlighted.

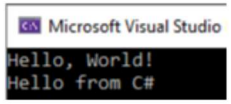
Press **F11**.

Look at the **Console** output window. You can see line 2 has run:



Press **F11** again.

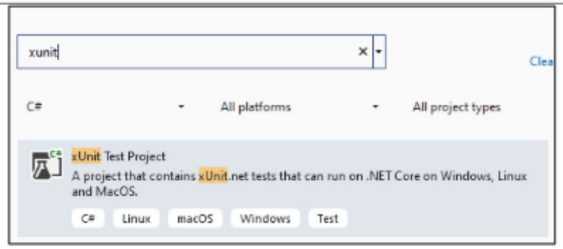
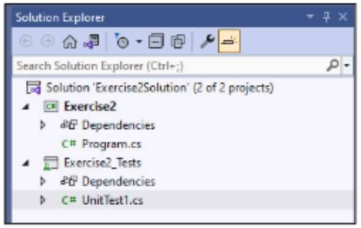
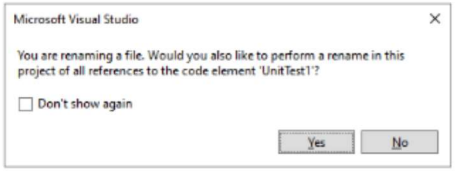
The program ends because the last line of code has been run successfully. You can see the complete result in the output console:



18 Some of the exercises on the course use tests to validate code behaviour so you will now add a test project to your solution.

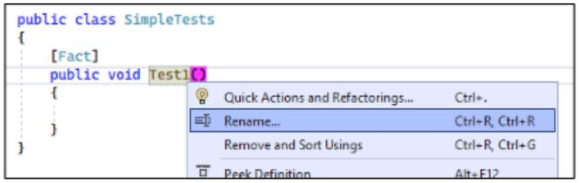
Right-click the **Exercise2Solution** and choose **Add -> New Project**

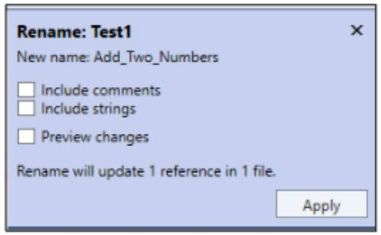
In the search box type **xunit**.

	 Select **xUnit Test Project** and click **Next**. Name the project **Exercise2_Tests** and click **Next**. Ensure .**NET 6.0 (Long-term support)** is selected and click **Create**.
19	Your **Solution Explorer** window now contains one solution with two projects: 
20	Rename **UnitTest1.cs** (by right-clicking on the file name) to **SimpleTests.cs** and select **YES** when the following prompt displays: 
21	Your **SimpleTests.cs** file contains the following starter code:

```
using Xunit;

namespace Exercise2_Tests
{
    public class SimpleTests
    {
        [Fact]
        public void Test1()
        {

        }
    }
}
```

22 Rename **Test1** to **Add_Two_Numbers**:

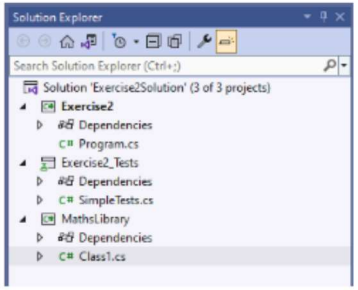


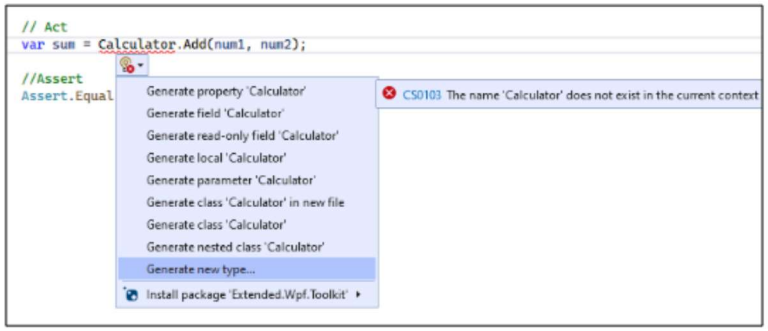
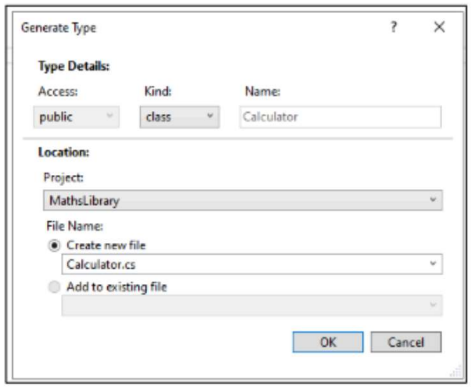


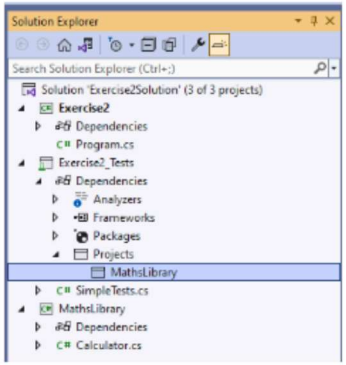
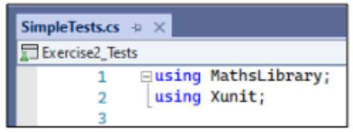
Click **Apply**.

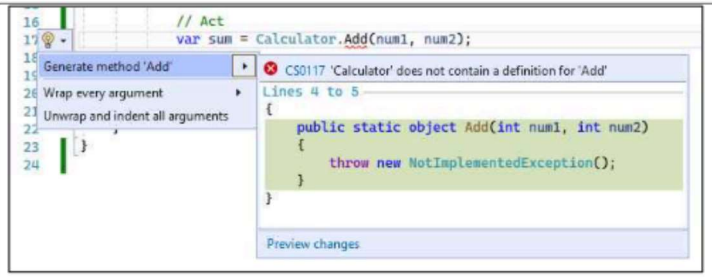
```
using Xunit;

namespace Exercise2_Tests
{
    public class SimpleTests
    {
        [Fact]
        public void Add_Two_Numbers()
        {

        }
    }
}
```

23 You are going to write some simple tests for a Calculator. This calculator is going to be created in a new project of type Class Library.

	Add a new **class library** project to the solution called **MathsLibrary**.  Solution Explorer should now look as follows: 
24	Delete the file **Class1.cs**.
25	In **SimpleTests.cs** add the following code to **Add_Two_Numbers**: `[Fact]` `public void Add_Two_Numbers()` `{` `  // Arrange` `  var num1 = 5;` `  var num2 = 2;` `  var expectedValue = 7;` `  // Act` `  var sum = Calculator.Add(num1, num2);` `  //Assert` `  Assert.Equal(expectedValue, sum);` `}`
	This test code uses the standard testing pattern called the triple A pattern: *Arrange, Act, Assert.* *Arrange* is for setting up items you need for the test. *Act* is for carrying out the action you are testing. *Assert* is for confirming the acted upon code behaves as expected.
26	The arrange phase creates three variables: **num1**, **num2**, and **expectedValue**.

	The act phase calls an **Add** method on a **Calculator**, passing in **num1** and **num2** as parameters and assigning the result to a variable called **sum**. The assert phase checks whether the **expectedValue** and the **sum** values are equal. The Calculator type does not exist so you will use Visual Studio to help you create it. Press **Ctrl+.** (Ctrl+dot) on **Calculator** to see the available options:  You want Calculator to be created in your **MathsLibrary** project rather than locally within the Test project so choose '**Generate new type...**'
27	In the dialog box, ensure a **public class** will be created and change the project to **MathsLibrary** and **Create new file**:  Click **OK**.

28	Visual Studio has created a new class (a kind of type) called **Calculator** in **MathsLibrary**: `namespace MathsLibrary { public class Calculator { } }` Visual Studio has also added a *reference* to the **MathsLibrary** project:  It has also imported the **MathsLibrary** *namespace* into your test project: 
29	The act phase of the test code now recognises the **Calculator** type but displays an error because **Calculator** does not contain a definition for **Add**: `// Act` `var sum = Calculator.Add(num1, num2);`
30	Use **Ctrl+dot** on **Add** to generate the method:



31 **Calculator.cs** now contains an **Add** method:

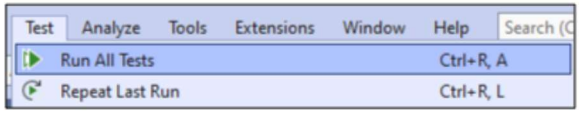
```
public class Calculator
{
    public static object Add(int num1, int num2)
    {
        throw new NotImplementedException();
    }
}
```

32 You want your **Add** method to return whole numbers so change the word **object** to **int**.

```
public static int Add(int num1, int num2)
{
    throw new NotImplementedException();
}
```

33 You will run the test and observe the outcome.

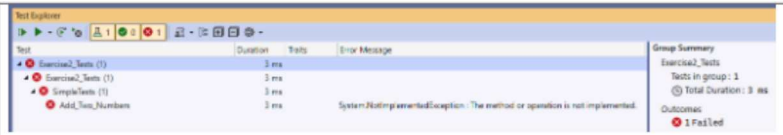
Test -> Run All Tests.



34 Ensure the Test Explorer window is visible:

Test -> Test Explorer.

Expand the Test until you see the **Add_Two_Numbers** failed test (it appears in red) alongside the error message: **The method operation is not implemented.**



35 You will edit the **Add** method code to ensure the method is implemented and confirm that the test passes.

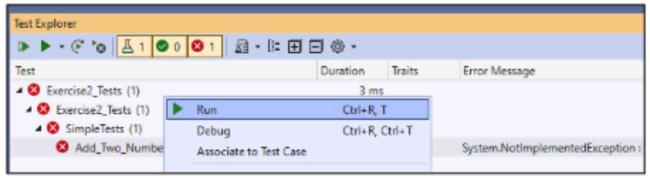
Delete the line of code: **throw new NotImplementedException;**

Replace the code with: **return 7;**

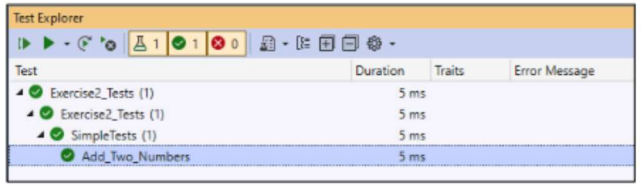
```
public static int Add(int num1, int num2)
{
    return 7;
}
```

This is hard-coding the expected value, which allows you to confirm the test is working correctly.

36 Right-click the failed test in **Test Explorer** and select **Run**.



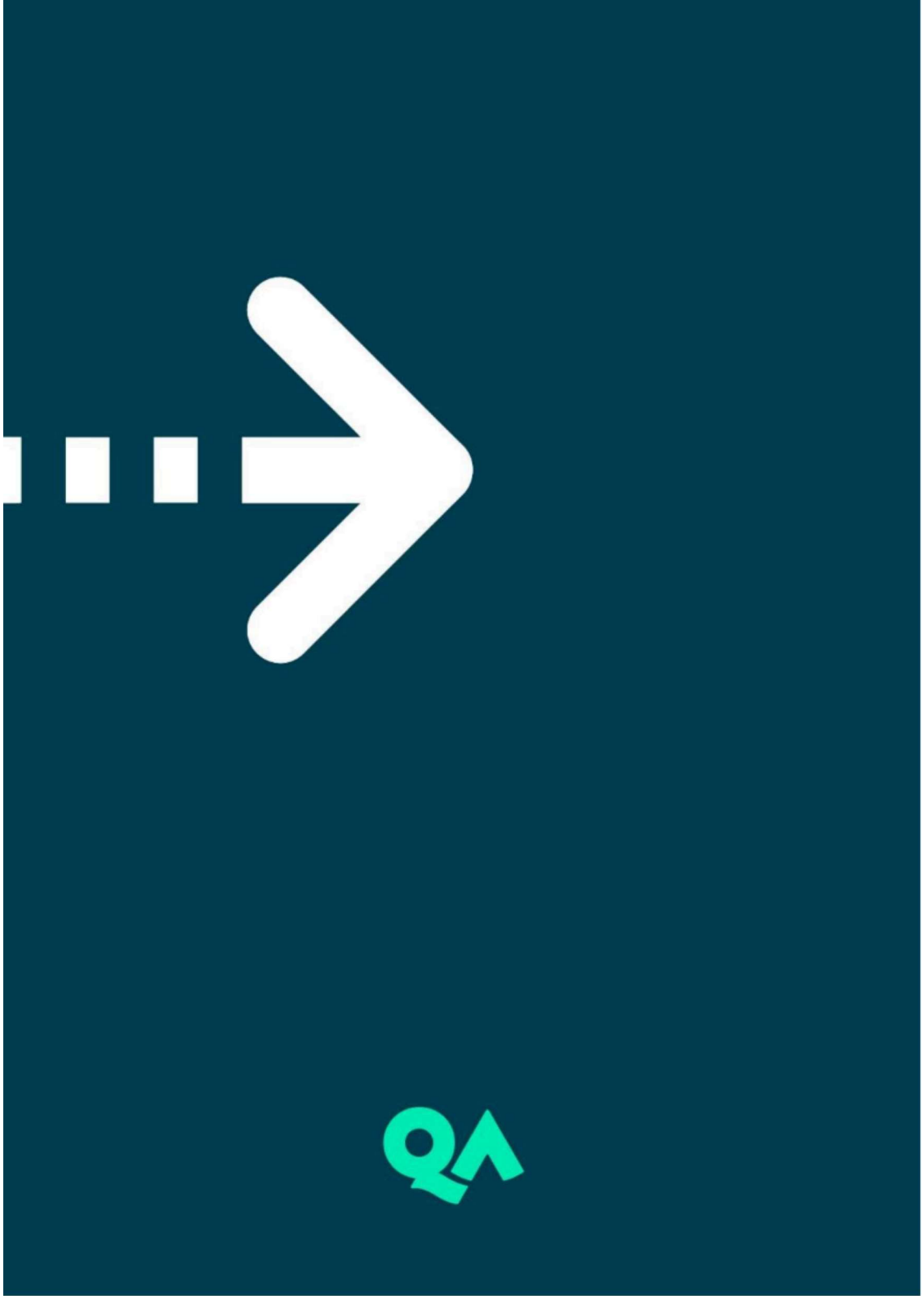
The test should now pass:



37 The final step is to refactor the code within the method to perform the calculation so that additional tests adding different integers will also pass.

Edit the **Add** method as follows:

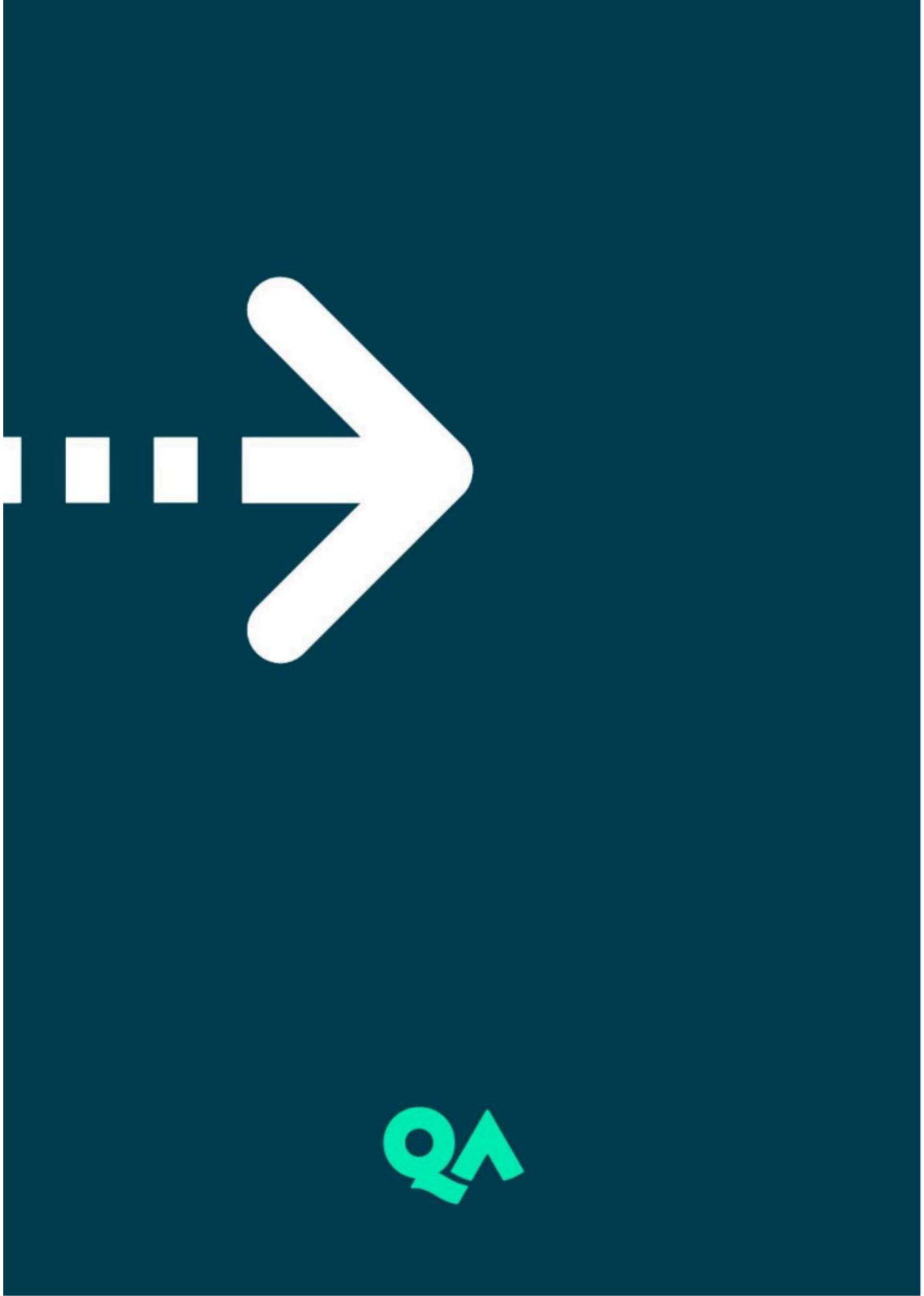
	`` public static int Add(int num1, int num2) { return num1 + num2; } ```
38	Re-run the test to ensure it continues to pass. The process that you just followed is called Test-Driven Development (TDD). It follows a three-stage approach referred to as *red-green-refactor,* whereby you write a test before implementing the code. You ensure the test fails. This is the red stage. This is to guard against any false positives. You then write enough implementation to get the test to pass. This is the green stage. You then refactor the code to improve the implementation, ensuring the tests still pass.
39	If you have time, write a test for a **Subtract** method, then use Visual Studio to help build the implementation. Use **Test Explorer** to run your tests.
40	Solutions are provided in the **End** folder for your reference.



Variables and Datatypes

The objective of this exercise is to consolidate your understanding of variables, datatypes, string formatting, and operators.

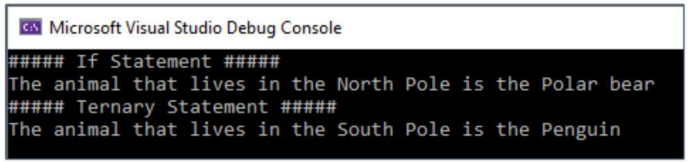
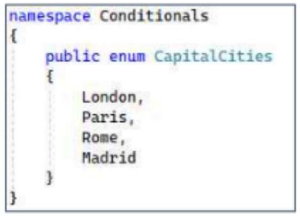
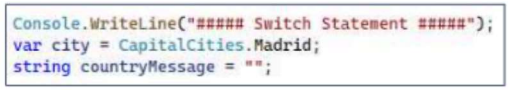
1	Open the **Begin Solution** for Exercise 3. C:\Courseware\QACS\Labs\03_Variables_and_Datatypes\Begin\Exercise3Solution\Exercise3.sln
2	All of the code is commented out. Go through all the numbered points (1 to 14), uncommenting each line as you go. In the case of question 10, there are a few lines to uncomment. Almost all of the lines of code have something wrong: either they won't compile or if they compile and run, they give an unexpected result. Point 14 involves setting a breakpoint to confirm the code behaves as expected. For all other code, identify what is wrong and fix it.
3	There is a Hints project if you need some tips.
4	A complete Corrected Solution is provided in the **End** folder for your reference.

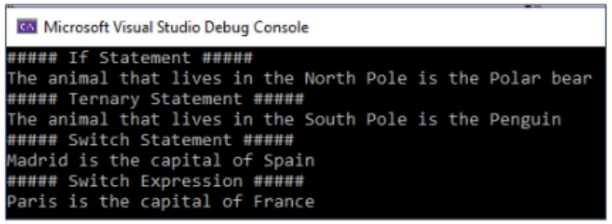


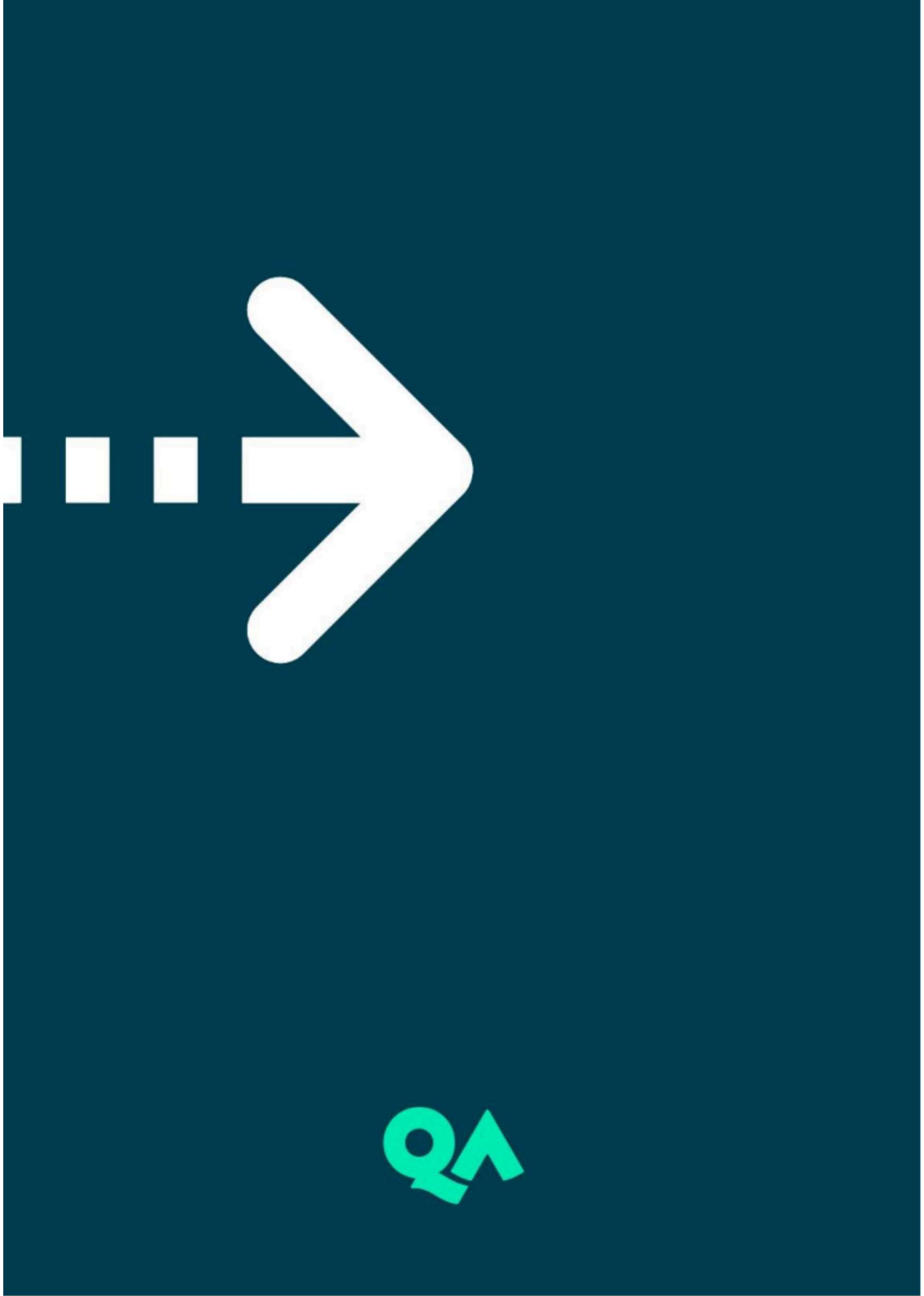
Conditionals

The objective of this exercise is to consolidate your understanding of conditional statements, including the if statement, the ternary statement, switch statements, and switch expressions.

1	Create a new **console** project called **Conditionals** in: **C:\Courseware\QACS\Labs\04_Conditionals\Begin**
2	Delete the contents of **Program.cs.**
3	Add an **enum** to the project in a file called **Pole.cs.** `namespace Conditionals` `{` `  public enum Pole` `  {` `  North,` `  South` `  }` `}`
4	In **Program.cs**, declare a variable called **pole** and assign it the value of **Pole.North.** Fix any issues using Visual Studio Quick Actions and Refactorings (**Ctrl+dot**). Create a second variable of type **string**, called **animal**. Write an **if statement** that tests whether the value of pole is equal to North and if true, assigns the value of **'Polar bear'** to the **animal** variable. Otherwise, assign the value '**Penguin'** to the animal variable.
5	Output a message to the console: **Console.WriteLine($"The animal that lives in the {pole} Pole is the {animal}");** Run the app and confirm the logic works as expected.
6	Now assign the value of **Pole.South** to your **pole** variable and perform the same conditional test using the **ternary statement**.

7	Output a message to the console: **Console.WriteLine($"The animal that lives in the {pole} Pole is the {animal}");** Run the app and confirm the logic works as expected. You should now have two outputs: 
8	You will now practise with *switch statements* and *switch expressions*. Add an **enum** to the project called **CapitalCities**: 
9	In **Program.cs**, declare and initialise the following variables: 
10	Write a **switch statement** that switches on the **city** value against the four values in the enumeration. Within each block, assign a message to the **countryMessage** variable: **countryMessage = $"{city} is the capital of France";**
11	Add a default case label and after the switch statement, output the following: **Console.WriteLine(countryMessage);**

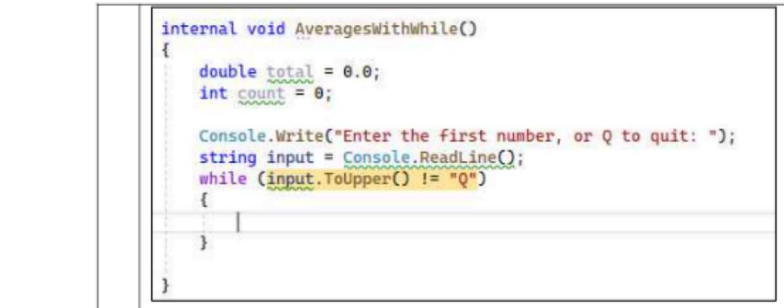
12	Now see if you can achieve the same behaviour with a **switch expression**. Assign the value of **Paris** to the **city** variable *before* the switch expression and output a message to the console *after* the switch expression.
13	When you have completed your code, run the program. You should now have the following output: 
14	A suggested solution is provided in the **End** folder for your reference.



Loops and Collections

The objective of this exercise is to consolidate your understanding of loops and collections.

1	Create a new **console** project called **Averages** in: **C:\Courseware\QACS\Labs\05_Loops_and_Collections\Begin**
2	In **Program.cs**, delete the comment and the line of code and replace with the following code: `AverageCalculator calculator = new AverageCalculator();` `calculator.AveragesWithWhile();` `Console.WriteLine("==========");` `calculator.AveragesWithDoWhile();` `Console.WriteLine("==========");` `calculator.AveragesWithFor();`
3	Resolve the issues as follows: • Ctrl-dot on AverageCalculator, and select **Generate class in new file** • Ctrl-dot on each of the three methods and select **Generate method** • In each of the three generated methods in the AverageCalculator class, remove the line with the NotImplementedException
4	In the **AveragesWithWhile** method, put the following code: `double total = 0.0;` `int count = 0;` `Console.Write("Enter the first number, or Q to quit: ");` `string input = Console.ReadLine();`
5	On the following line, type **'while'** then press **tab twice** to insert a code snippet. As the snippet is inserted, the word 'true' is highlighted. Overtype that with: `input.ToUpper() != "Q"` The full method should now look like the following:



```
internal void AveragesWithWhile()
{
    double total = 0.0;
    int count = 0;

    Console.Write("Enter the first number, or Q to quit: ");
    string input = Console.ReadLine();
    while (input.ToUpper() != "Q")
    {

    }

}
```

6 Inside the while loop, put the following code:

```
total += double.Parse(input);
count++;

Console.Write("Enter another number, or Q to quit: ");
input = Console.ReadLine();
```

7 After the end of the while loop, put the following code:

```
Console.WriteLine($"The average of those numbers is {total / count}");
```

8 Test **AveragesWithWhile** by running the program.

Press F5, enter some numbers when prompted, and when you're ready to see the average, enter **Q** to quit.

9 Take a moment to look at the code you've just written. Note the following points:

- In the condition for the while loop, you use **input.ToUpper** rather than just **input**. This means that the loop will end when the user types either 'q' or 'Q', because you turn the user's input to upper-case before comparing it to the letter 'Q'.
- You have to read from the console immediately before the start of the while loop to decide whether to enter it for the first time. You do this again at the end of the loop to decide whether to repeat the loop. This pattern, where you set a variable before the loop and then change it at the very end of the loop, is quite common.
- Note how you use **double.Parse** to parse the string input from the user. **Console.ReadLine** will only ever return a string, so if you want to treat the input data as any other data type, it will always need to be parsed first.
- If the user types '**Q**' in place of the first number, the loop will not execute at all. This is a key feature of while loops; they execute zero or more times.

	• What do you think the outputted result will be if you type 'Q' in place of the first number? Try it and see if you are right.
10	In the previous step, when you tried entered '**Q**' to quit without performing any average, the program displayed the output '**NaN**'. This stands for 'Not a Number', because when you divide by zero there is no valid numeric answer. (As an aside, if you divide an *integer* by zero, you will get an *exception*, but if you divide a *double* by zero the answer is *NaN* and there is no exception.) Let's fix that problem. Replace the final **Console.WriteLine** with the following: `if (count == 0)` `{` `  Console.WriteLine("You didn't enter any numbers");` `}` `else` `{` `  Console.WriteLine($"The average of those numbers is {total / count}");` `}`
11	Confirm the behaviour of the program by running it and immediately entering 'Q' versus running it and entering some valid numbers before quitting.
12	Now see if you can achieve the same behaviour with a **do...while** loop. Write code within **AverageWithDoWhile**. Use the 'do' code snippet to get started.
13	When you have completed your code, run the program. When you press F5, the first sequence of numbers you enter is controlled with a **while** loop. After you press Q, it will prompt you for a second set of numbers; this is what is being controlled by the new code that you have written using a **do...while** loop. Check that both types of loops give correct averages.
14	Now see if you can achieve the same behaviour with a **for** loop. Write code within **AverageWithFor**. Use the '**for**' code snippet to get started.

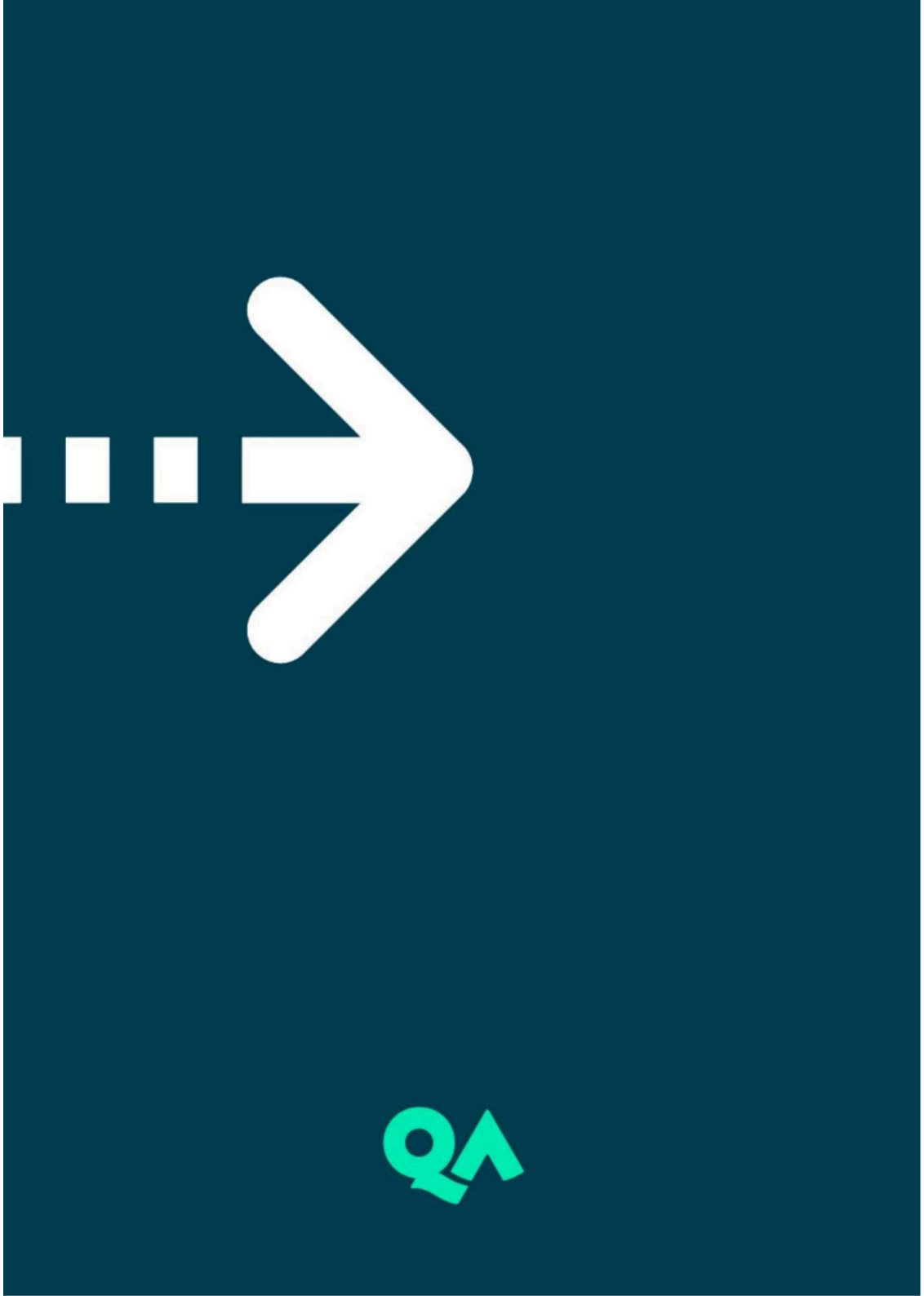
	Because **for** loops run a set number of times, start off by asking the user how many numbers they have before entering the loop.
15	When you have completed your code, run the program. When you press F5, the first sequence of numbers you enter is controlled with a **while** loop and the second set are controlled by the **do...while** loop. You should then be prompted for how many numbers you have before the **for** loop calculates and displays the average. Check that all three loops give correct averages.
16	A suggested solution is provided in the **End** folder for your reference.

Collections

1	Create a new **console** project called **Collections** in: **C:\Courseware\QACS\Labs\05_Loops_and_Collections\Begin**
2	Delete the contents of **Program.cs**.
3	Create an *array of strings* called **muppets** to store the following values: 'Kermit the Frog', 'Miss Piggy', 'Fozzie Bear', 'Gonzo', 'Rowlf the Dog', 'Scooter', 'Animal', 'Rizzo the Rat', 'Pepe the Prawn', 'Walter', 'Clifford'
4	Use a **foreach** loop to write the strings to the console.
5	Create a strongly typed *list of strings* called **muppetList**.
6	Add the contents of the muppets array to the list using a single method call.
7	Output the following values to the console: • The first muppet • The last muppet
8	Add a new string to the list: '**Beaker**'

9	Output the value of the last muppet in the list to the console and confirm it is now 'Beaker'.
10	You will now confirm the list is strongly typed by attempting to add non-string types to the list: • Attempt to add the Boolean value *true* to the list • Attempt to add the int value *3* to the list Comment out any code that will prevent a successful build.
11	Your next task is to sort the list. Firstly, use a loop to display all the strings on one line separated by commas. Then, sort the list and re-display the strings to confirm they are sorted alphabetically.
12	You will now use different operators to extract specific strings from the list. Use an *indexer*, the *index from end* and *range* operators, to extract the following strings: • The first string • The last string • The second to last string • A slice of the strings in position 5 and 6
13	You will now create a *dictionary* to store strings for the keys and values. Call this dictionary **muppetdict**.
14	Add the following items to the dictionary: • 'Beaker', 'Meep' • 'Miss Piggy', 'Hi-ya!' • 'Kermit', 'Hi-ho!' • 'Cookie Monster', 'Om nom nom'
15	Create a variable **catchphrase** and extract the value for **Miss Piggy** Output this to the console.

16	You will now write three loops to iterate over the **KeyValue** pairs, **Keys**, and **Values** respectively. Within each loop, output a string containing the iterated item.
17	A suggested solution is provided in the **End** folder for your reference.



Object-Oriented Programming (OOP)

The objective of these exercises is to consolidate your understanding of object-oriented programming (OOP) principles.

Exercise 1 – OOP Principles

The following abbreviations apply:

[**C**]lass or classes, [**O**]bject or objects

[**Y**]es [**N**]o

Only one box should be ticked for each question.

Please tick the appropriate box (class(es) or object(s)).

		C	O
1	Behaviour is defined in a(n)	☐	☐
2	Fields are defined in a(n)	☐	☐
3	Field values exist in a(n)	☐	☐
4	A message is (normally) sent to a(n)	☐	☐
5	Inheritance is a relationship between	☐	☐

Please respond to the statements by ticking a box (yes or no).

		Y	N
6	An object is an instance of a class.		
7	An object must belong to a class.		
8	A class must have an object.		
9	Behaviour is implemented by methods.		
10	A message is a request from one object to another object.		
11	An object always expects a response from a message.		
12	Derived classes inherit behaviour from their base class(es).		
13	A base class inherits the fields and methods of their child classes.		
14	A class can have a field of another class.		
15	A derived class is another name for a base class.		
16	A derived class can have more than one base class.		
17	In class 'Mammal', 'Weight' could be a field.		
18	In the class 'StopWatch', 'Start' could be a behaviour.		
19	A base class has the same number as or fewer fields than a derived class.		

Solutions overleaf

Page 30

Author: Pete Behague Subject: Highlight Date: 11/29/2022 3:23:29 PM

Solutions: Exercise 1 – OOP Principles

		C	O
1	Behaviour is defined in a(n)	X	
2	Fields are defined in a(n)	X	
3	Field values exist in a(n)		X
4	A message is (normally) sent to a(n)		X
5	Inheritance is a relationship between	X	

Please respond to the statements by ticking a box (yes or no).

		Y	N
6	An object is an instance of a class.	X	
7	An object must belong to a class.	X	
8	A class must have an object.		X
9	Behaviour is implemented by methods.	X	
10	A message is a request from one object to another object.	X	
11	An object always expects a response from a message.		X
12	Derived classes inherit behaviour from their base class(es).	X	
13	A base class inherits the fields and methods of their child classes.		X
14	A class can have a field of another class.	X	
15	A derived class is another name for a base class.		X
16	A derived class can have more than one base class.		X
17	In class 'Mammal', 'Weight' could be a field.	X	
18	In the class 'StopWatch', 'Start' could be a behaviour.	X	
19	A base class has the same number as or fewer fields than a derived class.	X	

Exercise 2 – OOP Relationships

For each of the following questions, select the correct relationship (association / aggregation / inheritance / interface). Think carefully about the context.

Then, if you think it's an:

Association: Suggest a role name for each end of the association

Aggregation: Suggest a delegated operation

Inheritance: Suggest an inherited operation

Interface: Suggest an implemented operation

Remember:

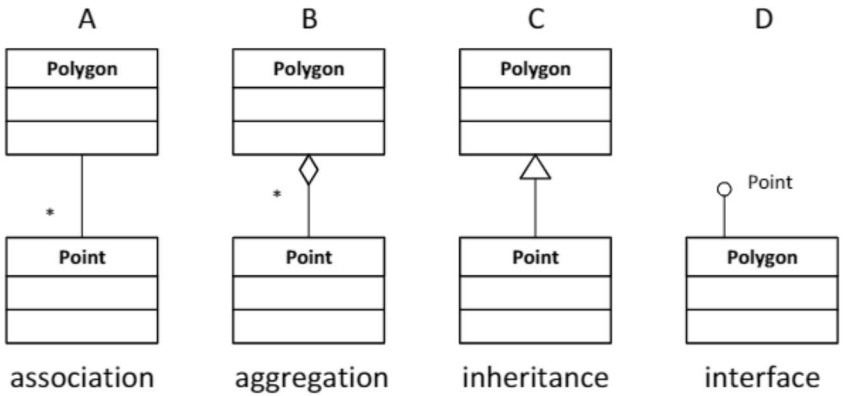
Association = *"Knows about"*

Aggregation = *"Made up of"*

Inheritance = *"Is a type of"*

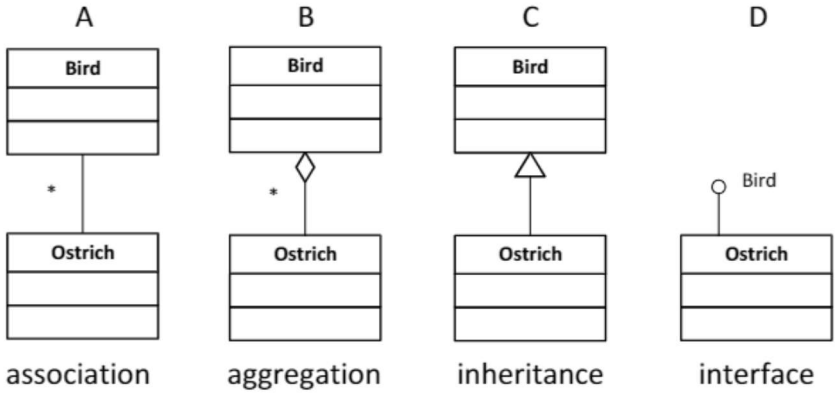
Interface = *"Can do"*

Polygon / Point



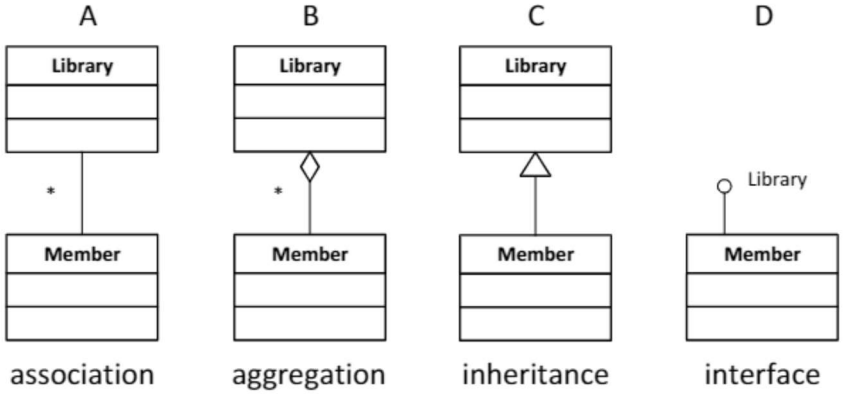
Bird / Ostrich

Context: A Film Animation Company



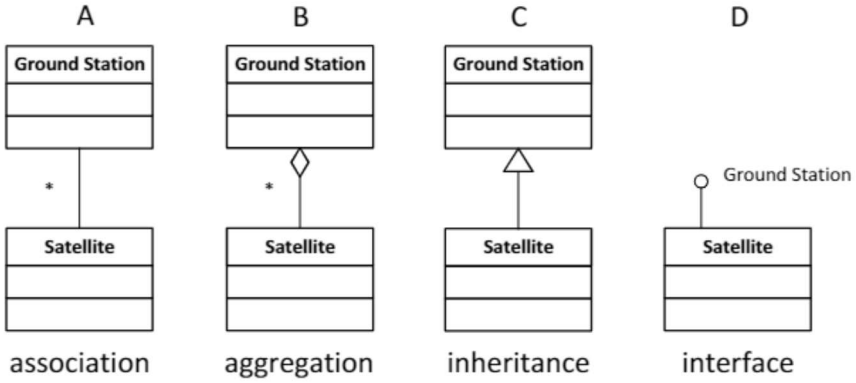
Library / Member

Context: A Public Library Checkout system

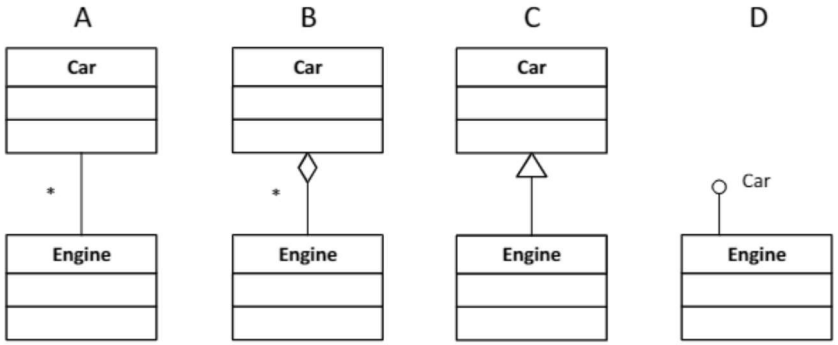


Ground Station / Satellite

Context: Goonhilly, which monitors any satellite it can see

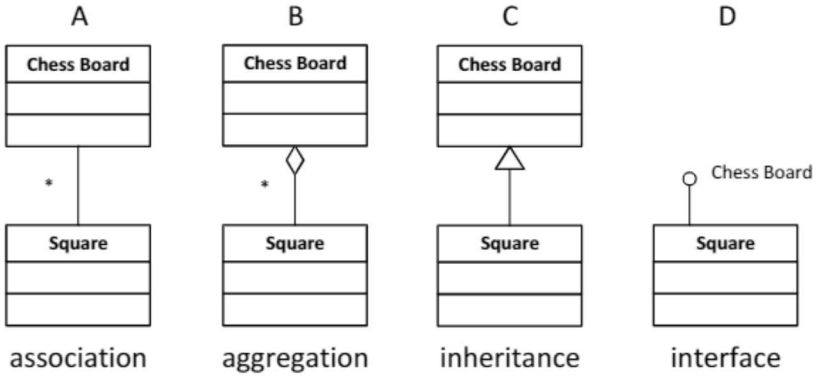


Car / Engine



Chess Board

Context: A conference room repeater display of a chess match

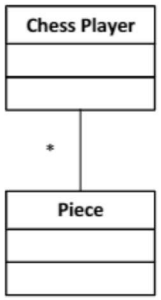


Author: Pete Behague Subject: Highlight Date: 11/29/2022 3:38:57 PM

Chess Player

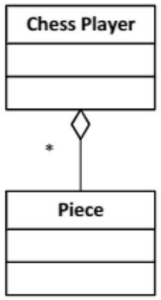
Context: The repeater board software. The "Player" is software, not the real player.

A



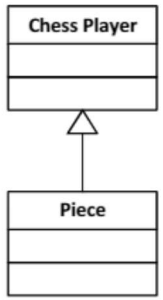
association

B



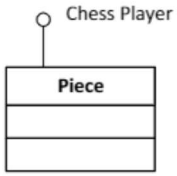
aggregation

C



inheritance

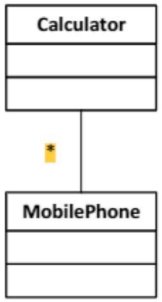
D



interface

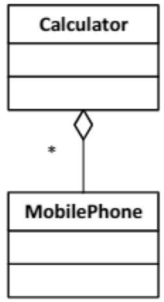
A Mobile Phone's Calculator app

A



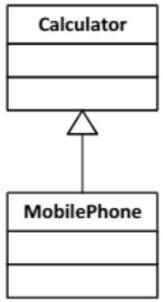
association

B



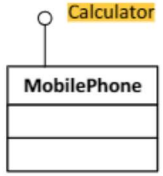
aggregation

C



inheritance

D



interface

Solutions overleaf

Page 36

Author: Pete Behague Subject: Highlight Date: 11/29/2022 3:44:00 PM

Author: Pete Behague Subject: Highlight Date: 11/29/2022 3:42:18 PM

Solution: Exercise 2 - OOP Relationships

Polygon / Point

Aggregation.

A delegated operation: Move()

Bird / Ostrich

It's inheritance, *provided* you can define meaningful behaviour for "Fly()" – assuming your Birds fly.

If your Birds really have to fly in the air then an Ostrich is *not* a type of Bird. It might be a type of Flightless Bird (i.e. a specialisation of Bird) provided there is also a sub-type Flighty Birds (which can fly). Of course, if the animation company is making a movie about an ostrich that flies, then there's no problem!

You could have Flyable as an interface.

An inherited operation: Fly() [or at least, FlapWings()!]

Library / Member

Aggregation.

A delegated operation: CheckoutBook()

Ground Station / Satellite

Association. If the Ground Station controlled the satellites, then there is an argument for aggregation.

Role Names of association: tracker – target

Car / Engine

Aggregation.

The car is the sum of its parts – of which Engine is one.

A delegated operation: Start()

Chess Board

Aggregation. The board is a collection of squares and achieves its purpose through the squares.

Delegated operation: Draw() (implemented by asking each square to draw itself)

Chess Player

Even though the English flows as a "Chess Player *has* Pieces" and the real human player achieves their purpose by moving Pieces, in the software world the Player is just the owner of the pieces and the software player does not move the pieces (this might be different for a chess-playing game, rather than our repeater software).

So, association.

Role Names: owner – owned by

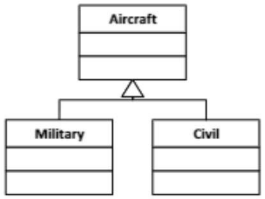
A mobile phone that has a calculator

This can be represented either as aggregation (but with the parent class being Mobile Phone) or by interface. It is not inheritance – a mobile Phone is *not* a type of calculator.

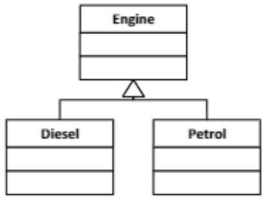
Exercise 3 – OOP Subclassing

For these questions state whether, *given the supplied context*, sub-classing is correct or not. Give a yes/no answer and suggest a reason. The acid test is 'do they have different rules, or is it just a passive piece of data?'.

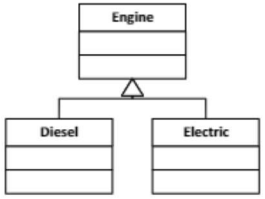
Context: Air Traffic Control



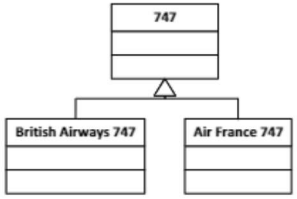
Context: Fuel Duty Payable



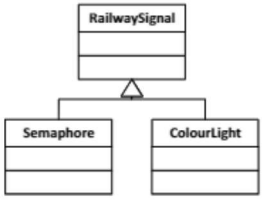
Context: Passenger making a train journey



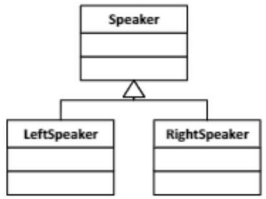
Context: Paris CDG Air Traffic Control



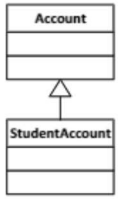
Context: Railway Signal



Context: Hi-fi system

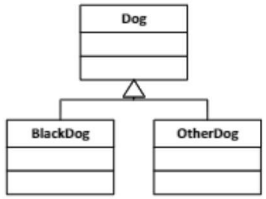


Context: Bank Account

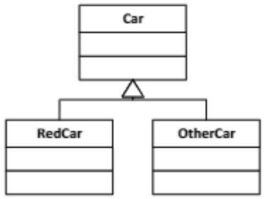


The only difference between a student account and a regular account is that on opening, a student gets a £50 “sweetener”.

Context: Dog Training Club



Context: Insurance Company



Solutions Overleaf

Solution: Exercise 3 - OOP Subclassing

Context	?	Notes
Air Traffic Control	Yes	Military & Civil aircraft operate under different rules
Fuel Duty Payable	Yes	Again, different rules
Passenger making a journey	No	Do they care?
Paris Airport ATC	Yes	Air France planes get priority in France.
Signal	Yes	
Stereo Hi-Fi	No	We just have 2 speakers. We call one left & one right.
Student Account	No	It's just a bit of (starting) data that has no effect on subsequent behaviour. This should be handled by a parameter passed in at Object construction time. However, we do need to store this rule somewhere.
Dog Training Club	No	The colour of the dog is just a bit of data and does not affect behaviour. However, I have heard that Golden Retrievers just cannot be trained the same way as other dogs, so there may be sub-classing "Golden Retriever" and "Other".
Insurance Company	No	There are many more factors than just colour taken into account when assessing risk.

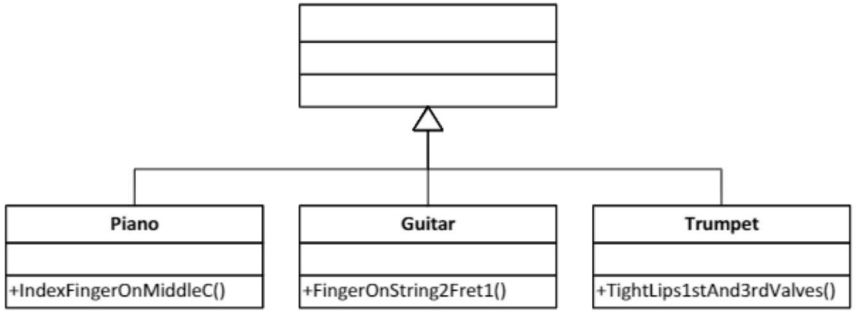
Exercise 4 - OOP Superclass Operations

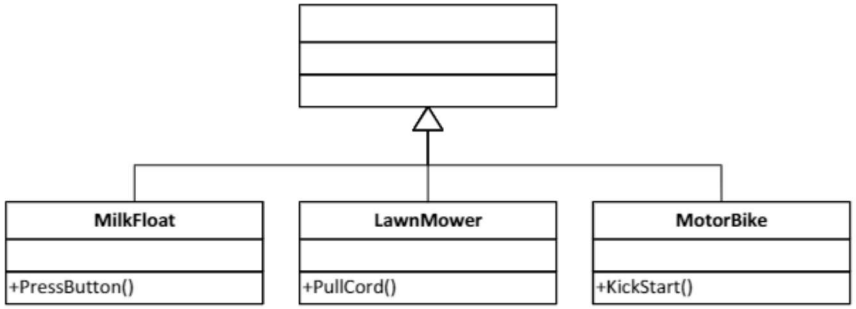
In the following two scenarios, some sub-classes are shown with methods.

Give a name to the superclass and state what the polymorphic operation might be at the superclass that may call the given methods in the subclasses. Assume that the polymorphic method at the subclass will merely call the stated method.

Remember that the superclass should have no knowledge of the subclass's behaviour; this extends to choosing a suitable name for the superclass operation so that the name doesn't inadvertently expose the behaviour.

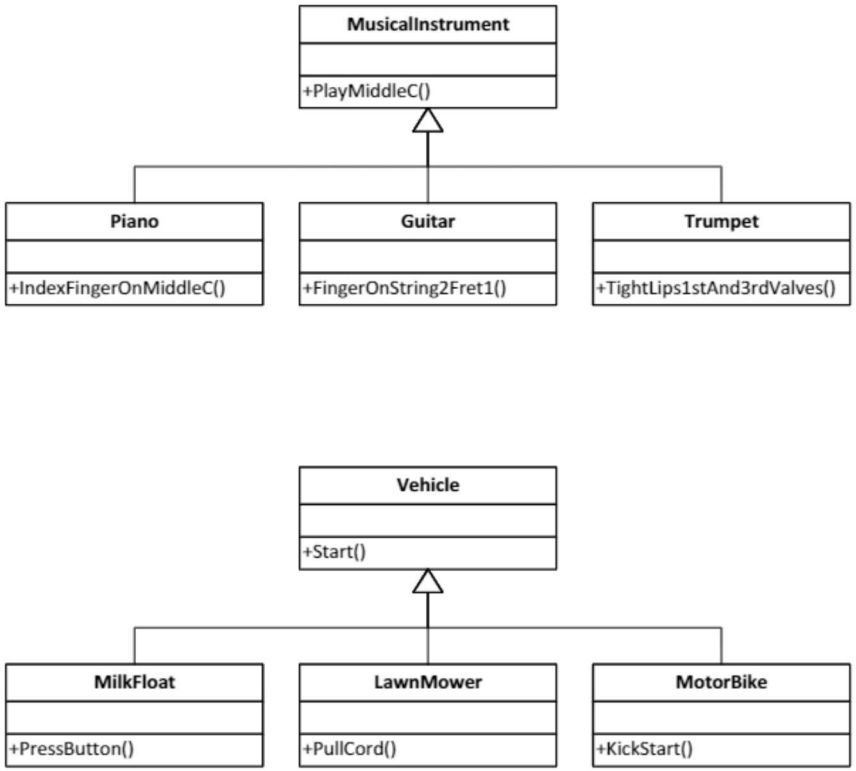
Note that although the exercise refers to super classes, the same could equally be applied to interfaces.

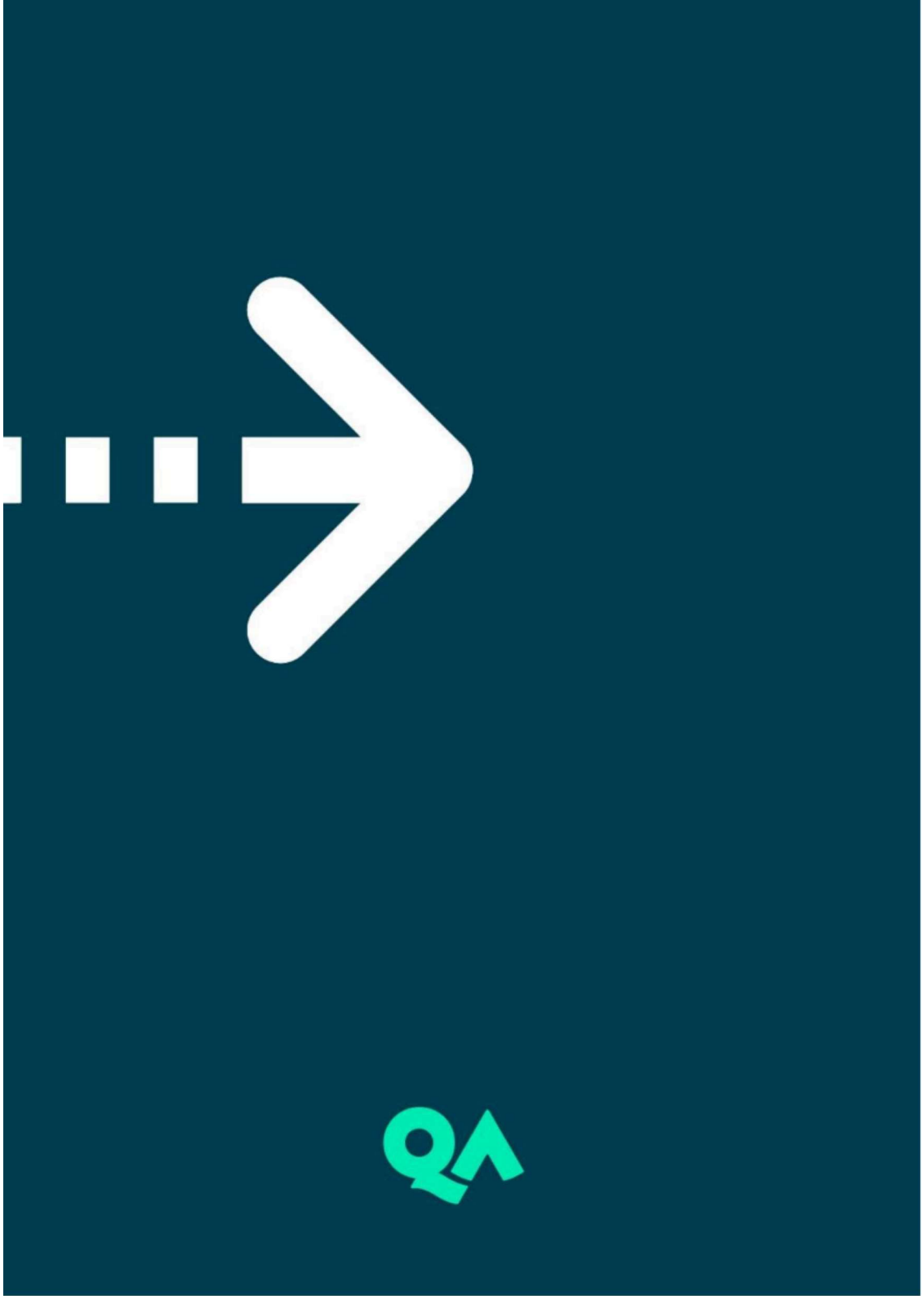




Solutions Overleaf

Solution: Exercise 4 - OOP Superclass Operations

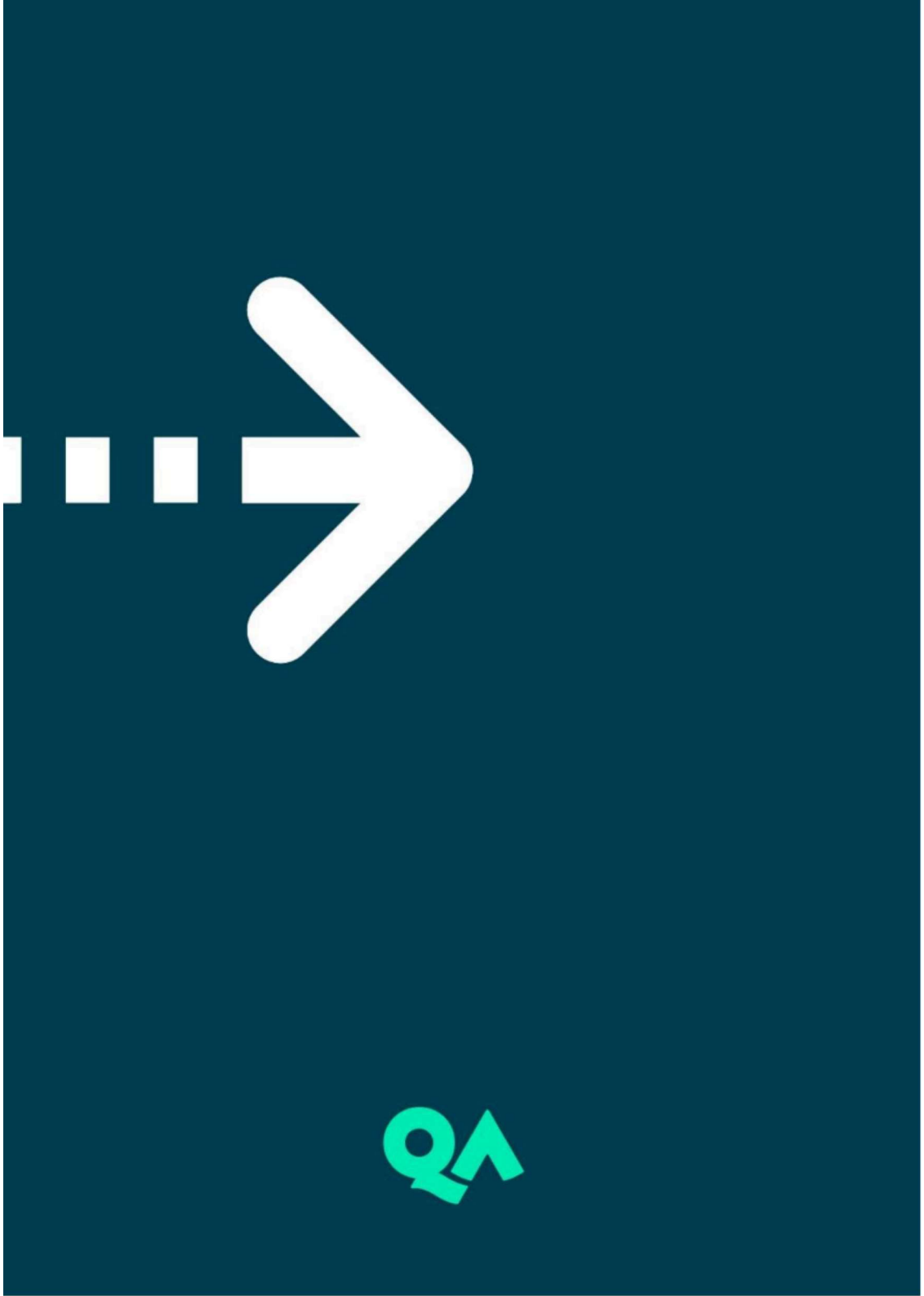




Methods

The objective of this exercise is to consolidate your understanding of C# syntax, value and reference types, method parameters, statics, and extension methods.

1	Create a new Class Library project called **MethodsLibrary**. Delete **Class1.cs**
2	Add an XUnit project to the Solution called **TestProject**. Rename **UnitTest1.cs** to **MethodTests.cs** Add a Project Reference to the **MethodsLibrary** project.
3	In the **Assets** folder, you will find many notepad files. Each one contains a test or two. For each test, copy the test code into **MethodTest.cs**. Use your knowledge of C# to resolve the issues, one test at a time, to ensure each test passes. Use the Visual Studio Quick Actions and/or **Ctrl+dot** wherever possible, but remember that not everything can be solved with Quick Actions.
4	Solutions are provided in the **End** folder for your reference.



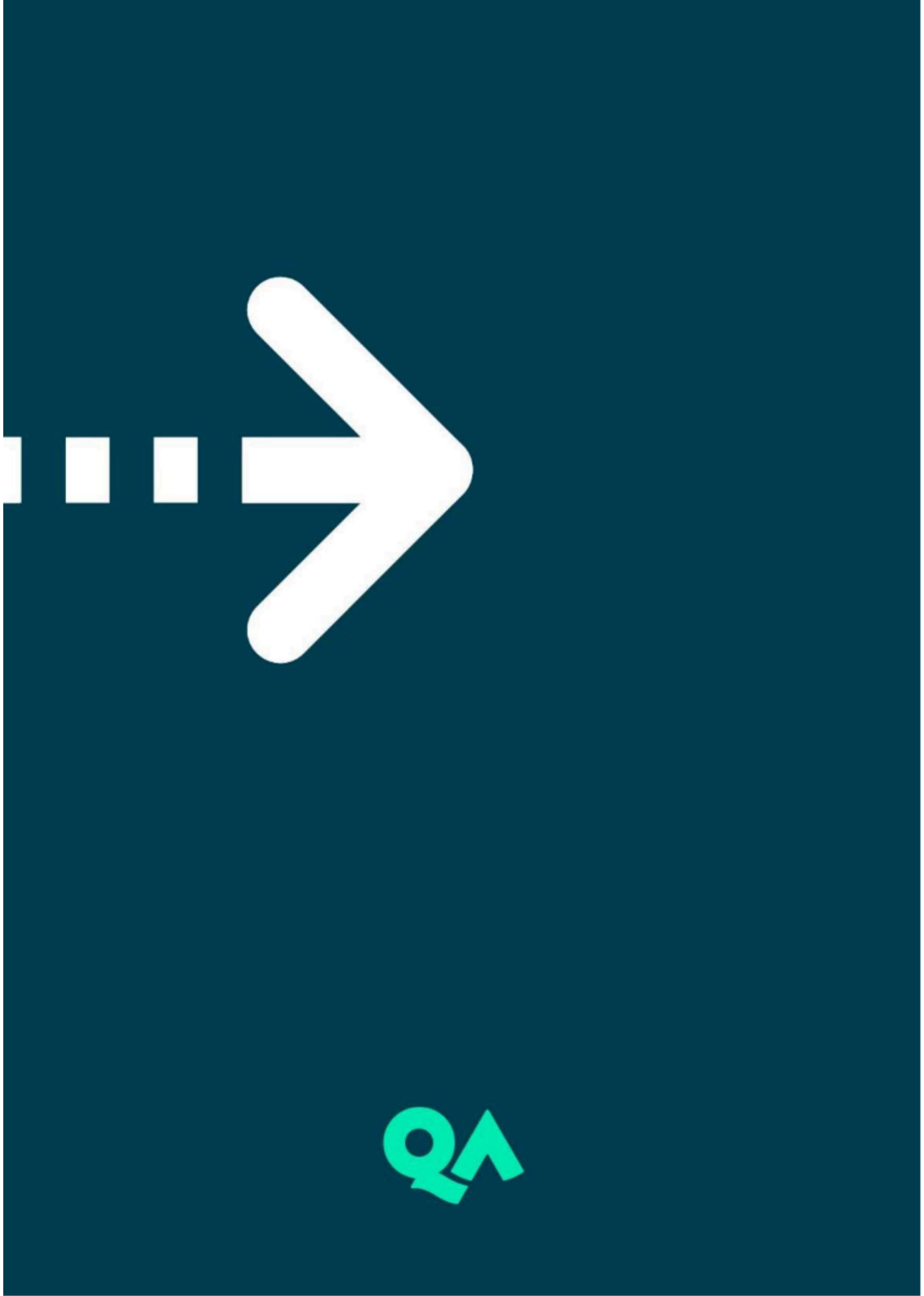
Properties and Constructors

The objective of this exercise is to consolidate your understanding of C# properties and constructors.

1	Create a new Class Library project called **CarLibrary**. Rename **Class1.cs** to **Car.cs**
2	Add a Console Application project to the Solution called **CarConsole**. Set **CarConsole** as the start-up project. In **CarConsole**, add a project reference to the **CarLibrary** project.
3	In **Car.cs**, create a property of type *int* called **Speed** with a *backing field* **speed.** Validate that the speed set is above zero but under 100.
4	Add an auto-implemented property of type *string* called **RegistrationNumber**.
5	Add a calculated expression bodied property called **SpeedInKilometres** of type *double*. To calculate the speed in kilometres, multiply the speed by 1.609344
6	Add string properties for **Make**, **Model**, and **Colour**.
7	In **CarConsole**, in **Program.cs**: Delete the line of code that outputs 'Hello, World!' Instantiate a car object, **c1**. Issue a using directive to bring the **CarLibrary** namespace into scope. Write the name of the instance to the console: Console.WriteLine(nameof(c1)); Build and run the console application to confirm the object can be successfully instantiated.

	Set the make of **c1** to be '**Ford**'. Write the *make* of c1 to the console. Write the *model* of c1 to the console. What value is displayed?
8	In the **Car** class, create a constructor that accepts a *make* and a *model* only. Initialise these values within the constructor.
9	In **CarConsole**: Re-run the app. Does it build successfully?
10	Create a *parameterless* constructor Set the make and model to be **Unknown** and the colour to be **Black**. Confirm the console app builds and runs successfully. What value is displayed for the model?
11	In **CarConsole**: Instantiate a car object, **c2**, using the overloaded constructor. The make is **Audi**, the model is **TT**; Write the make and model of **c2** to the console. Set the colour property to **Red**. Write c2's colour property to the console. Set the speed of **c2** to **30 miles per hour**. Display the speed in the console in both *miles per hour* and *kilometres per hour*.
12	In **CarConsole**: Instantiate a car object, **c3**, using the overloaded constructor (**BMW**, **X5**) and an object initialiser that sets the colour to **Grey** and the registration number to **ABC 123**. Write the property values of **c3** to the console.

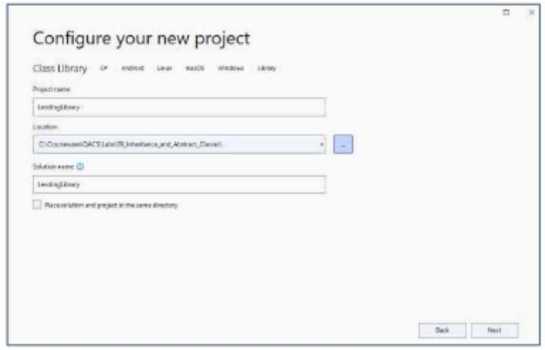
13	In **Car.cs**, chain the parameterless constructor to the overloaded constructor, passing **Unknown Make** and **Unknown Model** as the parameters. In the body of the parameterless constructor, remove the make and model and set the colour to be **White**. Confirm the console application still builds and runs successfully.
14	In **CarConsole**: Instantiate a car object, **c4**, using the parameterless constructor. Write the property values of **c4** to the console. Confirm **c4** is an unknown make and model that is white with an empty registration number.

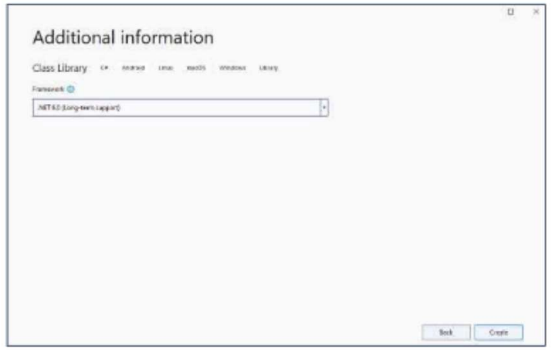


Inheritance

The objective of this exercise is to consolidate your understanding of inheritance by building an inheritance hierarchy.

1 Create a new Visual Studio project of type Class Library called **LendingLibrary**.





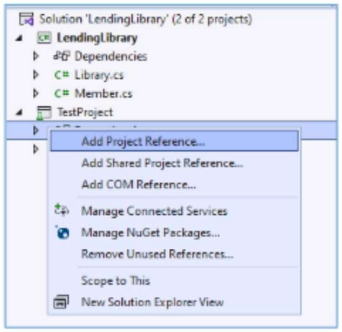
Delete **Class1**

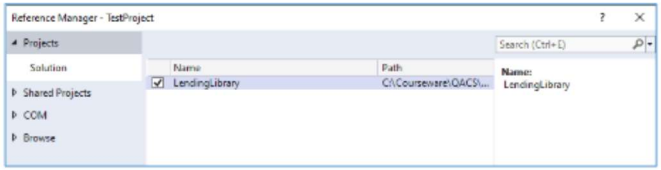
Add to this solution an XUnit Test project called **TestProject**

2	In the Test project, replace the starter test with the 'Create' test as found in the Assets folder (copy just the first part of the file, up to the end of the test). `[Fact]` `public void Create()` `{` `  Library library = new Library();` `  Member greta = library.Add(name: "Greta Thunberg", age: 15);` `  Member donald = donald = library.Add(name: "Donald Trump", age: 73);` `  Assert.Equal(2, library.NumberOfMembers);` `  Assert.Equal(1, greta.MembershipNumber);` `  Assert.Equal(2, donald.MembershipNumber);` `}`
3	Create **Library** and **Member** classes in the **LendingLibrary** project, then copy in the code listed just after the 'Create' test. `namespace LendingLibrary` `{` `  public class Library` `  {` `  Dictionary<int, Member> members = new Dictionary<int, Member>();` `  public int NumberOfMembers => members.Keys.Count;` `  int GetNextFreeMembershipNumber()` `  {` `  return (members.Keys.Count == 0) ? 1 : members.Keys.Max() + 1;` `  }` `  public Member Add(string name, int age)` `  {` `  Member member = new Member(name, age, GetNextFreeMembershipNumber());` `  members.Add(member.MembershipNumber, member);` `  return member;` `  }` `  }` `}`

```
namespace LendingLibrary
{
    public class Member
    {
        public string Name { get; }
        public int MembershipNumber { get; }
        public int Age { get; }

        public Member(string name, int age, int membershipNumber)
        {
            this.Name = name;
            this.Age = age;
            this.MembershipNumber = membershipNumber;
        }
    }
}
```

In the test project, add a project reference to the **LendingLibrary** project.





Add a using statement 'using LendingLibrary;' to **UnitTest1.cs**

Run the **Create** test and confirm it passes.

4 We are going to need the **library** instance and **greta** and **donald** member objects in further tests, so to avoid duplication of code, move the declarations of these to the class level and initialise them in the constructor.

	Note: If you get stuck, we've shown it in the Assets folder.
5	Later on, we're going to need some additional properties of **Member**, so add these in and get Visual Studio to create the properties. `public UnitTest1()` `{` `  library = new Library();` `  greta = library.Add(name: "Greta Thunberg", age: 15);` `  greta.Street = "Queen Street";` `  greta.City = "Stockholm";` `  greta.OutstandingFines = 25M;` `  donald = donald = library.Add(name: "Donald Trump", age: 73);` `  donald.Street = "Trump Tower";` `  donald.City = "New York";` `  donald.OutstandingFines = 2500M;` `}`
6	In this library we have different rules for borrowing a book, dependent on whether the member is over 16 or not. We are going to need this enum. Add it to your **LendingLibrary**, in a file called BookCategory.cs `namespace LendingLibrary` `{` `  public enum BookCategory` `  {` `  Children,` `  Adult` `  }` `}`
	We will need a **Book** class. Add the class and the properties and get Visual Studio to generate the constructor: `public class Book` `{` `  public string Title { get; }` `  public BookCategory Category { get; }` `  public int BookCode { get; }` `  public Book(string title, BookCategory category, int bookCode)` `  {` `  Title = title;` `  Category = category;` `  BookCode = bookCode;` `  }` We will also need some Book code in the **Library** class:

```
Dictionary<int, Book> books = new Dictionary<int, Book>();

public Book GetBook(int code)
{
    return books?[code];
}

public Library()
{
    books.Add(100, new Book("Walls have ears", BookCategory.Adult, 100));
    books.Add(101, new Book("Noddy goes to Toytown", BookCategory.Children, 101));
}
```

7 In the TestProject, add in the **Child_Borrows_Child_Book_OK** test from the **Assets** folder.

Get Visual Studio to resolve the **Borrow()** method:

Populate the method like this:

```
public bool Borrow(Book book)
{
    return true;
}
```

Run the Test and confirm it passes.

8 Add in the test **Child_Borrows_Adult_Book_Fails**.

```
[Fact]
public void Child_Borrows_Adult_Book_Fails()
{
    // a junior member (under 16) can borrow only child category books
    Book adultBook = library.GetBook(100);
    Assert.False(greta.Borrow(adultBook));
}
```

Run this Test. It fails because it is currently hard-coded to return true.

	Now we need to modify Borrow: `public bool Borrow(Book book)` `{` `  return book.Category == BookCategory.Children;` `}` Re-run the test. It should now pass.
9	Add the test **Adult_Can_Borrow_Any_Book.** `[Fact]` `public void Adult_Can_Borrow_Any_Book()` `{` `  // an adult member (over 16) can borrow any book` `  Book adultBook = library.GetBook(100);` `  Book childBook = library.GetBook(101);` `  Assert.True(donald.Borrow(adultBook));` `  Assert.True(donald.Borrow(childBook));` `}` Run this Test. It fails because the **Borrow**() method only returns true for children's books. Modify the **Borrow** code again: `public bool Borrow(Book book)` `{` `  if (Age >= 16)` `  {` `  return true;` `  }` `  else` `  {` `  return (book.Category == BookCategory.Children);` `  }` `}` Run the Tests. All tests should now pass.
10	In this library, fines are handled differently for juniors and adults. Juniors must provide a CashFund; Adults must provide BankTransfer details. Add in the two 'Fines' tests:

```
[Fact]
public void Child_Pays_Fine_From_Cash_Fund()
{
    greta.CashFund = 20M;
    greta.PayFine(7M);
    Assert.Equal(13M, greta.CashFund);
}

[Fact]
public void Adult_Pays_Fine_By_Bank_Transfer()
{
    donald.SetUpBankTransferLimit(20M);
    donald.PayFine(7M);
    Assert.Equal(13M, donald.BankTransferAvailable);
}
```

Add this to your **Member** class:

```
public decimal CashFund { get; set; }

public void PayFine(decimal fine)
{
    if (Age < 16)
    {
        CashFund -= fine;
    }
    else
    {
        BankTransferAvailable -= fine;
    }
}

public decimal BankTransferAvailable { get; private set; }
public void SetUpBankTransferLimit(decimal amount)
{
    BankTransferAvailable += amount;
}
```

Confirm all tests pass.

Where we are so far

OK – it works.

But can you see how we are constantly changing working code as we discover more about the junior and adult rules that apply to this library.

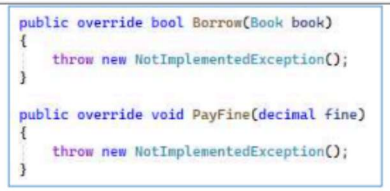
In a real project, this means we are *constantly* breaking working code.

Now we will switch to **Inheritance** to see if this helps the situation.

11	We are going to refactor the **Member** class. In order to compare versions, we will do this: 1) Copy+Paste **Member.cs**

	2) Rename the original to **Member1.cs**. When it asks you if you want Visual Studio to perform a rename, answer **No**. 3) Rename the copy to **Member2.cs**. 4) In Member1.cs, press **Ctrl+A Ctrl+K Ctrl+C** to comment out the entire class. 5) Create tqo folders in LendingLibrary: a. 01 Without Inheritance b. 02 With Inheritance 6) And move Member1.cs to **01 Without Inheritance** and Member2.cs to **02 With Inheritance**. Your project now only has one uncommented Member class in Member2.cs. All tests will pass as no code has been changed.
12	In the 02 With Inheritance folder, add two new classes: **JuniorMember** and **AdultMember**. Adjust their namespaces to be just **LendingLibrary**.
13	Get both of these subclasses to derive from the **Member** base class. Implement a constructor that passes all parameters to the base class constructor: `namespace LendingLibrary` `{` `  public class JuniorMember : Member` `  {` `  public JuniorMember(string name, int age, int membershipNumber) :` `  base(name, age, membershipNumber)` `  {` `  }` `  }` `}` Do the same for **AdultMember**.
14	We need to modify **Library.Add** to create either a *Junior* or an *Adult* member. Change Library.Add() to this:

	`` public Member Add(string name, int age) { Member member; if (age < 16) { member = new JuniorMember(name, age, GetNextFreeMembershipNumber()); } else { member = new AdultMember(name, age, GetNextFreeMembershipNumber()); } members.Add(member.MembershipNumber, member); return member; } ``` All tests should pass.
15	Actually, we want to disallow creating 'Member' objects – clients should be forced to create either Junior or Adult members. Make the Member class **abstract**.
16	In Member2.cs, copy Borrow(), CashFund, PayFine(), BankTransferAvailable and SetUpBankTransferLimit() into notepad, deleting them from Member.
17	In Member, insert these two abstract methods: ``` public abstract bool Borrow(Book book); public abstract void PayFine(decimal fine); ``` These abstract methods replace the concrete versions we just deleted.
18	Go to **JuniorMember**. You will see a red squiggly. ``` public class JuniorMember : Member { public JuniorMemb base(name, ag { } } ``` class LendingLibrary.JuniorMember CS0534: 'JuniorMember' does not implement inherited abstract member 'Member.PayFine(decimal)' CS0534: 'JuniorMember' does not implement inherited abstract member 'Member.Borrow(Book)' Show potential fixes (Alt+Enter or Ctrl+.) Using **Ctrl+dot**, implement the Abstract class. This will put in the signatures of the methods defined in Member:

	
19	Inside each of these members, copy the relevant code from that which you stored in notepad that is specific just to *Junior* members Repeat for AdultMember. • You will no longer need the 'if' statement because you are removing the code that doesn't apply • You will need to paste in the **CashFund** property for the JuniorMember, and the **BankTransferAvailable** and associated **SetUpBankTransferLimit()** method for the AdultMember.
20	If you now go back to the tests, you can see that it doesn't know that greta is a JuniorMember and Donald is an AdultMamber: `public class UnitTest1` `{` `Library library;` `Member greta;` `Member donald;` There are two ways of fixing this: 1) Make greta a Junior and donald an Adult. 2) Find a form of word that works for both such that the client software is unaware as to whether they are Junior or Adult. We'll do both...
21	Make these changes:

	`` Library library; JuniorMember greta; AdultMember donald; public UnitTest1() { library = new Library(); greta =(JuniorMember) library.Add(name: "Greta Thunberg", age: 15); greta.Street = "Queen Street"; greta.City = "Stockholm"; greta.OutstandingFines = 25M; donald = donald = (AdultMember)library.Add(name: "Donald Trump", age: 73); donald.Street = "Trump Tower"; donald.City = "New York"; donald.OutstandingFines = 2500M; } ```
22	The tests will pass, however, this is not a great solution because the client code is now acutely aware of the subclasses, meaning if a new subclass is invented, for example, *StudentMember*, you will have to modify the client code.
23	Use Ctrl+z (Undo) to remove the changes you made in step 21.
24	A better way of looking at it is: *'Is there some form of words that could operate at the Member level (i.e., for every type of Member) that makes sense to both Junior and Adult members and could be interpreted correctly by both of them?'* i.e., the same intent but they have different implementations? How about **SetFineLimit**() and **GetFineCredit**() ?
25	Make these changes: TestProject

```
[Fact]
public void Child_Pays_Fine_From_Cash_Fund()
{
    greta.SetFineLimit(20M);
    greta.PayFine(7M);
    Assert.Equal(13M, greta.GetFineCredit());
}

[Fact]
public void Adult_Pays_Fine_By_Bank_Transfer()
{
    donald.SetFineLimit(20M);
    donald.PayFine(7M);
    Assert.Equal(13M, donald.GetFineCredit());
}
```

Member:

```
public abstract void SetFineLimit(decimal amount);
public abstract decimal GetFineCredit();
```

When adding methods to JuniorMember and AdultMember, you should use **ctrl-dot** on the red squiggly to create the method signatures for you, then delete the **NotImplementedException** and replace it with the relevant code for the specific class.

JuniorMember:

```
private decimal CashFund { get; set; }// now private

public override void SetFineLimit(decimal amount)
{
    CashFund = amount;
}

public override decimal GetFineCredit()
{
    return CashFund;
}
```

AdultMember:

```
private decimal BankTransferAvailable { get; set; } // now private

public override void SetFineLimit(decimal amount)
{
    SetUpBankTransferLimit(amount);
}

public override decimal GetFineCredit()
{
    return BankTransferAvailable;
}
```

All tests should pass.

State Pattern

If you don't have time:

Then the moral of this section is 'Always ask the question – can a subtype morph into another subtype?'. For example, can a dog become a cat, keeping the original mammally bits? If the answer is 'No', as is the case with dogs and cats, then inheritance (as we've done so far in this lab) is fine.

But often one type can morph into another. In our case, a JuniorMember can become an AdultMember when they turn 16. Therefore, the statement should be:

A LibraryMember has a MembershipType (and a Junior MembershipType is a type of MembershipType)

rather than:

A LibraryMember is a Junior Member

If you have time, then carry on:

Note: If this section is difficult to appreciate, don't worry. It's something you should, in time, be aware of.

Unfortunately, even though the solution we've just developed looks really elegant as there are almost no 'if' statements in there and all the concerns are separated, there is still a problem.

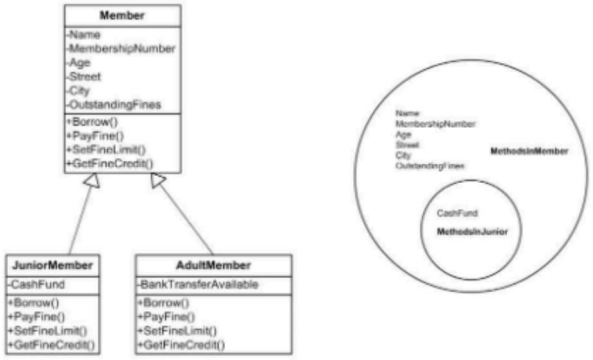
What happens when Greta turns 16?

Easy, you say – you just create her as an AdultMember

The rules of inheritance are that you cannot morph one type into another, so you have to destroy the original JuniorMember and create a brand new AdultMember. This presents some problems:

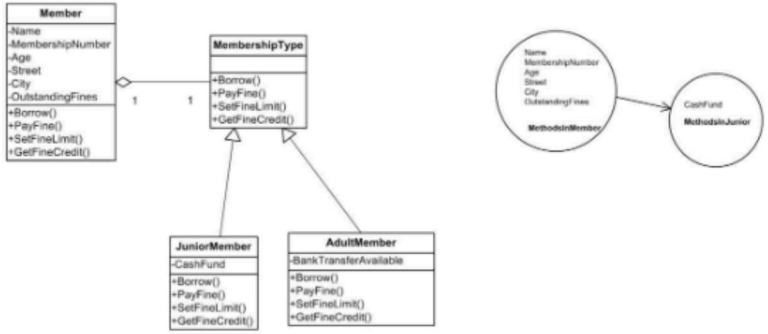
1. You have to transfer all the data (name, street, city, etc.) - what if some of that data is private?
2. Greta will feature in various collections. She would need to be removed from these and the new Greta would need to be inserted silently (probably breaking the rules we have established).

Our current class diagram and a representative object is shown below:



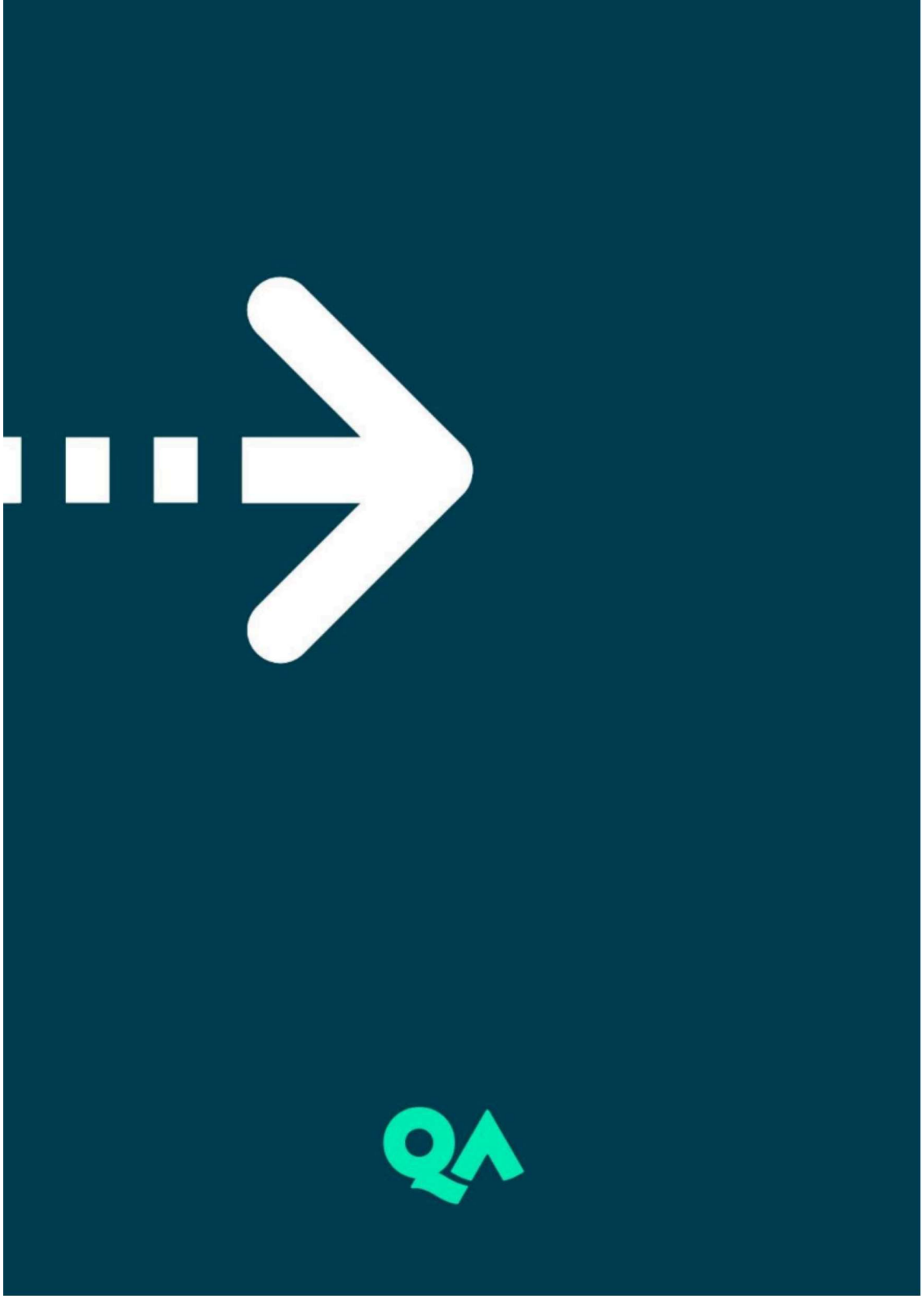
It's like we have one object built from two recipes – the Member recipe and the JuniorMember recipe. And when we destroy this object, we not only throw away the CashFund and MethodsInJunior (which is fine), we also throw away all the data and methods in the Member bit of the object.

What we need is the following. We need two objects such that we only throw away the fields and methods specific to being a Junior. For example, a Member has a MembershipType rather than a Member is a Junior from birth to death.



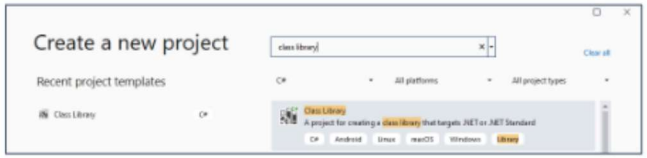
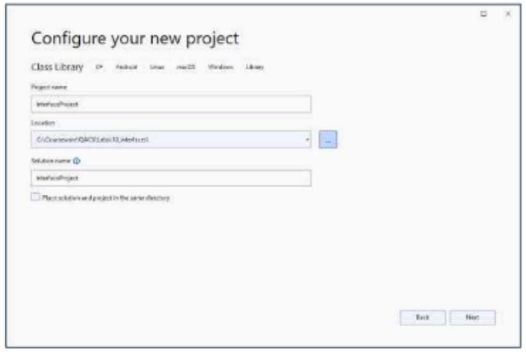
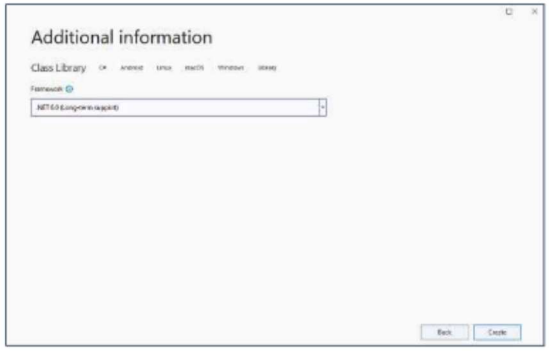
If you have plenty of time and feel very confident, then go ahead and have a go at changing your solution to be the State Pattern.

However, if it's been a long lab already, it's best to open the solution (**End2**), go to the **View** menu and select **Task List**. We have already made the required changes, and we have also annotated the few changes that were needed.

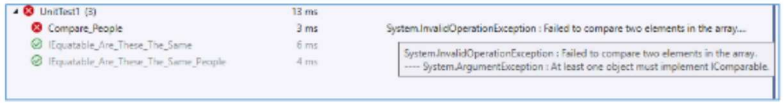


Interfaces

The objective of this exercise is to consolidate your understanding of interfaces.

1	Create a new Visual Studio project of type Class Library called **InterfaceProject**.    Delete **Class1**. Add to this solution an XUnit Test project called **TestProject**.

2	We have provided a **Person** class in the Assets folder. Drag this file onto your **InterfaceProject**. `namespace InterfaceProject` `{` `  public class Person` `  {` `  public string Name { get; }` `  public string Address { get; }` `  public DateTime Dob { get; }` `  public Person(string name, string address, DateTime dob)` `  {` `  Name = name;` `  Address = address;` `  Dob = dob;` `  }` `  }` `}`
3	Replace the provided Test1() with the two tests in **IEquatable.txt** from the **Assets** folder and resolve the red squiggles.
4	Run the tests. The first test fails, the second test passes by accident. The problem is that the compiler doesn't know how to figure out if two Persons are the same person. For example, if your bank account has a balance of £100 and so does mine – does that mean they are the same account?
5	Make Person implement the interface **IEquatable<Person>** This will make you implement the **Equals** method. As far as we are concerned, if the Name, Address, and DateOfBirth are the same, it is the same person. Put in this code and ensure both tests now pass. `public bool Equals(Person? other)` `{` `  return (` `  Name == other.Name &&` `  Address == other.Address &&` `  Dob == other.Dob` `  );` `}`

6	You will now compare some people, the Spice Girls, by putting them in order based on their DateOfBirth. We'll put these in DateOfBirth order. Copy in the test in the Assets folder called **Compare_People** and resolve the red squiggles. `[Fact]` `public void Compare_People()` `{` `  List<Person> spiceGirls = new List<Person>() {` `  new Person("Baby", "Finchley", dob:new DateTime(1976, 1, 21) ),` `  new Person("Posh", "Harlow", dob:new DateTime(1974, 4, 17) ),` `  new Person("Ginger", "Watford", dob:new DateTime(1972, 8, 6) ),` `  };` `  spiceGirls.Sort();` `  Assert.Equal("Ginger", spiceGirls[0].Name);` `  Assert.Equal("Posh", spiceGirls[1].Name);` `  Assert.Equal("Baby", spiceGirls[2].Name);` `}` Run the test. It will fail. Look at the error message:  It fails because it does not know how to sort Persons into order because they are not comparable.
7	Implement the interface **IComparable<Person>** Notice the **CompareTo** method returns an int. We want to rank Person by **DateOfBirth**, so: CompareTo should return **+1** if DoB is greater than the compared to object's' DoB. **-1** if smaller Otherwise return **0**. Enter the code and run the test to ensure it now passes.

8	In the Assets folder is a class **AssassinatedPresident**. Drag this onto your **InterfaceProject**. Add the test **Compare_Presidents**. `[Fact]` `public void Compare_Presidents()` `{` `  List<AssassinatedPresident> assassinations = new List<AssassinatedPresident>() {` `  new AssassinatedPresident("Kennedy", "Dallas", dob:new DateTime(1917, 5, 29), assassinated:new DateTime(1963, 11, 22)),` `  new AssassinatedPresident("Lincoln", "Ford Theatre", dob:new DateTime(1809, 2, 12), assassinated:new DateTime(1865, 4, 15)),` `  new AssassinatedPresident("Perceval", "Houses of Parliament", dob:new DateTime(1762, 11, 1), assassinated: new DateTime(1812, 5, 11)),` `  };` `  assassinations.Sort();` `  Assert.Equal("Lincoln", assassinations[0].Name);` `  Assert.Equal("Perceval", assassinations[1].Name);` `  Assert.Equal("Kennedy", assassinations[2].Name);` `}` Now implement **IComparable** for **AssassinatedPresident** where we want the sort order to be by the *month* in which they were *assassinated*. Run the Test and confirm it passes.
9	You will now experiment with cloning objects. Copy in the **Clone_A_Spice_Girl** test. The interface you need is **ICloneable**. Make Person implement the interface **ICloneable**. Conveniently, there is a protected method on the Object class which does this for us: `public object Clone()` `{` `  return this.MemberwiseClone();` `}` But you'll get a red squiggly in the test. This is because **Clone** method returns an object, so we have to cast the result to **Person** in the test: `[Fact]` `public void Clone_A_Spice_Girl()` `{` `  Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));` `  Person babyClone = (Person)baby.Clone();` `  Assert.Equal(baby, babyClone);` `}` Re-run the test. It should now pass.
10	This works, but it's not ideal.

It's a pity that the framework does not offer a generic version of **ICloneable.**

Have a go at creating your own generic **ICloneable<T>** interface.

Hint: Do an F12 on the generic interfaces **IEquatable** and **IComparable** to see how they are created.

Implement your generic version of **ICloneable<T>** in Person instead of the non-generic **ICloneable**. You can perform the explicit cast in the implemented **Clone** method.

Your test should no longer require the explicit cast and should be as follows:

```
[Fact]
public void Clone_A_Spice_Girl()
{
    Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));
    Person babyClone = baby.Clone();
    Assert.Equal(baby, babyClone);
}
```

Run the Tests. All tests should now pass.

11 You will now explore another interface **ITaxRules** and see how it is implemented in different classes.

Into your **TestProject**, copy the **Tax Tests** from the **Assets** folder:

```
[Fact]
public void Evaluate_UK_Tax()
{
    Product product = new Product(50M, new UKTaxRules(true));
    Assert.Equal(18.75M, product.GetTotalTax());
}

[Fact]
public void Evaluate_US_Tax()
{
    Product product = new Product(50M, new USTaxRules());
    Assert.Equal(7.5M, product.GetTotalTax());
}
```

We have provided all the code for you.

Copy the **Product** folder from the **Assets** folder and drop it onto your **InterfaceProject**.

	Have a look at the **Product** class. See that it is passed in an **ITaxRules** in its constructor but it doesn't know or care what sort of tax rules they, are as long as they implement **ITaxRules.** Review the code and confirm all tests pass.

Explicit interfaces

Because one class can implement many interfaces, it's possible that there could be a clash of method name. In the example coming up, we create an **AmphibiousVehicle** (this is the DUKW – pronounced Duck), and it was actually the US Army coding for such a vehicle.

D=1942

U=Utility

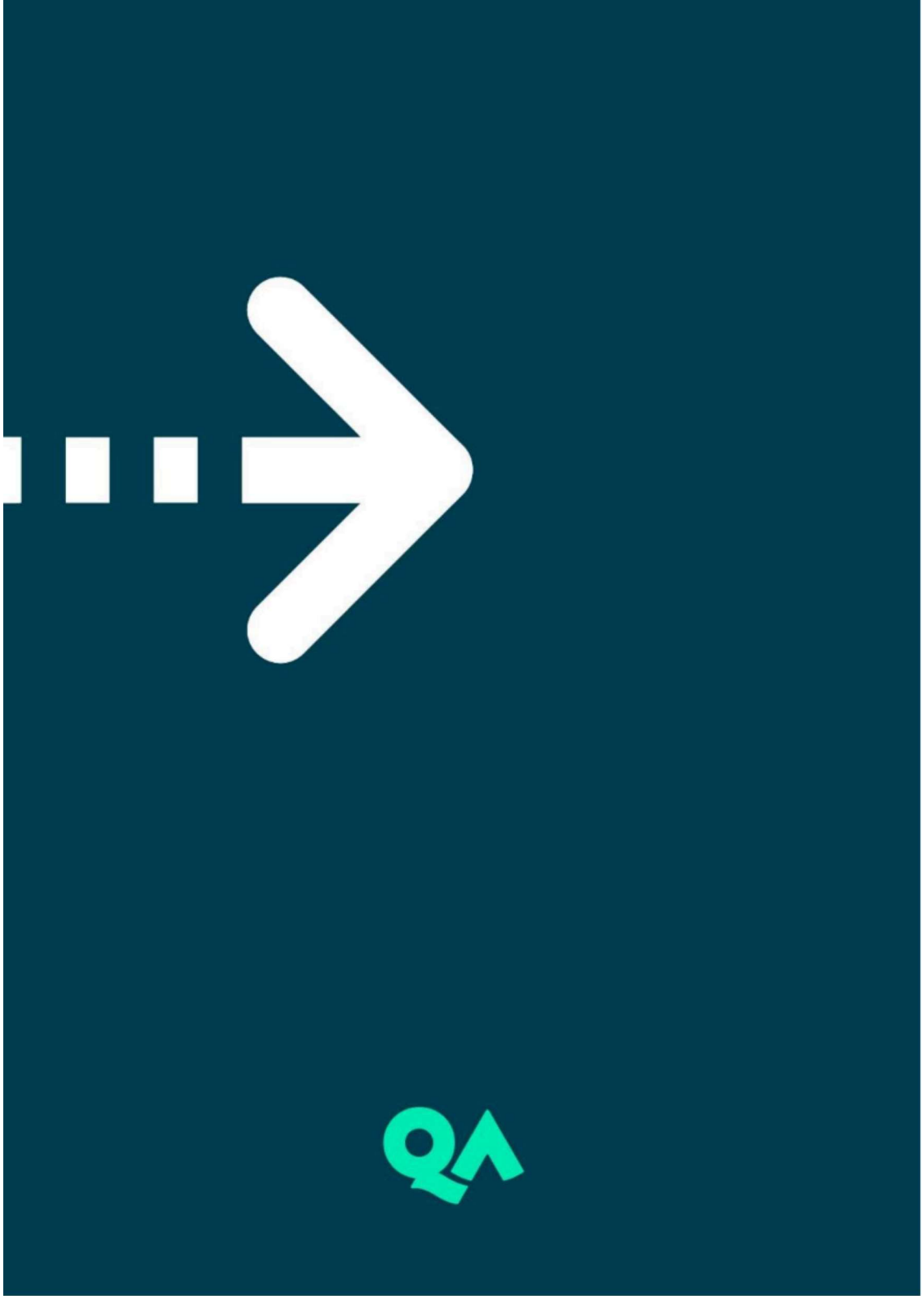
K=all-wheel drive

W=2 powered rear axles

This remarkable vehicle is truly a water vehicle and truly a land vehicle. In other technologies we might describe this by using multiple inheritance. .NET outlaws multiple inheritance, so we would achieve this by implementing multiple interfaces.

1	Drag the folder **DUKW** onto your **InterfaceProject** and have a look at the three files. In particular, notice that both interfaces have a **Brake** method. A land vehicle brakes by squeezing the disc pads. A water vehicle brakes by putting the propeller into reverse. Therefore, we need two methods.
2	Open the **AmphibiousVehicle** class and implement the interface **ILandVehicle**. Build the project to confirm there are no errors. You will see it gives you one method and both interfaces are satisfied. However, it gives you no chance to have two methods. Delete the **Brake** method.
3	Now implement the **IWaterVehicle** interface *explicitly* by placing the cursor on **IWaterVehicle** in **AmphibiousVehicle** and using **Ctrl+dot** -> **Implement all members explicitly**.

	Delete the **throw NotImplementedException**. Now place the cursor on **ILandVehicle** and use **Ctrl+dot** -> **Implement interface**. You get the Brake method. Delete the **throw NotImplementedException**.
4	You will now use a test to call the implicitly implemented brake method and the explicitly implemented break method. Add this test to **TestProject**: `[Fact]` `public void Explicit_Interfaces()` `{` `  AmphibiousVehicle av = new AmphibiousVehicle();` `  av.Brake();` `  ((IWaterVehicle)av).Brake();` `}` Add a breakpoint: `[Fact]` `public void Explicit_Interfaces()` `{` `  AmphibiousVehicle av = new AmphibiousVehicle();` `  av.Brake();` `  ((IWaterVehicle)av).Brake();` `}` Right-click in the test and choose **Debug Tests.** Step into the code using and **Step Into (F11)** to confirm that both brake methods are called.

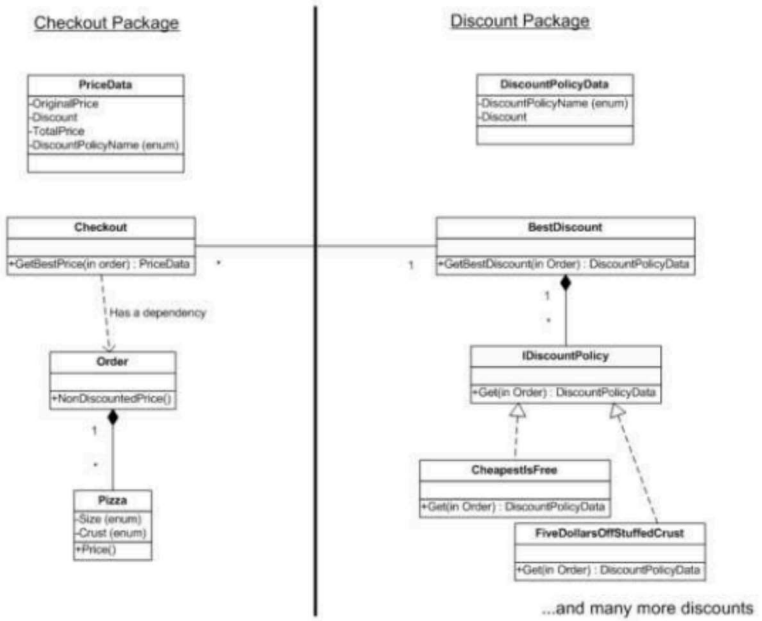


Delegates and Lambdas

The objective of this exercise is to consolidate your understanding of delegates and lambdas.

Scenario

We have the following classes and interfaces being used by a Pizza Ordering application.



An **Order** may contain one or many **Pizzas**.

Checkout needs to know the best discount to apply. We have a **BestDiscount** class to do this for us, which has a list of all available discounts. It passes in the order to each discount to see which one gets the best discount. It needs to return both the discount price and the name of the discount with said price. It does this by packaging the result into a **DiscountPolicyData** object and returning that to the Checkout.

The Checkout needs to know the price and name of the applied best discount, as well as what that discount was, the original price, hence the total to pay.

This exercise will take around 30 minutes.

1	Open the **'01 Pizza/Begin'** project and compile it.
2	To familiarise yourself with the problem, have a look at these classes: **Pizza** Notice that it has a Price property and it calculates the price from the size and the crust. To make it easier for you to follow the discounts, we have suffixed the size and crust with the price (so Small is actually Small_10 because its $10). We wouldn't do this in real life because it would be hard to maintain if the prices change, but for our purposes, it makes the tests easier to understand. **Order** Order is a List of pizzas. Notice how it sums the prices of each pizza to get the non-discounted total. **Checkout** The GetBestPrice is passed the order (which is a List of pizzas). It passes this off to the bestDiscount object to work it out. **BestDiscount** Has a list of IDiscountPolicies. To get the best discount, it asks each discount policy what its discount would be for this order. Just as an aside here, notice how we have an abstract class BestDiscount that is subclassed by various strategies (Weekday, Weekend, etc.) where we can apply different rules in different circumstances. **DiscountPolicyData** In BestDiscount, it compares these to see which gives the best discount. We have to implement IComparable. Finally, the policies themselves e.g., **CheapestIsFree**. There is no discount here if the order consists only of one pizza. If it's more than one, it loops round and remembers the cheapest.

3	Now look at the tests in **CheckoutTests**. We've written a few tests that return the discount, and it's here where the Small_10, Thin_4 etc. will help you see that it really has selected the best discount.
4	Run all tests and confirm they all pass.

As you can see, we can use Interfaces to achieve the decoupling we need – the BestDiscount class has no knowledge of the individual discount policies. It does have knowledge of IDiscountPolicy and all discount policies implement that.

But it is a bit clumsy:

For each discount policy, we must create a new class that implements IDiscountPolicy. That class has just one method which must be called 'Get'.

In this instance, we could have a neater syntax that doesn't care about the name of the discount policy method and just cares about the signature.

Look at the signature of the 'Get' method:

```
public interface IDiscountPolicy
{
    DiscountPolicyData Get(Order order);
}
```

We would describe this as a method that takes one **Order** as a parameter and returns something of type **DiscountPolicyData**.

We can use the **Func<T, TReturns>** delegate for this:

```
Func<Order,DiscountPolicyData> discountPolicy;
```

We refer to it as a generic delegate because you can provide any type. Note that, for Func<>, the last type is always the return type.

5	Delete the file **IDiscountPolicy.cs**
6	In the Discounts folder, create a **public static class DiscountPolicies**. This will contain all our policies. Change the namespace to just **PizzaProject**
	For each of the policies in the Policies folder, copy the **Get()** method and paste it into the class DiscountPolicies, replacing 'Get' with the name of the original class. Make the method static.

	For example, the method **CheapestIsFree.Get** becomes: `public static class DiscountPolicies` `{` `  public static DiscountPolicyData CheapestIsFree(Order order)` `  {` `  System.Diagnostics.Debug.WriteLine("CheapestIsFree");` `  var pizzas = order.Pizzas;` `  if (pizzas.Count < 2)` `  {` `  return new DiscountPolicyData(DiscountPolicyName.None, 0M);` `  }` `  // Loop round all pizzas in the order and get the one with the minimum price` `  decimal minPrice = decimal.MaxValue;` `  foreach (Pizza pizza in pizzas)` `  {` `  if (pizza.Price < minPrice)` `  {` `  minPrice = pizza.Price;` `  }` `  }` `  return new DiscountPolicyData(DiscountPolicyName.Cheapest_Is_Free, minPrice);` `  }` `}` Repeat for all discount policies.
7	Delete the folder **Policies** and the three files within it
8	If you compile now, it will fail because **BestDiscount** still relies on the now deleted **IDiscountPolicy**. Change this to: `protected List<Func<Order,DiscountPolicyData>> policies` `  { get; private set;}` `  = new List<Func<Order, DiscountPolicyData>>();`
9	Also, in BestDiscount, we now don't have classes like CheapestIsFree. `policies.Add(new CheapestIsFree());` But we do have methods that satisfy the signature of Func<Order,DiscountPolicyData>> Change it to: `policies.Add(DiscountPolicies.CheapestIsFree);`
10	Repeat for the other two policies. To make the code even simpler, put in a using statement: `using static PizzaProject.DiscountPolicies;`

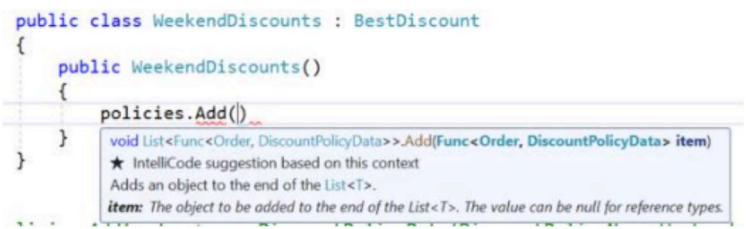
11	We are then left with one problem: `foreach (IDiscountPolicy policy in policies)` Resolve this as follows: `foreach (Func<Order,DiscountPolicyData> policy in policies)` `{` `    DiscountPolicyData thisOrdersPolicyData = policy(order);`
	Run the tests. They should all pass.

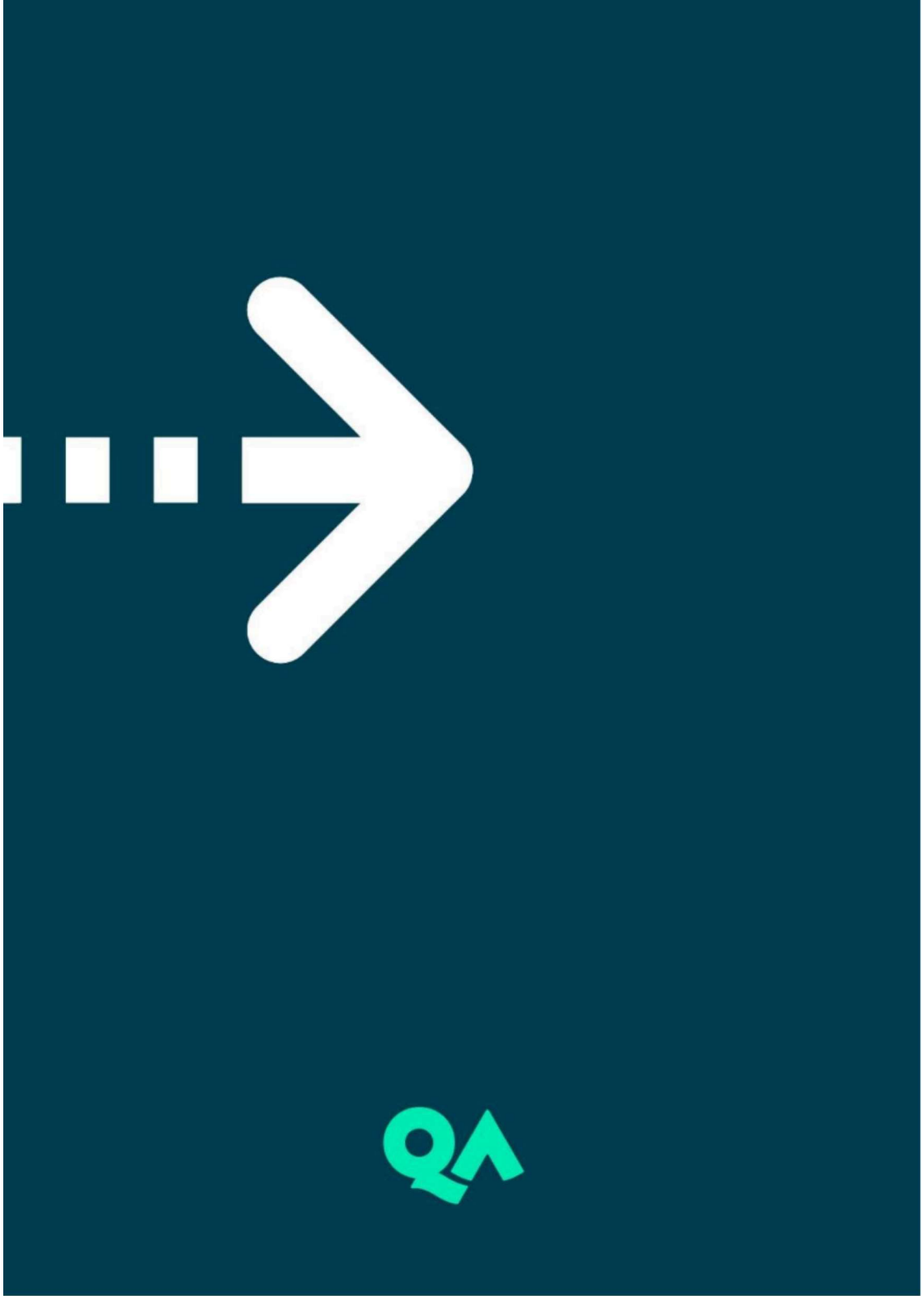
Have a think about what you've just done.

You had an interface with one method and wherever you define such a method it must be packaged into a class and you will probably never re-use either the class or the method. You have replaced this with a pointer to a method of the required signature.

Now we have delegates in play, we can use lambda expressions.

12	Add this test to your **CheckoutTests**. `[Fact]` `public void Weekend_Ten_Percent_Off()` `{` `    checkout = new Checkout(new WeekendDiscounts());` `    order.Pizzas.Add(new Pizza(Size.Small_10, Crust.Regular_2));` `    PriceData priceData = checkout.GetBestPrice(order);` `    Assert.Equal(10.8M, priceData.TotalPrice);` `    Assert.Equal(DiscountPolicyName.Weekend_10_Percent_Off,` `priceData.DiscountPolicyName);` `}` We want a new discount policy to operate on weekends where you get 10% off.
13	Add this to the **DiscountPolicyName enum**: `Weekend_10_Percent_Off`
14	In the BestDiscount file, there is a class WeekendDiscounts. Add a constructor. Within the constructor type the code:

	policies.Add(Have a look at the IntelliSense:  You will see it is expecting a parameter of type Func that takes an Order and returns a DiscountPolicyData. That means we can either point to a method of this signature (which is what we have done in the WeekdayDiscounts constructor) or we could put in a lambda expression.
15	Put in this lambda: `policies.Add(order=> new` `  DiscountPolicyData(DiscountPolicyName.Weekend_10_Percent_Off,` `  order.NonDiscountedPrice * 0.1M) );` Run the **Weekend_Ten_Percent_Off** test and confirm it passes.



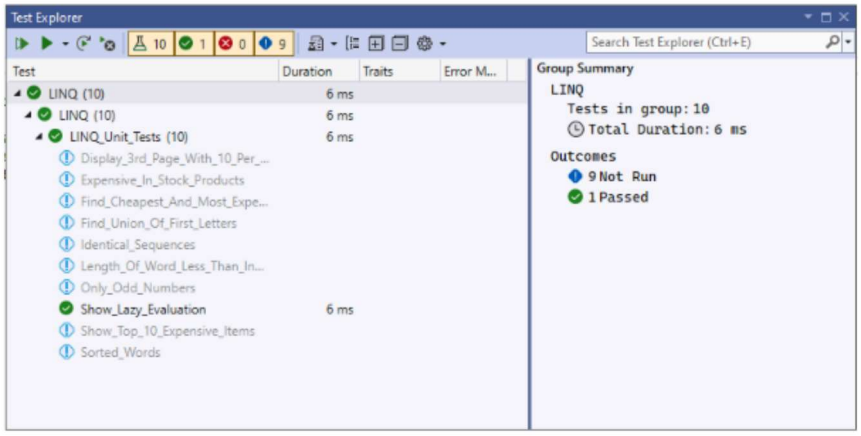
Language Integrated Query (LINQ)

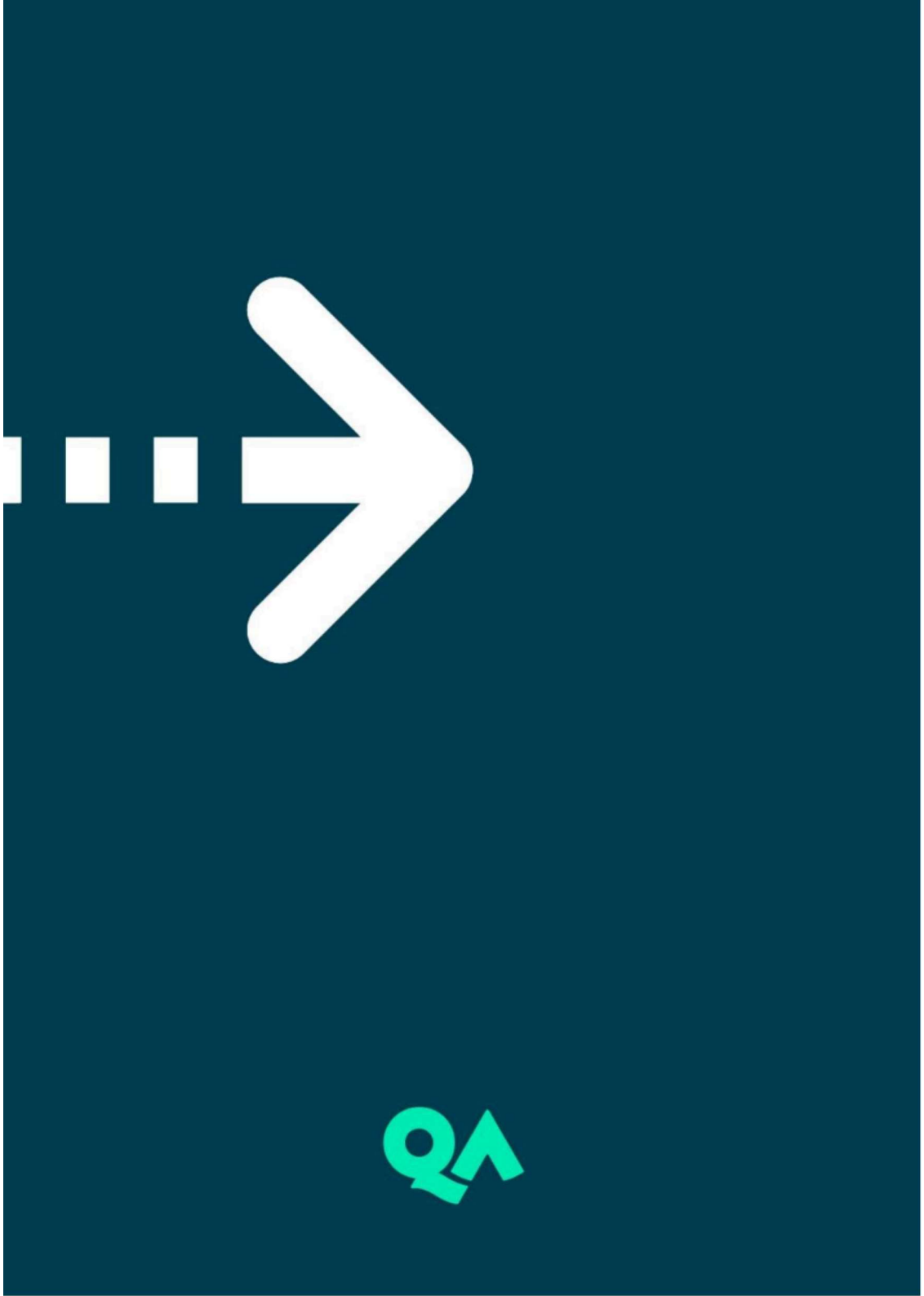
The objective of this exercise is to give you a wide view of the capabilities of Language Integrated Query (LINQ).

Exercise 1 – Writing LINQ Queries

Task 1 – Open the Solution

1. Open **LINQ_Solution.sln**
2. Open the **LINQ_Unit_Tests.cs** file and read the instructions at the top of the file.
3. Each test shows an imperative way of solving a problem. At the end of each test are clues as to how you might get started solving the same problem with LINQ. Have a go solving the problem with LINQ.
4. After writing the LINQ equivalent code, run each unit test by right-clicking within the unit test and choosing Run Tests. Ensure each test still passes.
5. As you progress, do a visual comparison of how much briefer the declarative LINQ solutions are compared to the provided non-LINQ solutions.





Exception Handling

The objective of this exercise is to consolidate your understanding of exception handling and creating and throwing custom exceptions.

1	Open the **CarLibrary** solution in: **C:\Courseware\QACS\Labs\13_Exception_Handling\Begin**
2	Comment out the existing code in **Program.cs**
3	Open **Car.cs** and override the **ToString** method to display detailed car information: `return $"Car Make is {Make}, Model is {Model}, Colour is {Colour}, Speed is {Speed} MPH";`
4	Add a new auto-implemented property called **RoadSpeedLimit**.
5	You will change the logic of the Speed setter to account for the road's speed limit and whether or not the value that you are setting is a legal driving speed for the current road. • If it is, set the value • If it is not, you will raise a custom exception
6	In a separate file in the class library project, create an exception class called **SpeedingException**. Don't forget to use inheritance.
7	Within the new custom exception class, create an auto-implemented property called **ExcessSpeed** and set this value within the constructor.
8	In **Car.cs**, if the car is not travelling at a legal speed, throw a new instance of **SpeedingException,** ensuring you pass in the excess speed value.
9	In **Program.cs**, create two new car instances: **slowCar** and **fastCar**
10	Set the following values for **slowCar**:

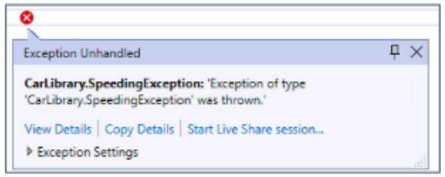
```
Car slowCar = new Car("Renault", "Clio");
slowCar.Colour = "Black";

slowCar.RegistrationNumber = "CLIO 1";
slowCar.RoadSpeedLimit = 30;
slowCar.Speed = 30;
Console.WriteLine(slowCar.ToString());
```

11 Set the following values for **fastCar**:

```
Car fastCar = new Car("BMW", "M5");
fastCar.Colour = "Silver";

fastCar.RegistrationNumber = "FAST 1";
fastCar.RoadSpeedLimit = 70;
fastCar.Speed = 80;
Console.WriteLine(fastCar.ToString());
```

12 Compile and run your application.

You should see an unhandled exception:



13 Uncomment the existing code in **Program.cs**.

Set the **RoadSpeedLimit** to **50** for Car **c2.**

Wrap the code in this file with **try...catch...finally** blocks to handle the exceptions that are thrown.

Add a catch block for **Exception** as well as for **SpeedingException**.

Utilise the properties of the exception class to display useful messages to the console:

```
Console.WriteLine($"A speeding exception occurred. The car is travelling {ex.ExcessSpeed} MPH above the limit");
```

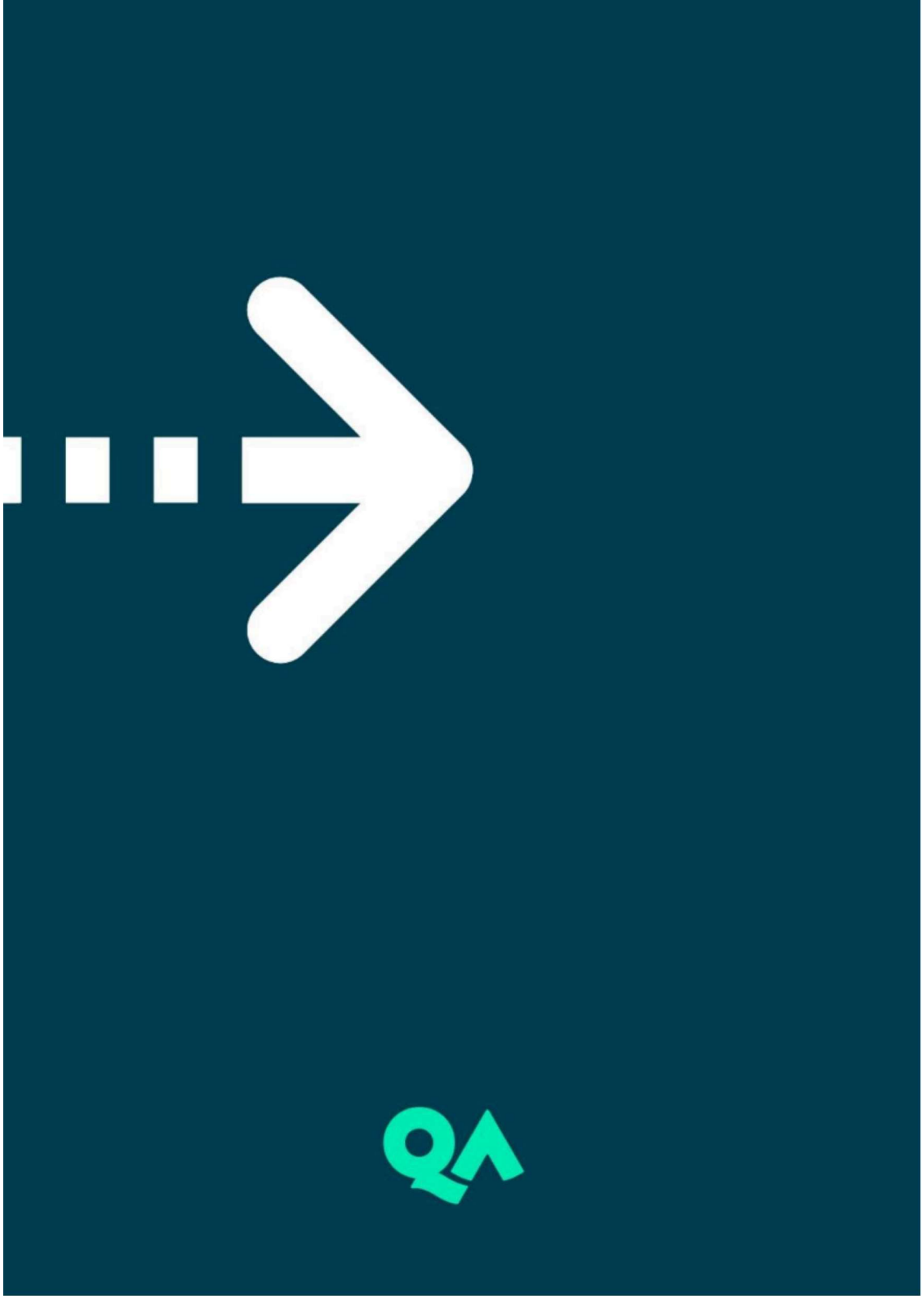
14 Compile and run your application.

Observe the exceptions that are thrown and caught.

15	Add a property to store the *Car instance* within the **SpeedingException** and use this to access information that can be output to the console to help identify the Car that is speeding: `catch (SpeedingException ex)` `{` `  Console.WriteLine($"A speeding exception occurred. The car is travelling {ex.ExcessSpeed} MPH above the limit");` `  Console.WriteLine($"A speeding exception occurred. Car {ex.Car.RegistrationNumber} is travelling {ex.ExcessSpeed} MPH above the limit");` `}`

If you have time

16	Create a list of valid colours within **Car.cs** and create a custom **InvalidColourException** that is thrown if the colour is not in the list.
17	Observe how this exception is caught by the generic Exception event handler.
18	Add a custom catch block to handle this specific type of exception.
19	A suggested solution is provided in the **End** folder for your reference.



FANCY A CHAT?

0345 757 3888

INFO@QA.COM

QALtd QA-Ltd QALtd QALimited

QA.COM

V1.0