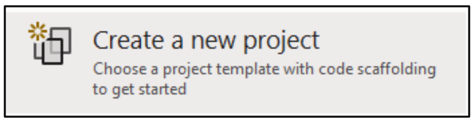
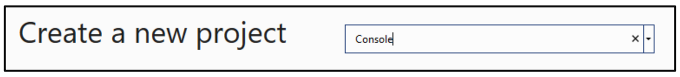
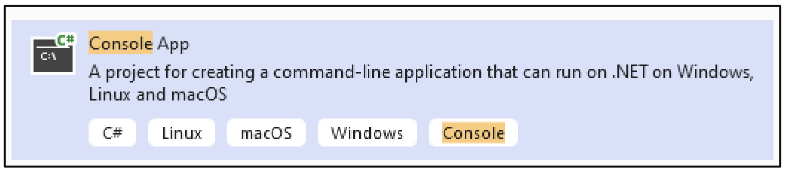
C# The Programming Language

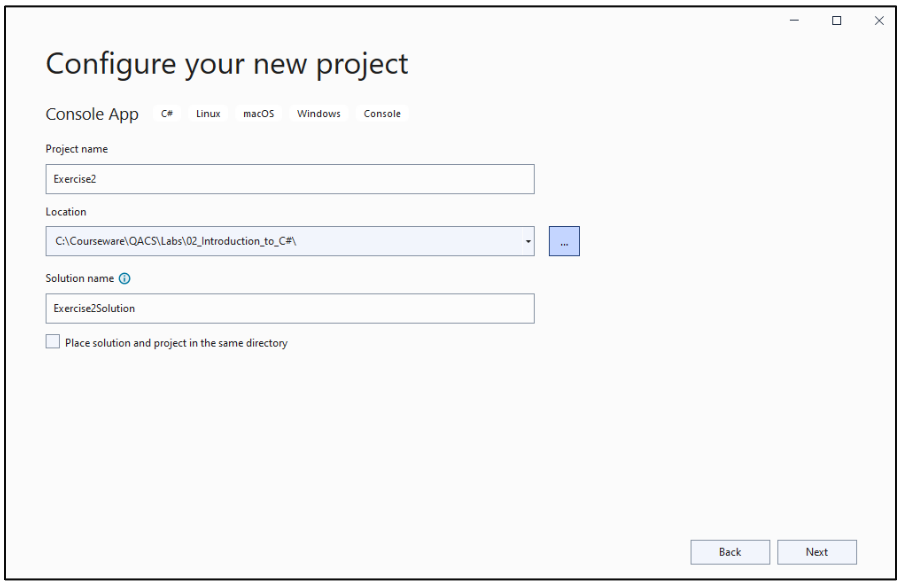
Exercise Workbook

Learn. To Change.

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'**  Create a new project Choose a project template with code scaffolding to get started
3	In the Search box, type **Console**  Create a new project Console
4	Select **Console App** and choose **Next**  Console App A project for creating a command-line application that can run on .NET on Windows, Linux and macOS C# Linux macOS Windows Console
5	Name the *Project* **Exercise2** and the *Solution* **Exercise2Solution**. Save the files in **\Labs\02_Introduction_to_C#\Begin**



Configure your new project

Console App · C# · Linux · macOS · Windows · Console

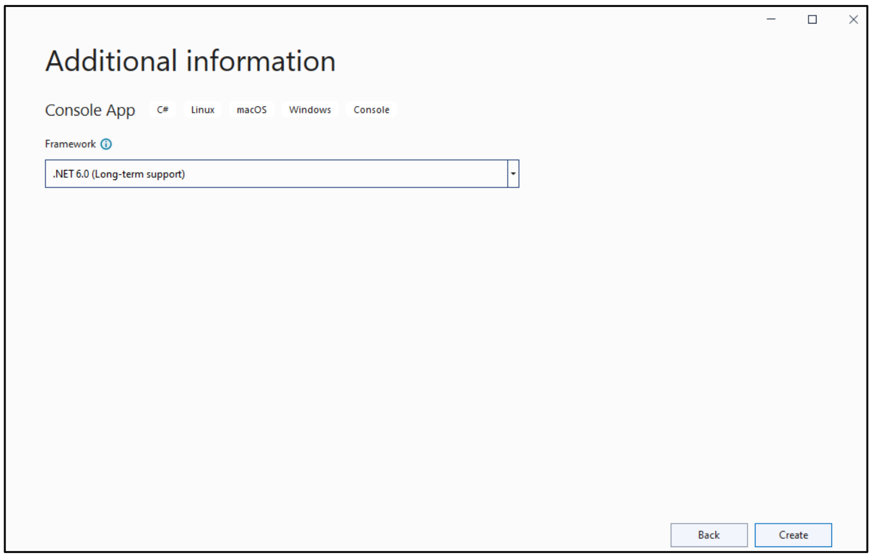
Project name: Exercise2

Location: C:\Courseware\QACS\Labs\02_Introduction_to_C#\

Solution name: Exercise2Solution

☐ Place solution and project in the same directory

6 Ensure **.NET 8.0 (Long-term support)** is selected in Additional information and click **Create**

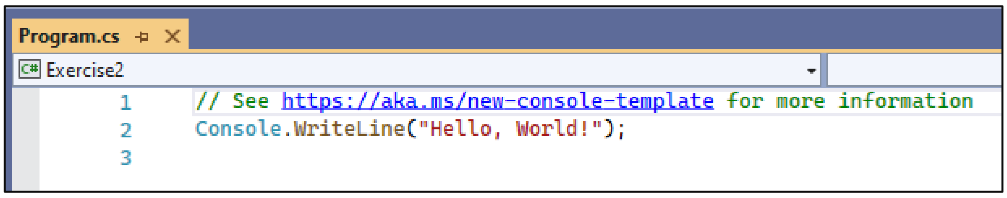


Additional information

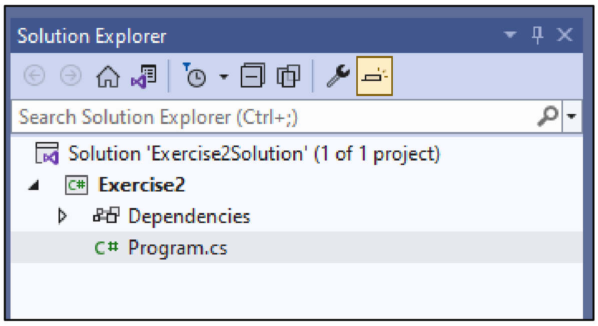
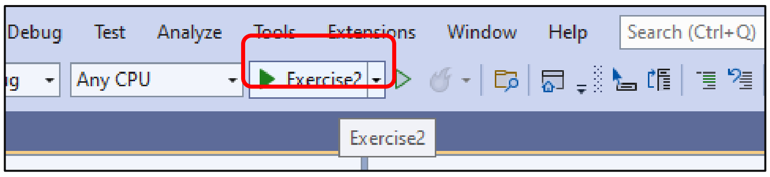
Console App · C# · Linux · macOS · Windows · Console

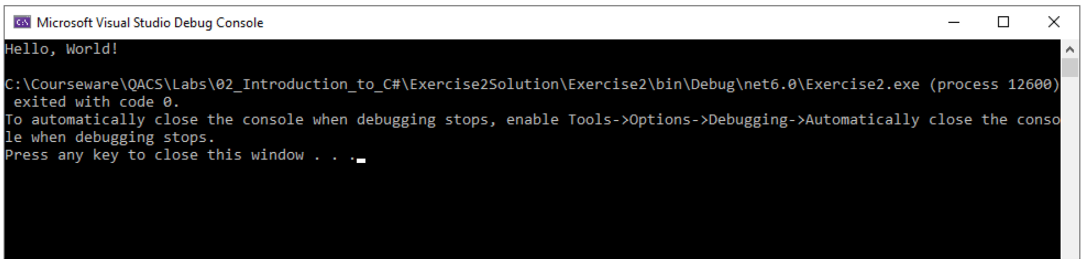
Framework: .NET 6.0 (Long-term support)

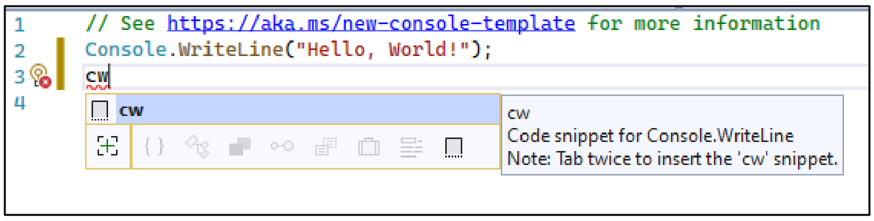
7 You will see a code editor window for a file called **Program.cs** containing the following code:

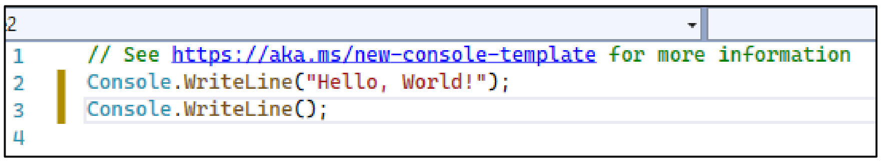


```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");

```

8	You can see the Solution Explorer window with **Program.cs** being tracked as the active file.  Solution Explorer Search Solution Explorer (Ctrl+;) Solution 'Exercise2Solution' (1 of 1 project) Exercise2 Dependencies Program.cs
9	From the toolbar, click the green arrow to **Start with Debugging**:  Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Any CPU Exercise2 Exercise2

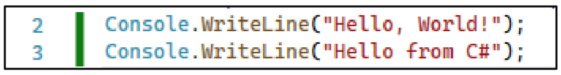
10	Observe the output of the program and press any key to close the console window.
	
	Microsoft Visual Studio Debug Console
	`Hello, World!`
	
	`C:\Courseware\QACS\Labs\02_Introduction_to_C#\Exercise2Solution\Exercise2\bin\Debug\net6.0\Exercise2.exe (process 12600)`
	` exited with code 0.`
	`To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso`
	`le when debugging stops.`
	`Press any key to close this window . . .`

11	Add a line of code at line 3 as follows. Type:
	
	cw
	
	Notice it is a recognised code snippet for **Console.WriteLine**:
	
	`1 // See https://aka.ms/new-console-template for more information`
	`2 Console.WriteLine("Hello, World!");`
	`3 cw`
	`4`
	
	cw
	Code snippet for Console.WriteLine
	Note: Tab twice to insert the 'cw' snippet.
	
	Tab twice to insert the code:



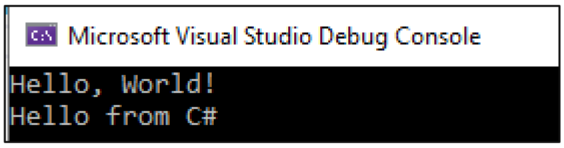
```
2
1  // See https://aka.ms/new-console-template for more information
2  Console.WriteLine("Hello, World!");
3  Console.WriteLine();
4
```

12 Add the text string **“Hello from C#”** as a parameter to the WriteLine method.



```
2  Console.WriteLine("Hello, World!");
3  Console.WriteLine("Hello from C#");
```

13 Run the program again, this time using the keyboard shortcut **F5**

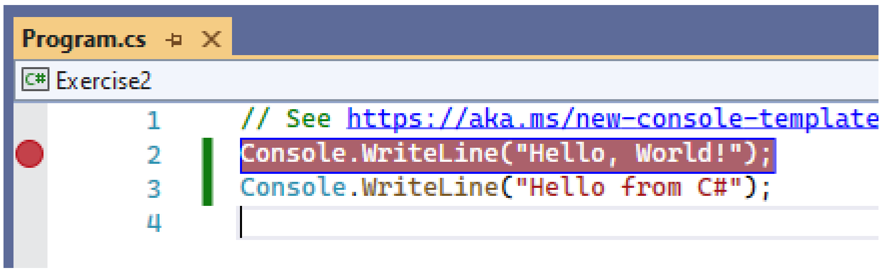


```
Hello, World!
Hello from C#
```

Press any key to quit the program.

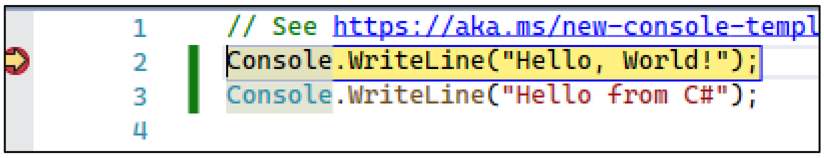
14 You will now do some very simple debugging.

Click in the margin well to the left of **line 2** to set a breakpoint:



```
1  // See https://aka.ms/new-console-template
2  Console.WriteLine("Hello, World!");
3  Console.WriteLine("Hello from C#");
4
```

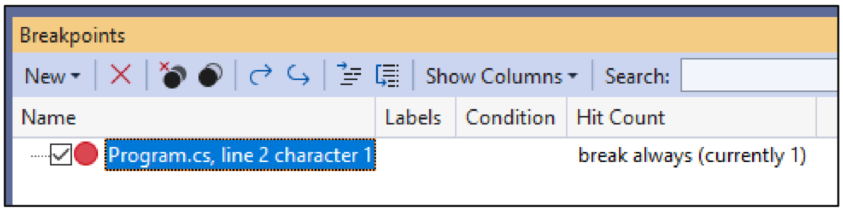
15 Debug the program with **F5** and notice how the application is now paused on your breakpoint:

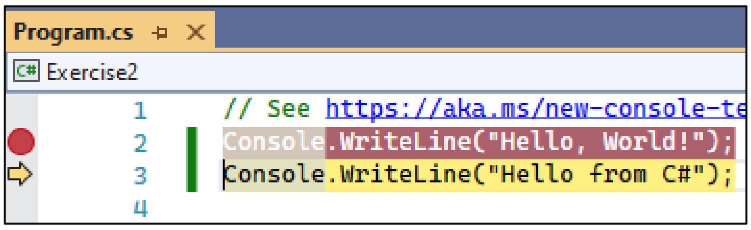
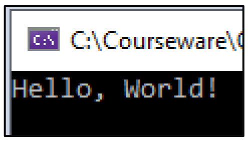
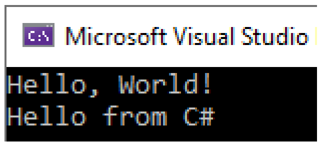


```
1  // See https://aka.ms/new-console-templ
2  Console.WriteLine("Hello, World!");
3  Console.WriteLine("Hello from C#");
4
```

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)”**  Breakpoints New ▾ Show Columns ▾ Search: Name | Labels | Condition | Hit Count ☑ ● Program.cs, line 2 character 1 | | | 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**.
	
	Program.cs
	Exercise2
	`1 // See https://aka.ms/new-console-te`
	`2 Console.WriteLine("Hello, World!");`
	`3 Console.WriteLine("Hello from C#");`
	`4`
	Look at the **Console** output window. You can see line 2 has run:
	
	C:\Courseware\
	`Hello, World!`
	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:
	
	Microsoft Visual Studio
	`Hello, World!`
	`Hello from C#`

If You Have Time – Using Git and GitHub

Use these instructions to familiarize yourself with GitHub (if you have not used GitHub previously)

Objective

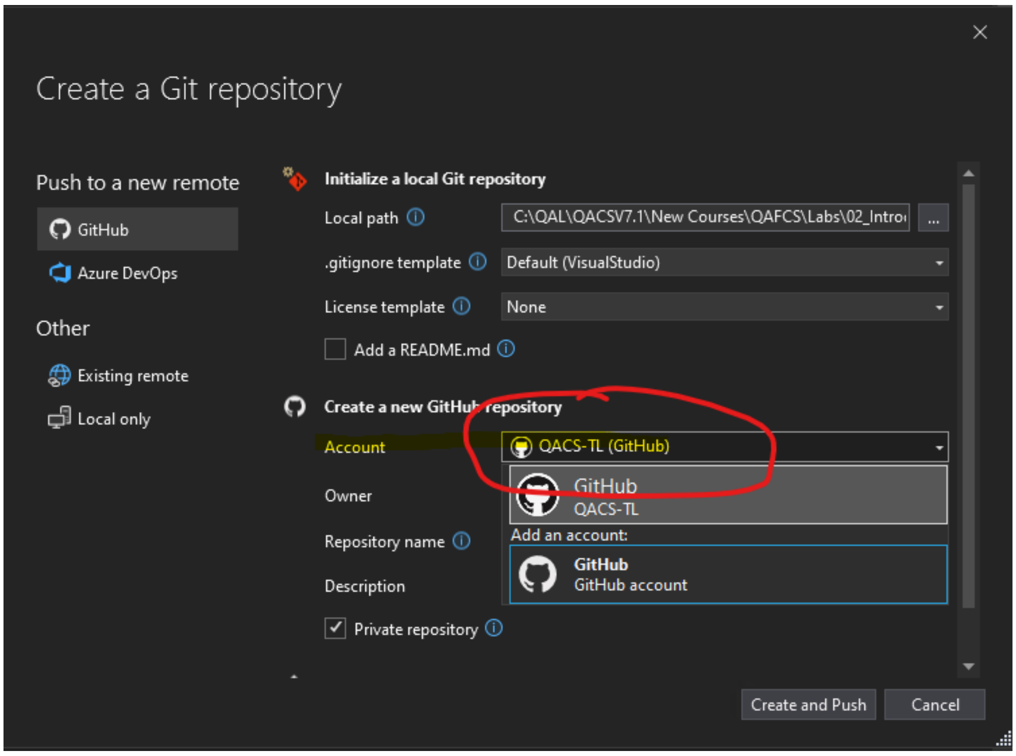
Gain an understanding of how GitHub can be used to work on projects in a collaborative environment. You are encouraged to use GitHub as a repository for **ALL** the code you work on as part of this course. Using it means you are less likely to be impacted on a virtual machine timing out at some point. This is especially true of LOD machines and less so if you are using GoToMyPC. However, **ALL** VM's will be reset at the course's conclusion so **backing things up is a really good idea**.

Requirements

Create a GitHub account (if necessary) and then follow a the "hello world intro to GitHub" on the GitHub portal. Then use Visual Studio in conjunction with GitHub to manage versioning.

Steps to Complete

1	Create your GitHub account https://docs.github.com/en/get-started/start-your-journey/creating-an-account-on-github
2	Create a repository and use it to merge changes into the main branch from a another https://docs.github.com/en/get-started/start-your-journey/hello-world
3	Using GitHub with Visual Studio a. Using Visual Studio Open the Solution called **Labs\02_Introduction_to_C#\Begin** b. Under the Git item on the main menu select Create Git Repository c. Sign in to GitHub using the account you used earlier.

4	

Create a Git repository

Push to a new remote
- GitHub
- Azure DevOps

Other
- Existing remote
- Local only

Initialize a local Git repository

Local path: C:\QAL\QACSV7.1\New Courses\QAFCS\Labs\02_Intro

.gitignore template: Default (VisualStudio)

License template: None

☐ Add a README.md

Create a new GitHub repository

Account: QACS-TL (GitHub)

GitHub
QACS-TL

Add an account:

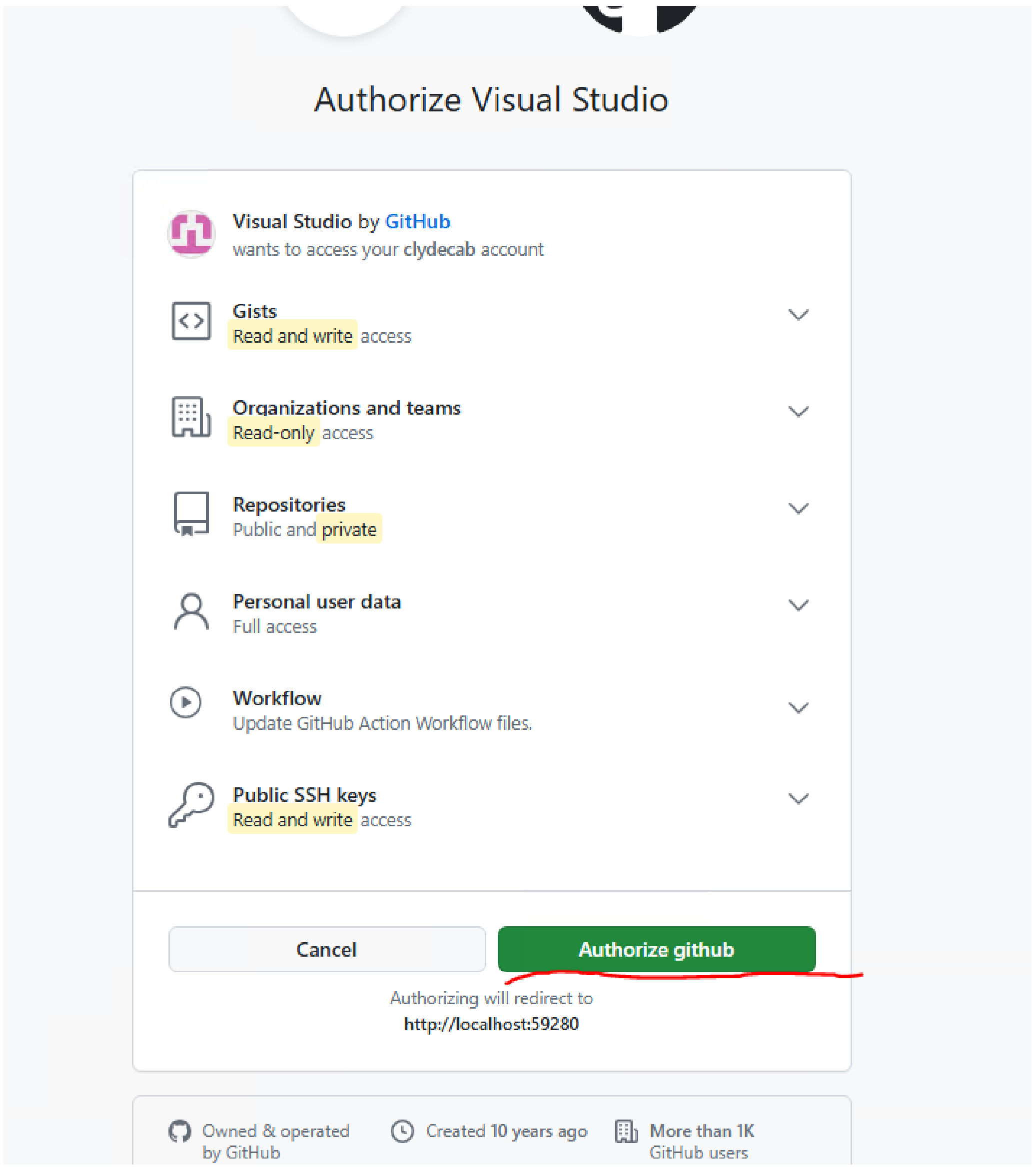
GitHub
GitHub account

Owner

Repository name

Description

☑ Private repository


Authorize Visual Studio

Visual Studio by GitHub
wants to access your clydecab account

Gists
Read and write access

Organizations and teams
Read-only access

Repositories
Public and private

Personal user data
Full access

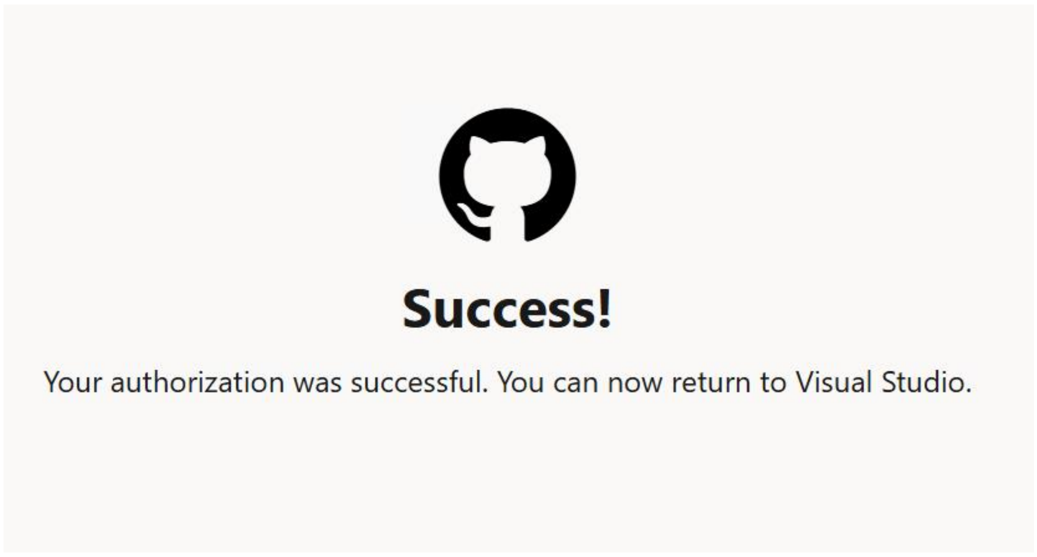
Workflow
Update GitHub Action Workflow files.

Public SSH keys
Read and write access

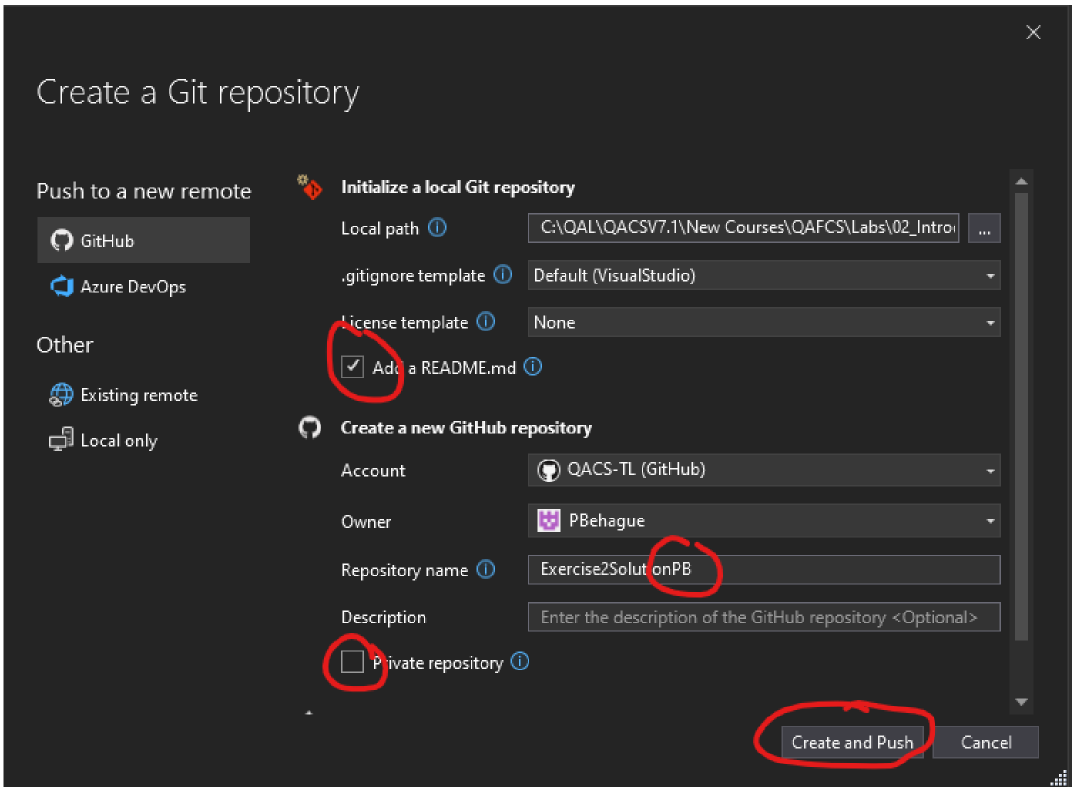
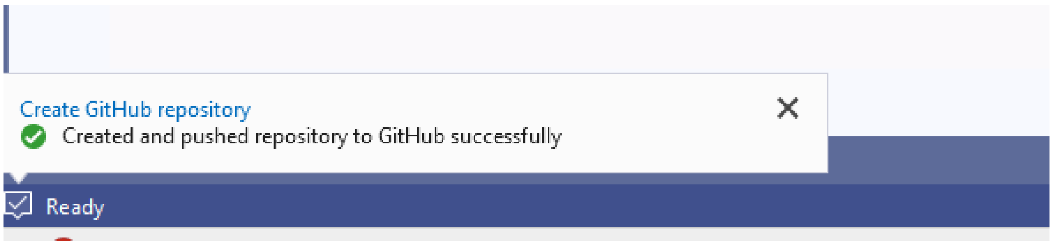
Cancel | Authorize github

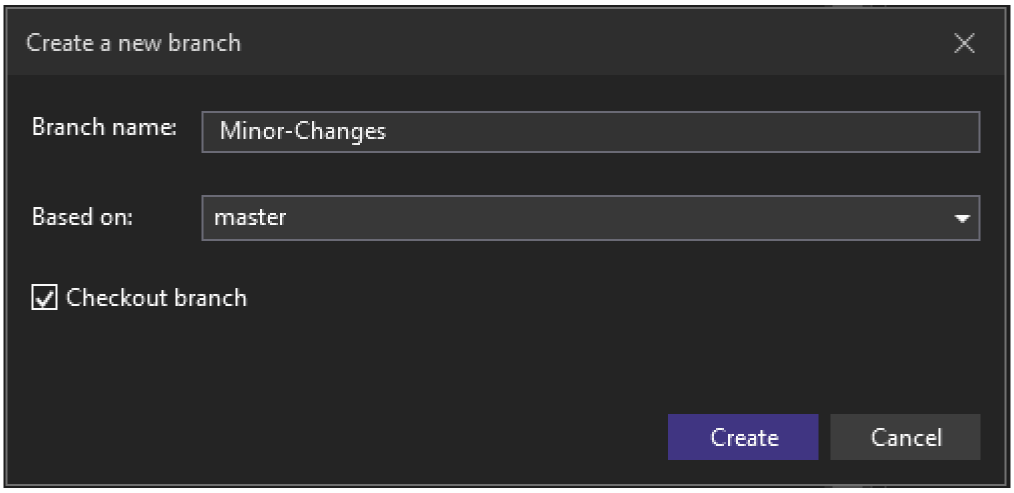
Authorizing will redirect to
http://localhost:59280

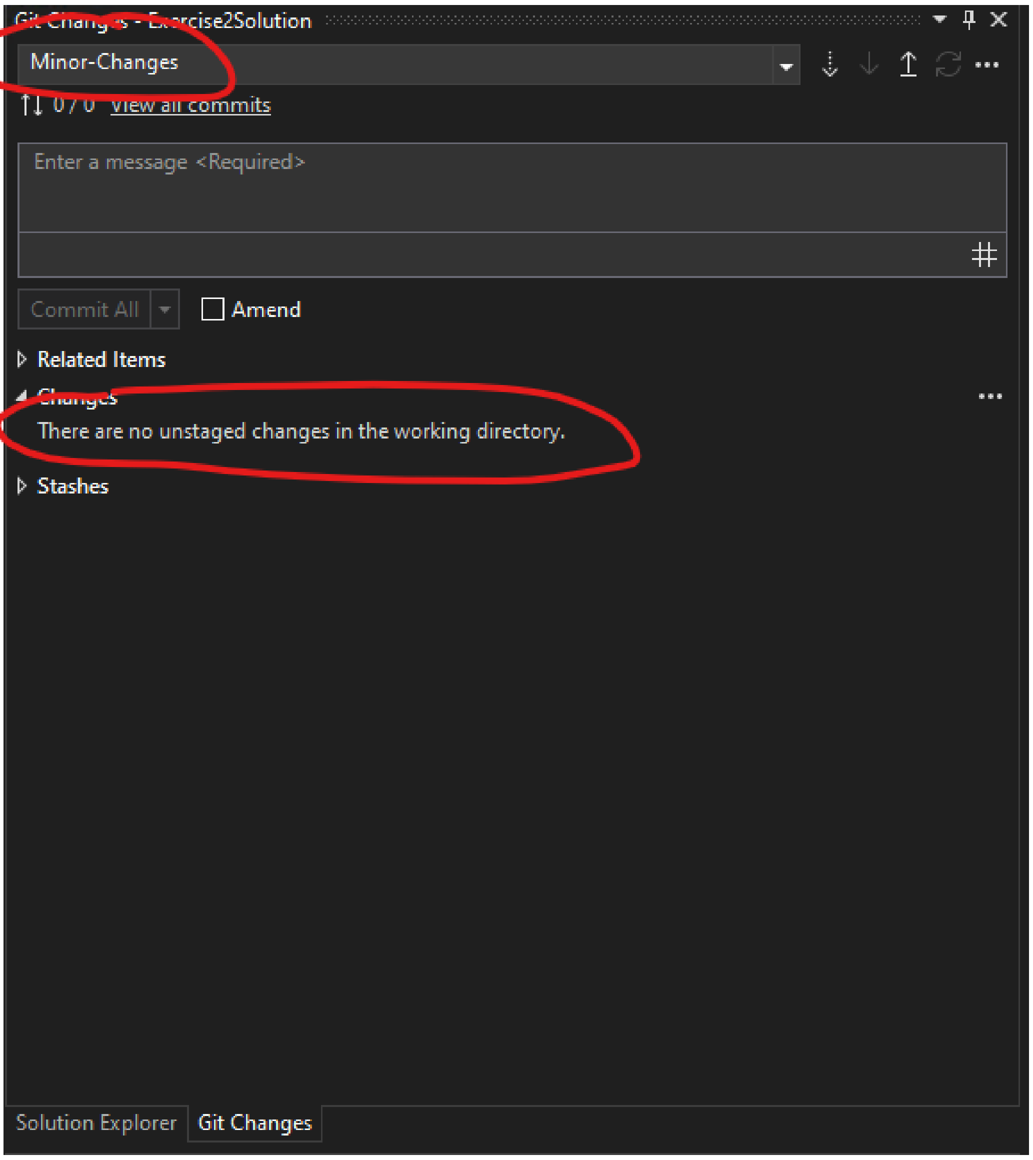
Owned & operated by GitHub | Created 10 years ago | More than 1K GitHub users


Success!

Your authorization was successful. You can now return to Visual Studio.

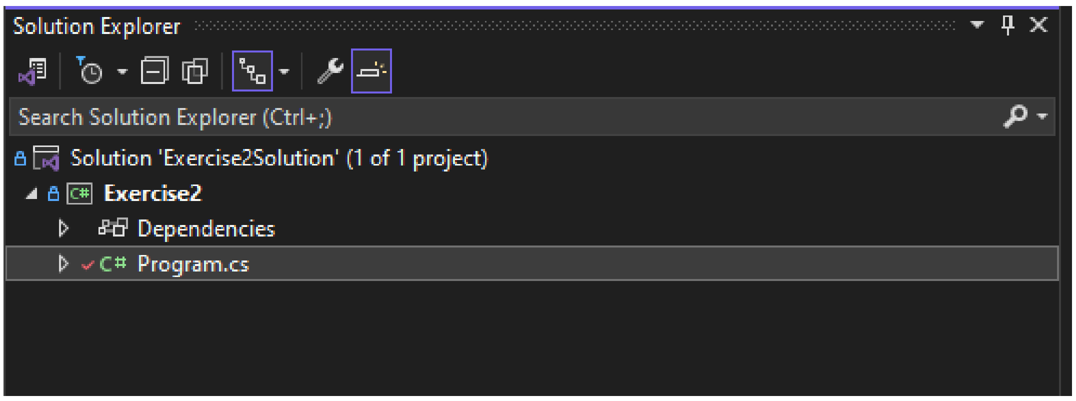
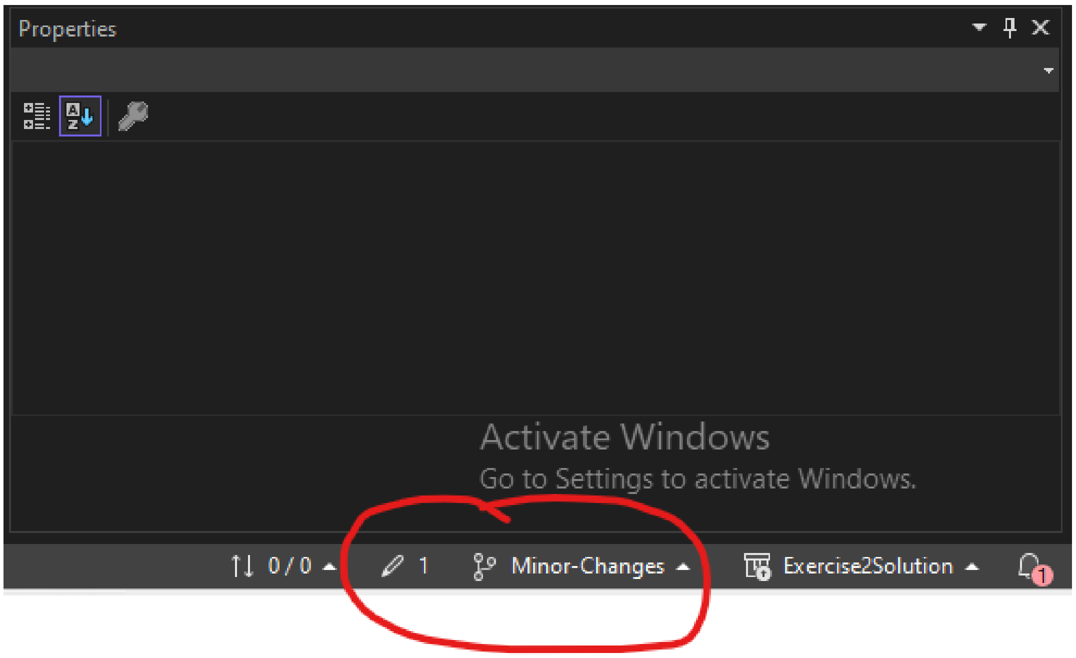
5	Add your initials to the end of the auto-generated repository name ("PB" in the example below), deselect the Private repository option and select the Add a README.md
6	 Create a Git repository Push to a new remote: GitHub, Azure DevOps Other: Existing remote, Local only Initialize a local Git repository Local path: C:\QAL\QACSV7.1\New Courses\QAFCS\Labs\02_Intro .gitignore template: Default (VisualStudio) License template: None ✓ Add a README.md Create a new GitHub repository Account: QACS-TL (GitHub) Owner: PBehague Repository name: Exercise2SolutionPB Description: Enter the description of the GitHub repository <Optional> ☐ Private repository Create and Push Cancel
7	Then click on the Create and Push button
8	Your project has now been uploaded to GitHub and you also have a local copy(clone) on the local file system.
9	 Create GitHub repository Created and pushed repository to GitHub successfully Ready

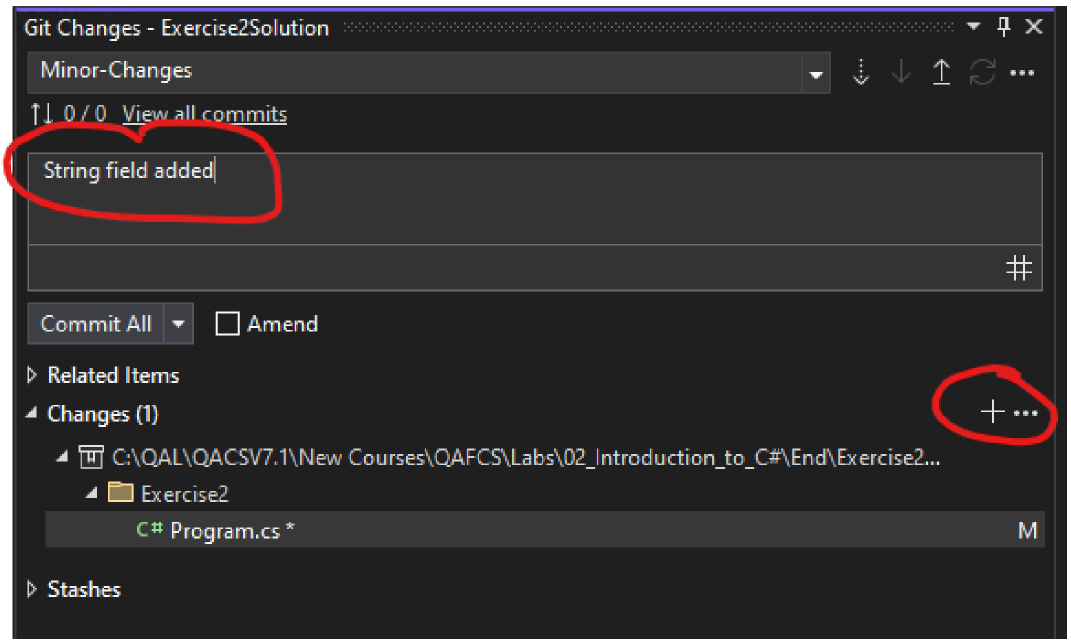
10	Now make a branch to support updates to the current project. Click on the Git item in the main menu of Visual Studio and select the **New Branch** Item. Name the branch **Minor-Changes** and confirm that the **Checkout branch** option is selected
11	 Create a new branch Branch name: Minor-Changes Based on: master ☑ Checkout branch
12	Click on create, and if prompted for further input, accept the defaults and continue.
13	The **Git Changes** window should show that there are currently no changes deployed to the branch named **Minor-Changes**

14	 Git Changes - Exercise2Solution Minor-Changes 0 / 0 View all commits Enter a message <Required> Commit All Amend Related Items Changes There are no unstaged changes in the working directory. Stashes Solution Explorer Git Changes
15	If not already done so, open the Program.cs file and declare a private field of type string named s1 and initialise this to have a default value of "test"

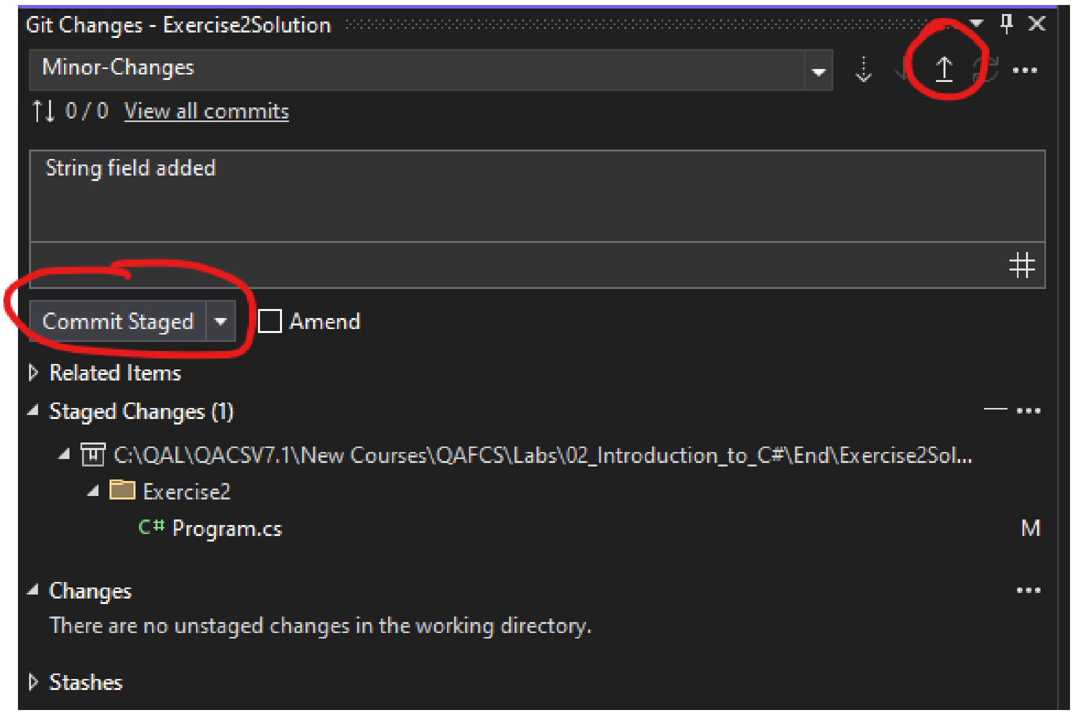
16	`// See https://aka.ms/new-console-template for more information`
	`Console.WriteLine("Hello, World!");`
	`Console.WriteLine("Hello from C#");`
	`String s1 = “test”;`

17	Notice that the modified file has a red tick associated with it in the Solution Explorer window

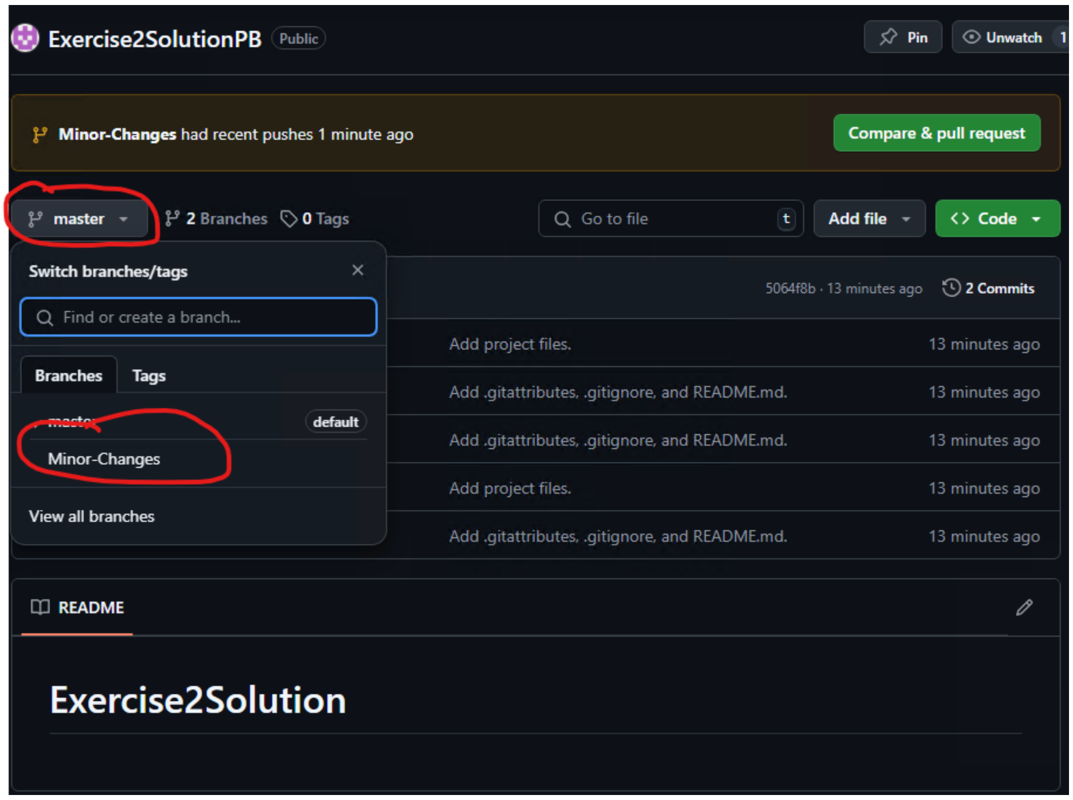
18	 Solution Explorer; Search Solution Explorer (Ctrl+;); Solution 'Exercise2Solution' (1 of 1 project); Exercise2; Dependencies; Program.cs
19	Open the Git Changes tab and **double click** on the Program.cs file. The diff view appears highlighting the change (Additional line) added to the source code
20	Additionally, the status bar at displayed at the bottom of the Visual Studio window displays 1 change to the file associated with the branch named Minor-Changes
21	 Properties; Activate Windows; Go to Settings to activate Windows.; 0 / 0; 1; Minor-Changes; Exercise2Solution

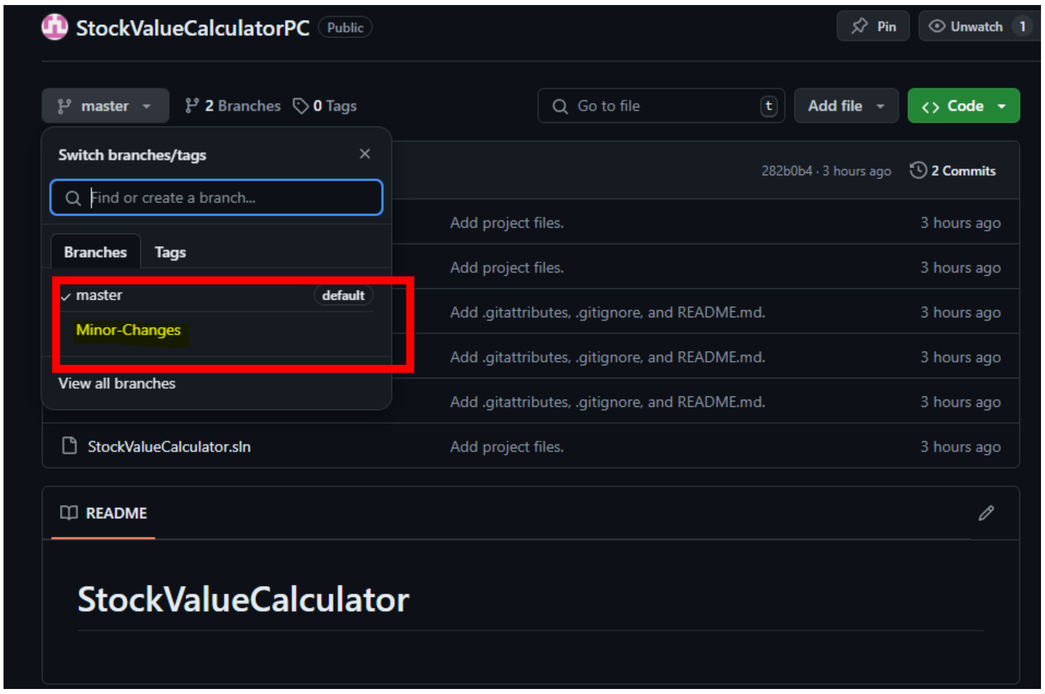
22	Now we will commit the changes to the GitHub Repo. In Solution Explorer right click on the Program.cs file select the Git menu item and then select the Commit or Stash item. Enter a description in the input text field, such as “string field added”. and click on the + Stage All button 

Git Changes - Exercise2Solution
Minor-Changes
0 / 0 View all commits
String field added
Commit All
Amend
Related Items
Changes (1)
C:\QAL\QACSV7.1\New Courses\QAFCS\Labs\02_Introduction_to_C#\End\Exercise2...
Exercise2
Program.cs * M
Stashes

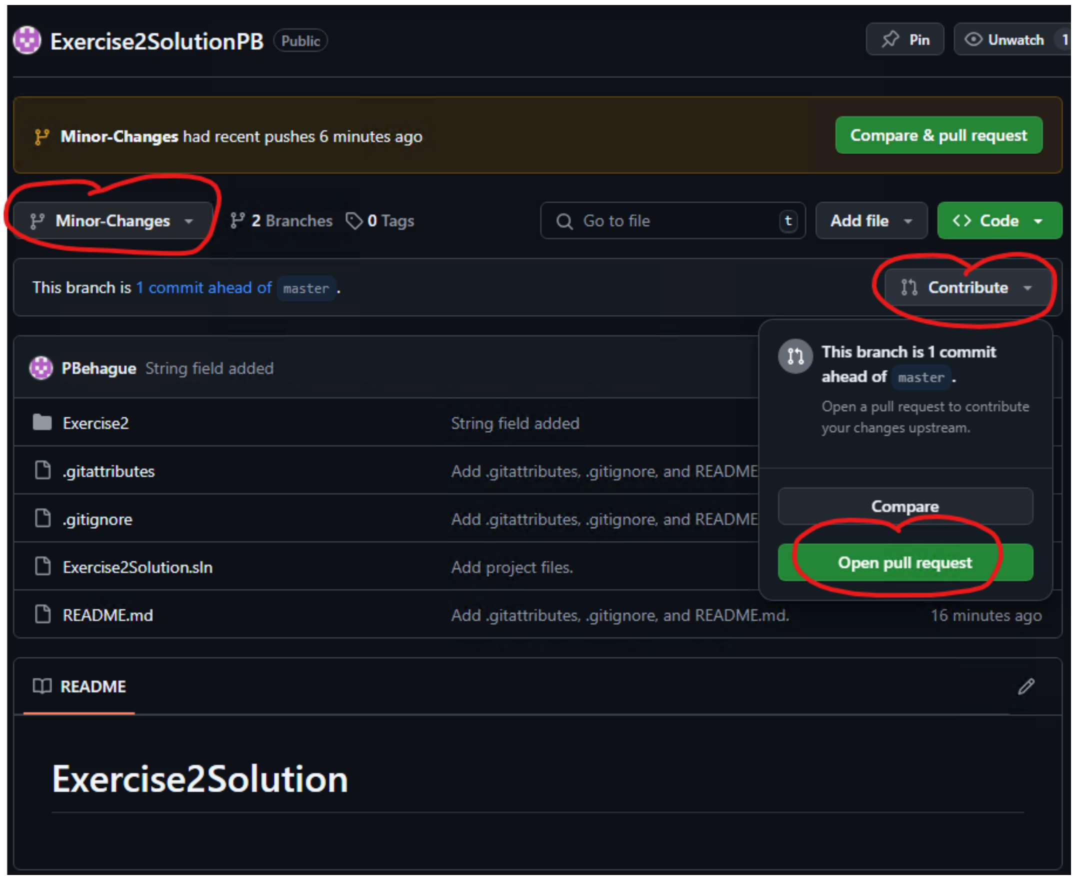
23	The UI should now display as shown below. Click on the **Commit Staged** button. Then click on the **up arrow** to upload the changes to the Minor-Changes Branch  Git Changes - Exercise2Solution Minor-Changes 0 / 0 View all commits String field added Commit Staged Amend Related Items Staged Changes (1) C:\QAL\QACSV7.1\New Courses\QAFCS\Labs\02_Introduction_to_C#\End\Exercise2Sol... Exercise2 Program.cs M Changes There are no unstaged changes in the working directory. Stashes
24	Check that the changes have been pushed to the repo by using a browser to navigate to https://github.com/ and select the Exercise2SolutionXX repo.

25 Navigate to the Minor-Changes branch using the drop down menu

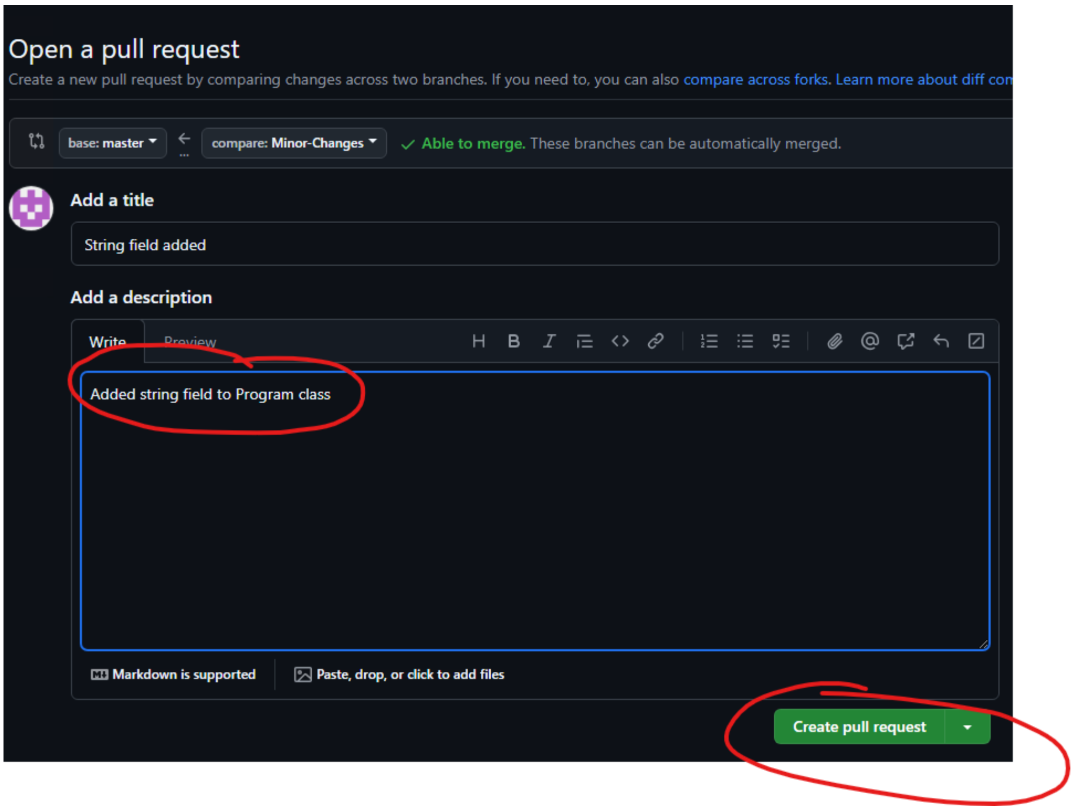




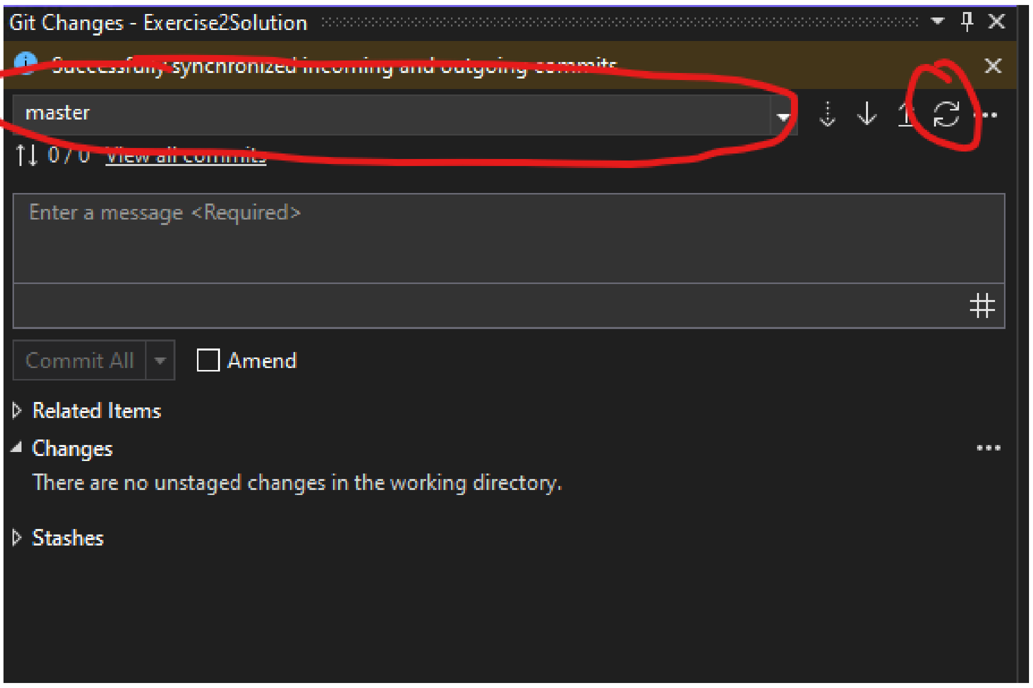
26 Open the Exercise2 folder that contains the Program.cs file. You should see the modifications have been pushed to the branch

	named Minor-Changes. The master branch should still have the original version. We will now merge the changes recorded in the Minor-Changes branch with the master branch
27	In the GitHub Portal ensure the Minor-Changes branch is selected in the dropdown. Then from the Contribute menu click Open Pull Request 

Exercise2SolutionPB
Public
Pin
Unwatch
Minor-Changes had recent pushes 6 minutes ago
Compare & pull request
Minor-Changes
2 Branches
0 Tags
Go to file
Add file
Code
This branch is 1 commit ahead of master.
Contribute
This branch is 1 commit ahead of master.
Open a pull request to contribute your changes upstream.
Compare
Open pull request
PBehague String field added
Exercise2
String field added
.gitattributes
Add .gitattributes, .gitignore, and README
.gitignore
Add .gitattributes, .gitignore, and README
Exercise2Solution.sln
Add project files.
README.md
Add .gitattributes, .gitignore, and README.md.
16 minutes ago
README
Exercise2Solution

28	In the form that appears add a description e.g. Added String Field to Program class and then Click on the **Create Pull Request** button.  Open a pull request Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about diff com base: master ← compare: Minor-Changes ✓ Able to merge. These branches can be automatically merged. Add a title String field added Add a description Added string field to Program class Write Preview Markdown is supported Paste, drop, or click to add files Create pull request
29	On the next page click on the **Merge pull request** and then **Confirm merge** buttons in order to commit the changes with the master branch

30 Go back to Visual Studio select the **master** branch from the dropdown in the **Git Changes** tab and click on the **Sync (Push and Pull**) button



Git Changes - Exercise2Solution

Successfully synchronized incoming and outgoing commits

master

↑↓ 0 / 0 View all commits

Enter a message <Required>

Commit All ☐ Amend

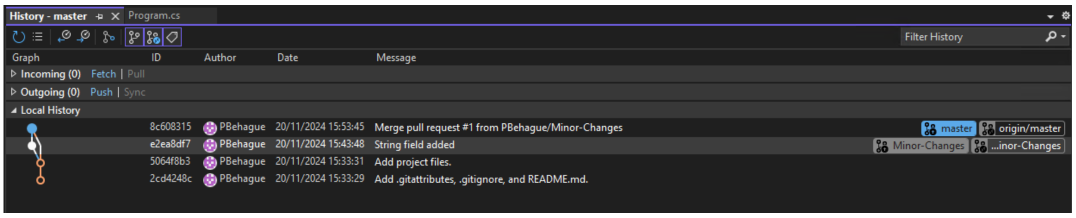
Related Items

Changes

There are no unstaged changes in the working directory.

Stashes

31 Go to Visual Studio's Git menu and select the View Branch History option. The graph displaying the updates to the master branch will now include the change you recorded the Minor-Changes branch.



History - master | Program.cs

Incoming (0) Fetch | Pull

Outgoing (0) Push | Sync

Local History

ID	Author	Date	Message	
8c608315	PBehague	20/11/2024 15:53:45	Merge pull request #1 from PBehague/Minor-Changes	master origin/master
e2ea8df7	PBehague	20/11/2024 15:43:48	String field added	Minor-Changes ...inor-Changes
5064f8b3	PBehague	20/11/2024 15:33:31	Add project files.	
2cd4248c	PBehague	20/11/2024 15:33:29	Add .gitattributes, .gitignore, and README.md.	

32 Feel free to use GitHub when working on other exercises in the labs that follow this one.

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.

Functions

The objectives of this practical session are twofold.

Firstly, to practice writing some simple method calls on some pre-supplied methods that have a varying number of parameters from 0-3, some of which return a value.

Secondly, to get some practice reading input that has been typed in at the Console in response to a prompt and processing that data further including converting it to a different type of data.

In this practical, you will open and code inside a couple of pre-supplied Console applications. In the first one you will follow the instructions that are inside the Program class' Main method that simply need to be followed accurately.

In the second, you will follow step by step instructions in this document.

Part 1 – Calling pre-supplied methods

1	Open the existing pre-supplied Visual Studio solution (.sln) called '**MethodCallingPractice**'. It can be found in folder **labs\04 Functions\ Begin\MethodCallingPractice**.
2	Open file **Program.cs** that contains the code of a class called '**Program'**. Expand the **Main** method and follow the 'tasks' reading carefully. Make use of the Intellisense trying to type as little as possible.
3	There is a small '**whenYouHaveTime()**' method which will enable you to find out 5 interesting facts about Chile, Russia, Mongolia, Finland & Zimbabwe.

Part 2 – Interacting with the Console, performing data conversions and authoring a helper routine

1	Open the existing pre-supplied Visual Studio solution (.sln) called '**DoingLunch**'. It can be found in folder **labs\04 Functions\Begin\DoingLunch**.
2	Open file **Program.cs** that contains the code of a class called '**Program'**. Expand the **Main** method. The scenario is going to mimic a serving line at a lunch hall in that we are going to prompt the user to answer certain questions. What would you like as a main dish? Then how many Roast Potatoes? How many Brussel Sprouts? Then display what their lunch is.
3	You will note that there are currently three prompts pre-coded with the lower down ones commented out. The first prompt is: `Console.WriteLine("What main dish would you like?");`

4	Add 2 lines of code.
	One that declares a string called mainCourse.
	Another that accepts an entry from the keyboard and stores it into the mainCourse variable. Use Console.ReadLine().
	
	`string mainCourse;`
	`mainCourse = Console.ReadLine();`

5	5Add code at the very bottom of the method that prints out a message to the user along the lines of: "Hello, your lunch is xxxxx" Where xxxx is the mainCourse the user entered.

	`Console.WriteLine ("Hello, your lunch is " + mainCourse);`
6	However, in view of the fact that this message is going to get significantly larger let's update it immediately using the 'special' version of **Console.WriteLine()** `Console.WriteLine ("Hello, your lunch is {0}", mainCourse);`
7	**Catching Numerical Information** If we want a user to input numerical information we have a slight problem in that **Console.ReadLine()** treats all input as a string. Consequently, we need to catch the string in a similar way to the previous steps and then convert it to a numerical value. Let's get the user to enter the number of roast potatoes they want.
8	Uncomment the "How many roast potatoes?" prompt. Immediately after this statement declare a string variable called sRoasties and store the inputted value into it.

9	We could clearly display this 'string' value in the message but later we will want to do arithmetic on it so we need to convert the string type into an int.
	So declare an int variable called roastCount and on the next line use the Convert.ToInt32() method in the following way:
	
	`int roastCount;`
	`roastCount = Convert.ToInt32(sRoasties);`

10	Uncomment the "How many roast potatoes?" prompt. Immediately after this statement declare a string variable called sRoasties and store the inputted value into it.
11	Repeat the previous three steps to retrieve and display information about the number of brussel sprouts the user would like.

12	The message should now look like “Hello, your lunch is xxxxx with yy roast potatoes and zz brussel sprouts.”

Creating a helper routine

Have you noticed the duplication of effort in our code?

The logic behind the requests for roasts and Brussel sprouts along with the associated numeric conversions is pretty much the same. This is the perfect opportunity to re-factor the code and create a method that does the job once but that can be called anytime we need numeric data.

The first thing we need to do is determine exactly what is common and what is unique.
Observe that when we ask for data we display unique messages that ask the user to enter either one thing or the other but the code that converts this “value” to an actual number is essentially the same.

13	Create a static method called GetInteger that takes a string called message as a parameter. The method’s return type should be int. Do this immediately after the end of the Main method. The parameter (if and when the method gets called) will contain the message that will be displayed to the user telling them what to enter. For example “How many roast potatoes?” This return value will be the number the user entered.
14	Add code to this method that displays the message parameter on the console, accepts a string input, converts this string to an integer and returns the result of this conversion to the caller. Words like ‘roast’ and ‘brussels’ have no meaning in this method, it has to work for all questions/answers. Your code may look like the following:

	`static int GetInteger(string message) {`
	`  Console.WriteLine(???);`
	`  ?? ?? = Console.ReadLine();`
	`  int ?? = Convert.ToInt32(??);`
	`  return ??;`
	`}`

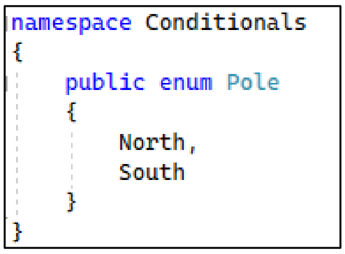
15	Can you see that this code is essentially the same as the code that retrieves the 'number of roasts' and 'number of brussels'? It's just that instead of "hard coding" the "How many roasts?" or "How many brussels?" we will pass them as parameters to the method and it will display them.
16	Replace the 3 lines of code in the main method that prompt for "how many roasts?" that catch the reply, and convert it to an integer with a one line call to the GetInteger method. You need to work this out for yourself:
17	Now in a similar way, replace the 3 lines of code that prompt for "how many brussels" and convert it to an integer with another one line call to the GetInteger method.
18	Build and test your application.
19	Can you see though that the GetInteger method only needs to be 2 lines long. The 2nd line will start with 'return'! Re-factor it now.
20	After receiving the message of what his lunch is he sits down to eat it. Unfortunately, brussel sprouts were the only vegetable choice which is why he took some. But he remembered he really does not like brussel sprouts at all but unwilling to see food wasted – and he is hungry - he comes up with a cunning plan. If he cuts the roast potatoes into quarters. He could then perhaps 'force down' one brussel sprout with each quarter of roast potato. Fortunately, his

	lunch partner has a spare whole roast potato which he gratefully ‘grabs’.
21	19.Write some code to work out how many spare quarters of roast potato he will have **after** receiving the extra one and cutting them into quarters **and** using up 1 per brussel sprout. Display an appropriate message. As the answer cannot be negative (in the situation where he has, say, 3 potatoes but 18 sprouts) use Math.Max to ensure that 0 is displayed if he runs out of quarter-roasts before he eats all the sprouts. Place this code at the bottom of the **Main()** method. Test it fully. Now reduce it to a single statement.

Conditionals

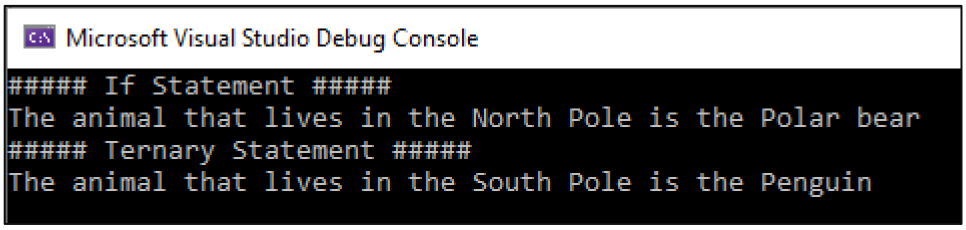
The objective of this exercise is to consolidate your understanding of conditional statements, including the if statement, the ternary statement, switch statements, and switch expressions.

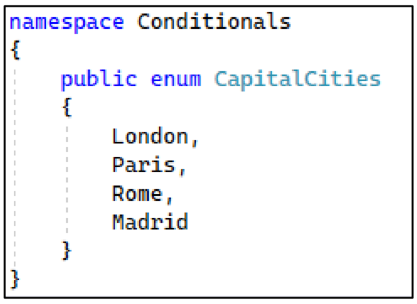
1	Create a new **console** project called **Conditionals** in: **C:\Courseware\QACS\Labs\04_Conditionals\Begin\**
2	Delete the contents of **Program.cs.**

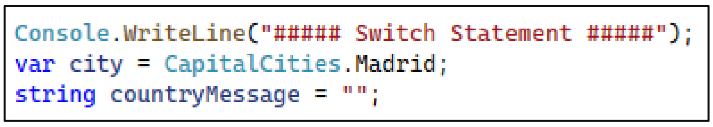
3	Add an **enum** to the project in a file called **Pole.cs.**
	
	
	`namespace Conditionals`
	`{`
	`    public enum Pole`
	`    {`
	`        North,`
	`        South`
	`    }`
	`}`

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.

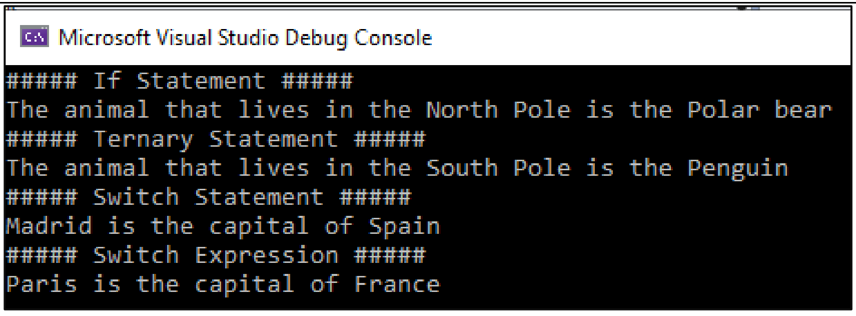
5	Output a message to the console: **Console.WriteLine($"The animal that lives in the {pole} Pole is the {animal}");** Run the app and confirm the logic works as expected.
6	Now assign the value of **Pole.South** to your **pole** variable and perform the same conditional test using the **ternary statement**.

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:
	
	
	Microsoft Visual Studio Debug Console
	`##### If Statement #####`
	`The animal that lives in the North Pole is the Polar bear`
	`##### Ternary Statement #####`
	`The animal that lives in the South Pole is the Penguin`

8	You will now practise with *switch statements* and *switch expressions*.
	
	Add an **enum** to the project called **CapitalCities**:
	
	
	`namespace Conditionals`
	`{`
	`    public enum CapitalCities`
	`    {`
	`        London,`
	`        Paris,`
	`        Rome,`
	`        Madrid`
	`    }`
	`}`

9	In **Program.cs**, declare and initialise the following variables:
	
	`Console.WriteLine("##### Switch Statement #####");`
	`var city = CapitalCities.Madrid;`
	`string countryMessage = "";`

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:

	 Microsoft Visual Studio Debug Console
	`##### If Statement #####`
	`The animal that lives in the North Pole is the Polar bear`
	`##### Ternary Statement #####`
	`The animal that lives in the South Pole is the Penguin`
	`##### Switch Statement #####`
	`Madrid is the capital of Spain`
	`##### Switch Expression #####`
	`Paris is the capital of France`

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.

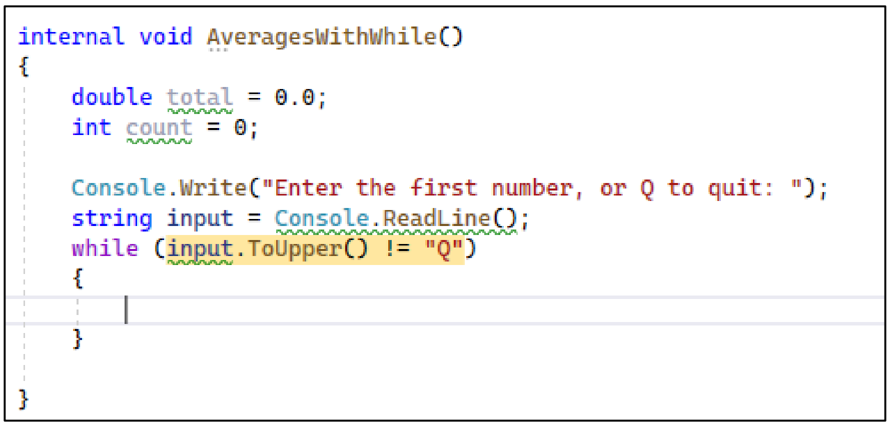
Averages

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**.

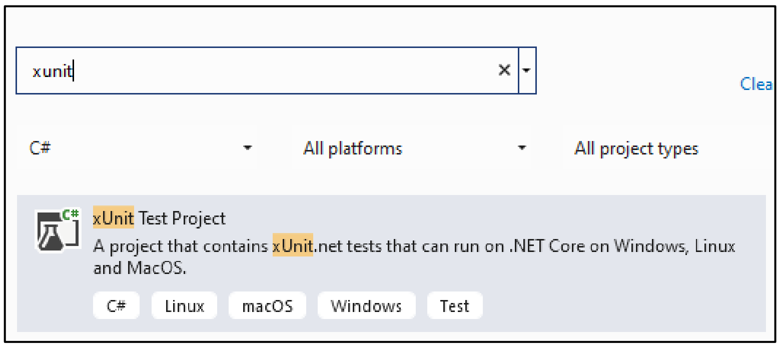
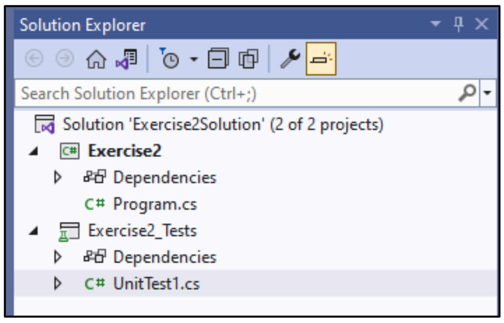
3	Create an *array of strings* called **muppets** to store the following values: ‘Kermit the Frog’, ‘Miss Piggy’, ‘Fozzie Bear’, ‘Gonzo’, ‘Rowlf the Dog’, ‘Scooter’, ‘Animal’, ‘Rizzo the Rat’, ‘Pepe the Prawn’, ‘Walter’, ‘Clifford’
4	Use a **foreach** loop to write the strings to the console.
5	Create 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.

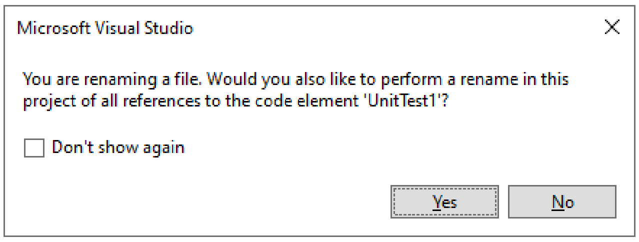
	Firstly, use a loop to display all the strings on one line separated by commas. Then, sort the list and re-display the strings to confirm they are sorted alphabetically.
12	You will now use different operators to extract specific strings from the list. Use an *indexer*, the *index from end* and *range* 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.

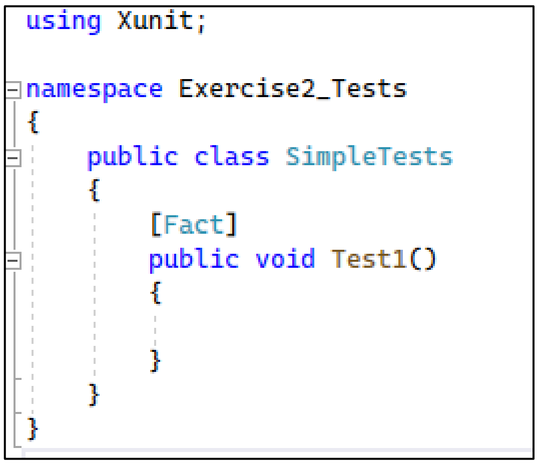
	Within each loop, output a string containing the iterated item.
17	A suggested solution is provided in the **End** folder for your reference.

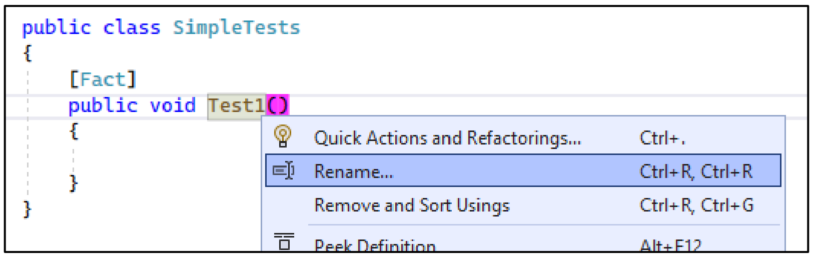
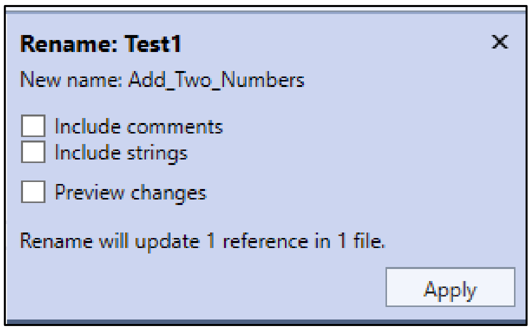
Unit Testing

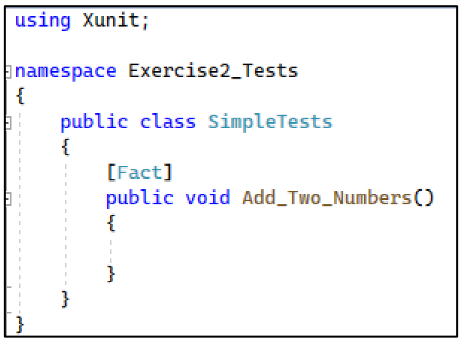
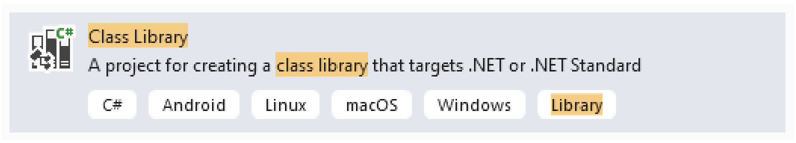
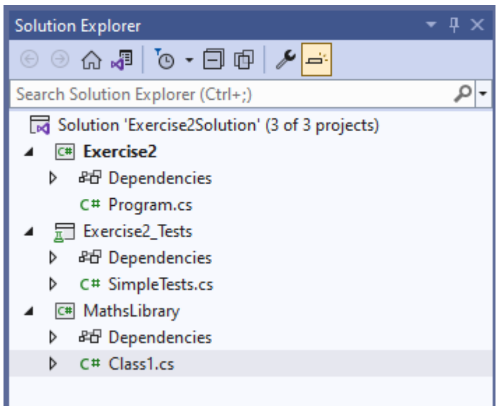
The objective of this exercise is to consolidate your understanding of Unit Testing in C#.

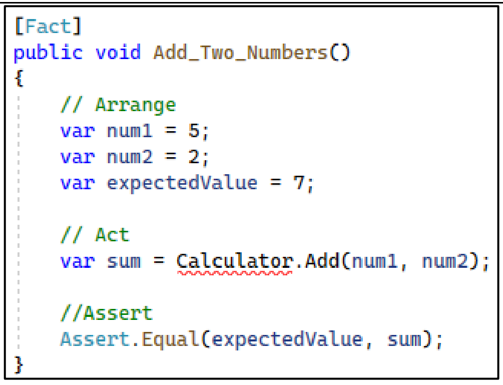
1	Create a new project in Visual Studio called **Exercise7Tests** in the Labs\07UnitTests\Begin folder. In the search box type **xunit**.  xunit C# All platforms All project types xUnit Test Project A project that contains xUnit.net tests that can run on .NET Core on Windows, Linux and MacOS. C# Linux macOS Windows Test Select **xUnit Test Project** and click **Next**. Name the project **Exercise7_Tests** and click **Next**. Ensure .**NET 8.0 (Long-term support)** is selected and click **Create**.
2	Your **Solution Explorer** window now contains one solution with two projects:  Solution Explorer Search Solution Explorer (Ctrl+;) Solution 'Exercise2Solution' (2 of 2 projects) Exercise2 Dependencies Program.cs Exercise2_Tests Dependencies UnitTest1.cs

3	Rename **UnitTest1.cs** (by right-clicking on the file name) to **SimpleTests.cs** and select **YES** when the following prompt displays:  Microsoft Visual Studio You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'UnitTest1'? ☐ Don't show again Yes No

4	Your **SimpleTests.cs** file contains the following starter code:  `using Xunit;`
	`namespace Exercise2_Tests`
	`{`
	`    public class SimpleTests`
	`    {`
	`        [Fact]`
	`        public void Test1()`
	`        {`
	`        }`
	`    }`
	`}`

5	Rename **Test1** to **Add_Two_Numbers**:  `public class SimpleTests`
	`{`
	`    [Fact]`
	`    public void Test1()`
	`    {`
	`    }`
	`}`
	Quick Actions and Refactorings... Ctrl+.
	Rename... Ctrl+R, Ctrl+R
	Remove and Sort Usings Ctrl+R, Ctrl+G
	Peek Definition Alt+F12
	 **Rename: Test1**
	New name: Add_Two_Numbers
	☐ Include comments
	☐ Include strings
	☐ Preview changes
	Rename will update 1 reference in 1 file.
	Apply

	Click **Apply**.  `using Xunit;` `namespace Exercise2_Tests` `{` `public class SimpleTests` `{` `[Fact]` `public void Add_Two_Numbers()` `{` `}` `}` `}`
6	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**.  Class Library A project for creating a class library that targets .NET or .NET Standard C# Android Linux macOS Windows Library Solution Explorer should now look as follows:  Solution Explorer Search Solution Explorer (Ctrl+;) Solution 'Exercise2Solution' (3 of 3 projects) Exercise2 Dependencies Program.cs Exercise2_Tests Dependencies SimpleTests.cs MathsLibrary Dependencies Class1.cs
7	Delete the file **Class1.cs**.
8	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);
}
```

9 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.

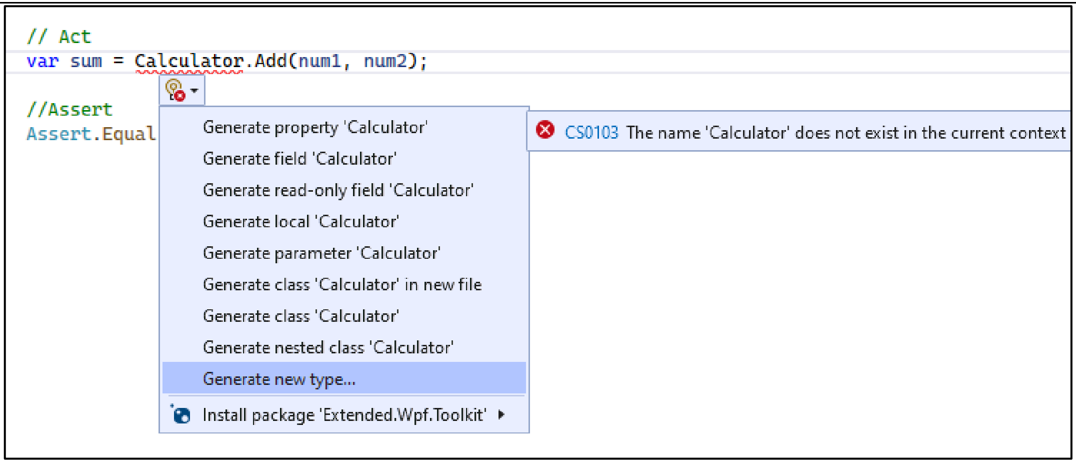
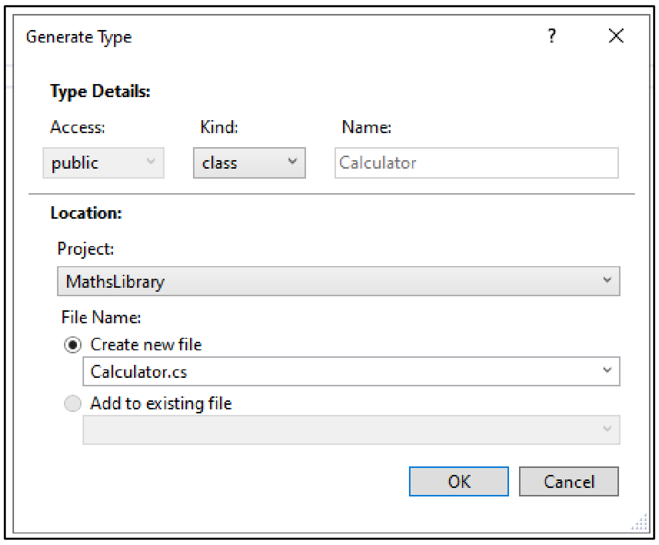
10 The arrange phase creates three variables: **num1**, **num2**, and **expectedValue**.

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…**'
11	In the dialog box, ensure a **public class** will be created and change the project to **MathsLibrary** and **Create new file**:  Click **OK**.
12	Visual Studio has created a new class (a kind of type) called **Calculator** in **MathsLibrary**:

```
// Act
var sum = Calculator.Add(num1, num2);

//Assert
Assert.Equal
```

Generate property 'Calculator'
Generate field 'Calculator'
Generate read-only field 'Calculator'
Generate local 'Calculator'
Generate parameter 'Calculator'
Generate class 'Calculator' in new file
Generate class 'Calculator'
Generate nested class 'Calculator'
Generate new type...
Install package 'Extended.Wpf.Toolkit'

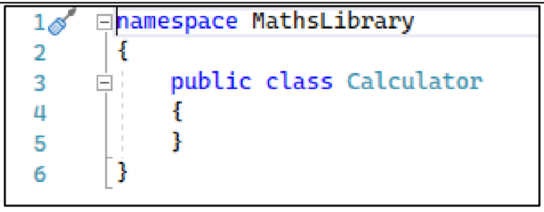
CS0103 The name 'Calculator' does not exist in the current context

Generate Type

Type Details:
Access: public
Kind: class
Name: Calculator

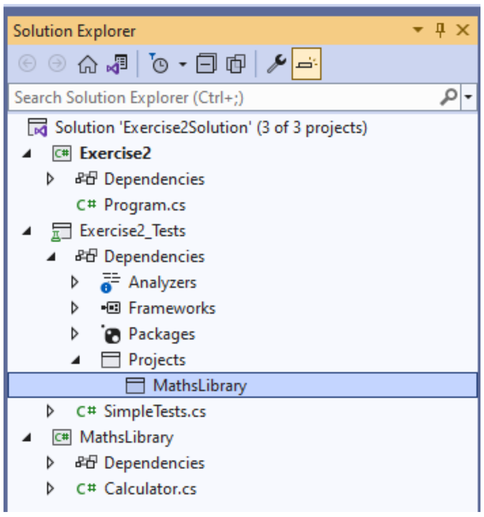
Location:
Project: MathsLibrary
File Name:
(●) Create new file: Calculator.cs
( ) Add to existing file

<table>
<tr><td></td><td>



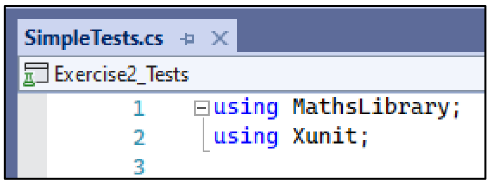
```
namespace MathsLibrary
{
    public class Calculator
    {
    }
}
```

Visual Studio has also added a *reference* to the **MathsLibrary** project:



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Exercise2Solution' (3 of 3 projects)
  Exercise2
    Dependencies
    Program.cs
  Exercise2_Tests
    Dependencies
      Analyzers
      Frameworks
      Packages
      Projects
        MathsLibrary
    SimpleTests.cs
  MathsLibrary
    Dependencies
    Calculator.cs
```

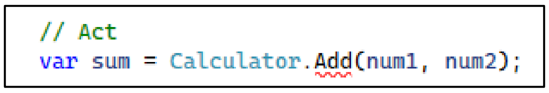
It has also imported the **MathsLibrary** *namespace* into your test project:



```
SimpleTests.cs
Exercise2_Tests
using MathsLibrary;
using Xunit;
```

</td></tr>
<tr><td>13</td><td>

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**:

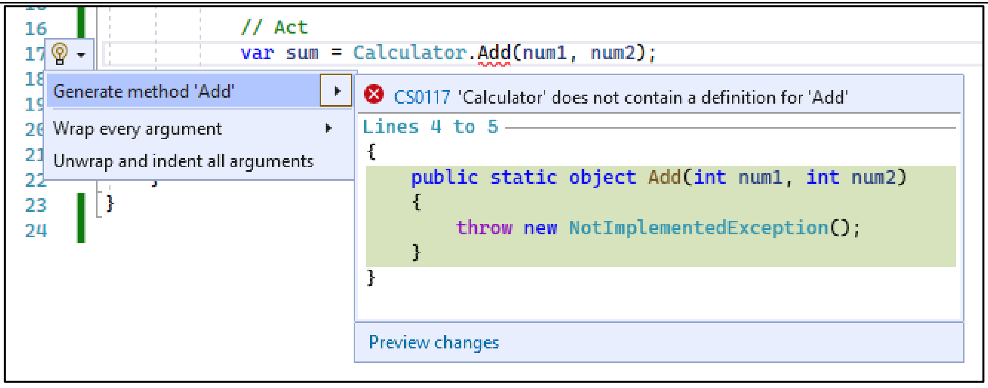


```
// Act
var sum = Calculator.Add(num1, num2);
```

</td></tr>
<tr><td>14</td><td>

Use **Ctrl+dot** on **Add** to generate the method:

</td></tr>
</table>



```
            // Act
            var sum = Calculator.Add(num1, num2);
```

CS0117 'Calculator' does not contain a definition for 'Add'

```
Lines 4 to 5
{
    public static object Add(int num1, int num2)
    {
        throw new NotImplementedException();
    }
}
```

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

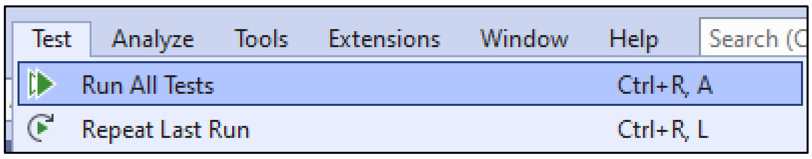
```
public class Calculator
{
    public static object Add(int num1, int num2)
    {
        throw new NotImplementedException();
    }
}
```

16 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();
}
```

17 You will run the test and observe the outcome.

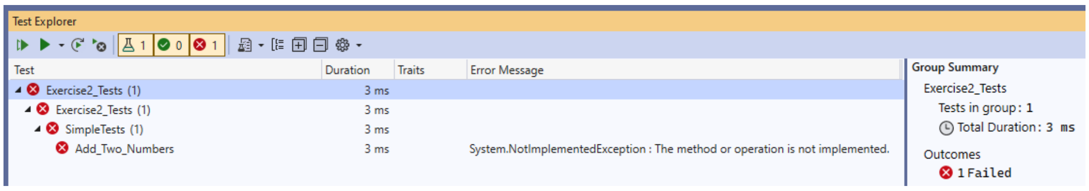
Test -> Run All Tests.



18 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.**



Test Explorer

Test	Duration	Traits	Error Message
Exercise2_Tests (1)	3 ms		
Exercise2_Tests (1)	3 ms		
SimpleTests (1)	3 ms		
Add_Two_Numbers	3 ms		System.NotImplementedException : The method or operation is not implemented.

Group Summary
Exercise2_Tests
Tests in group : 1
Total Duration : 3 ms
Outcomes
1 Failed

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

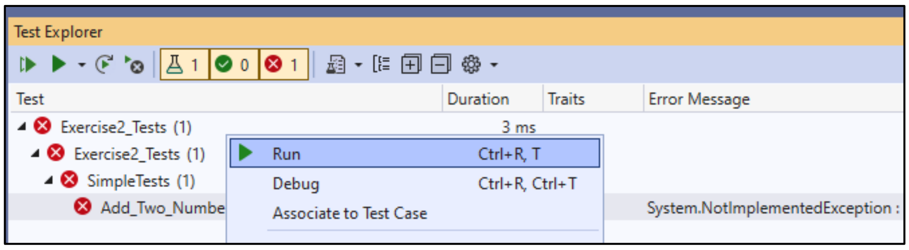
Delete the line of code: **throw new NotImplementedException;**

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.

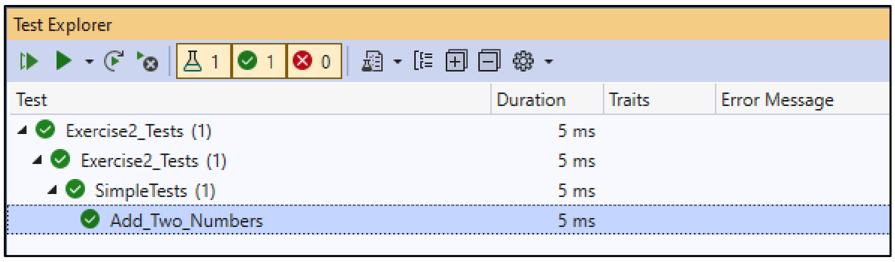
20 Right-click the failed test in **Test Explorer** and select **Run**.



Test Explorer

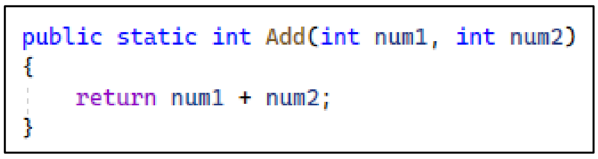
Test	Duration	Traits	Error Message
Exercise2_Tests (1)	3 ms		
Exercise2_Tests (1)			
SimpleTests (1)			
Add_Two_Numbe			System.NotImplementedException :

The test should now pass:



Test Explorer

Test	Duration	Traits	Error Message
Exercise2_Tests (1)	5 ms		
Exercise2_Tests (1)	5 ms		
SimpleTests (1)	5 ms		
Add_Two_Numbers	5 ms		

21	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;`
	`}`

22	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.
23	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.
24	Solutions are provided in the **End** folder for your reference.

If you have time

25	Open the `IfYouHaveTime` solution located in `Labs\07_Unit_Testing\Begin` folder.
26	Look at the code in the `Program.cs` file and notice it contains a reworking of the code you wrote for the 04 Functions lab. The code that determines which animal lives at a specified Pole has been refactored to live in a function called `GetArcticAnimal` and the code that determines which country a capital city is located in has also

	been refactored into a different function called `GetCountryForCapitalCity`
27	Add an xUnit Test project to the solution and create sets of tests that comprehensively prove both the `GetArcticAnimal` and `GetCountryForCapitalCity` functions work correctly. You may find it problematical to create a test that ensures the correct message for a capital city that does not exist in the CapitalCities enumeration gets returned by the `GetCountryForCapitalCity` function. Note, it is possible to cast an integer that has no equivalent value in an enumeration to be of that type: `CapitalCities city = (CapitalCities)99;`

Object-Oriented Programming (OOP) Concepts

The objective of these exercises is to consolidate your understanding of object-oriented programming (OOP) concepts and to experience the run time behaviours of reference and value types. The second exercise aims to start authoring classes with instance methods and data.

Part 1 – Calling pre-supplied methods

1	Open the existing pre-supplied Visual Studio solution (.sln) called 'MethodCallingPractice'. It can be found in folder labs\08 OOP\Begin\Types.
2	Open the file Program.cs. Inside the Main method you will see two commented out lines that make calls to a couple of test methods. The 2 test methods contain code that creates and manipulates objects of type Account (a reference type) and int (a value type).
3	Open the file Account.cs. You will see the implementation of a reference type (a class) that represents a bank account object. In Program.cs, expand the "Testing Ref Type behaviour" region and examine the TestRefType method and the two PassAccountByxxxx methods. You will note that the balances of the 2 accounts are being displayed 4 times. Create a table like the one below, either with a pen and paper or in NoteBook.exe and by just reading the code, write down what you

	think the Balance will be when read from the variables ac1 and ac2 at these four points in the code. ac1.Balance ac2.Balance 1st _________ __________ 2nd _________ __________ 3rd _________ __________ 4th _________ __________
4	Uncomment the line in the Main function that calls the TestRefType method and run the program checking how many of the values you got correct.
5	If you answers differed from those shown, use the Visual Studio debugger to step through the code. If you are unsure about the results, please ask your instructor to explain them.

Working with value types

In this section of the lab, you will see how value types work, and how they differ in their behaviour from reference types.

6	In C# the `int` data type (alias for `System.Int32`) is not a class it is a value type and thus will exhibit value type behaviour.
7	Comment out the call to `TestRefType` in the **Main** method of the Program class.
8	Uncomment the call to `TestValueType` in the **Main** method of the Program class.
9	Examine the method `TestValueType` and the two PassIntByxxxx methods. You will note that the ints are being displayed 4 times. Just by reading the code, write down in the table below what you think the value will be when read from the variables num1 and num2 at these four points in the code. num1 num2 1st __________ __________ 2nd __________ __________ 3rd __________ __________ 4th __________ __________
10	Run the program and check how many of the values you got correct. If you answers differed from those shown, use the Visual Studio debugger to step through the code. If you are unsure about the results, please ask your instructor to explain them
11	Now comment out the calls to `TestRefType` and `TestValueType` and uncomment the calls to TestIntArray and TestAccountArray.

12	Read the code being executed by each method, there are 2 simple questions to be answered, choose answers. Run the code and see if you were correct.

Part 2 – Creating a simple Account class

In this next exercise you will simultaneously author class Account whilst writing code in class Program that uses your new Account data type and the functionality it exposes.

1	Open the pre-supplied solution ClientExe.sln. You will find it in Labs\08_Object_Oriented_Programming_Concepts\Begin.
2	Open class Program. Expand the Main method. Type 'Acc' – there is no data type (class) called Account (yet).
3	Right click Solution 'ClientExe' in SolutionExplorer and choose Add/New Project. In the 'Add New Project' dialog choose 'Class Library' name it 'Finance' and ensure it is placed in the correct \Begin folder.
4	In the project 'Finance' you now have a file called Class1.cs containing the code of a class called 'Class1'. Rename the FILE in Solution Explorer to Account.cs and you will be prompted to rename the class as well.
5	You now have public class Account { }.
6	Inside Main type 'Acc' again – still no sign of data type Account.

7 In the ClientExe area of Solution Explorer right-click ‘Dependencies. Choose ‘Add Project Reference’, from the Projects tab choose ‘Finance’ and click ‘OK’.

8 Inside Main type ‘Acc’ again – still no sign of data type Account.

But if you type ‘Finance’<dot> you can see the `Finance.Account` data type.

Add a ‘using Finance’ clause at the top of the file and you can now type ‘Acc’ and see that ‘`Account`’ is a known data type.

9 Inside Main declare 2 variables of type Account

```
Account ac1, ac2;
```

10 Now type these 2 statements which show 2 different but useless objects being created.

```
ac1 = new Account();
ac2 = new Account();
```

11 Type a 4th statement and run your code and see that the result is false.

```
Console.WriteLine(ac1 == ac2); // these are DIFFERENT objs
```

12 Keep the 1st statement and DELETE the other 3.

13 You are going to add some functionality to the Account class. Declare 3 private fields inside the class noting the lowercase.

```
private string holder;
private decimal balance;  // will default to 0.00
private string accNo;    // can contain alphanumeric chars
```

14 Use the ‘ctor’ code snippet to author a ‘default constructor’.

A constructor is a special method that runs when an object is being instantiated from the class. Constructor methods can be configured to take parameters (just like ordinary functions). We will look at constructors more fully in the next session.

	Give it 2 parameters (name and balance) and 2 statements. The statements should store the parameters in the instance variables (giving the object some initial state) you will need to use the word 'this' once:
	
	`public Account(string name, decimal balance) {`
	`  holder = name;`
	`  this.balance = balance;`
	`}`

15	Returning to Main, the 2 statements that create the instances of the Account class will be showing syntax errors. This is because the constructors
	
	each need to be given a person's name and a starting balance. Edit the code as shown below.
	`ac1 = new Account("Fred", 100);`
	`ac2 = new Account("Susy", 200);`

16	Each account now has its own 'holder' and 'balance'. Each account also has its own 'accNo' but it has defaulted to null for each of them.
	
	We would like these 'Student Accounts' to each have an 'accNo' in the format 'SA-nnnn' where nnnn is an 'auto-incremented' sequential number padded to 4 digits with zeros. i.e. SA-0001, SA-0002 for Fred and Susy respectively.
	
	Declare a 4th field in Account as follows
	`private int nxtAccNo; // defaults to 0`
	Add this next statement to the bottom of the constructor method. Will this work do you think? Will the account numbers be SA-0001 and SA-0002 for 'Fred' and 'Susy' or will they both have SA-0001?
	`accNo = "SA-" + (++nxtAccNo).ToString().PadLeft(4,'0');`
	Well, they will both be SA-0001 because each account object will get its own version of nxtAccNo starting at zero.

	Solve the problem now by marking nxtAccNo as static this means there is only one copy of this numeric field shared between all the instances (but each instance will have its own accNo in format 'SA-nnnn' `private static int nxtAccNo; // defaults to 0 and is shared`

17	Let's prove that it has worked.
	
	Open class Account and author a method as follows (it won't compile yet)
	`public string GetDetails() {`
	`}`

18	We would like this method to return a tab-separated string containing the account number, the holder's name and the account balance. Easy to do via concatenation, but here is a perfect opportunity to see again how string interpolation works. Insert this single statement into the method. `return $“{accNo}\t{holder}\t{balance}”;`

19	Return to Main and after creating the 2 Account objects display the details of both objects on the Console. You really should not need to look at the code fragment below to see how to do this.
	`Console.WriteLine(ac1.GetDetails());`
	`Console.WriteLine(ac2.GetDetails());`
	To achieve this you could have typed no more than 'cw' Tab Tab an 'a' a <dot> a 'g' then '()'. If you typed more than that then you may want to try it again.
	
	Run the code. You should now be seeing output showing 'Fred' and 'Susy's correct account details with unique account numbers SA-0001 and SA-0002.

20	Now add further functionality to the Account class as follows
	`public void Deposit(decimal amt) {`
	`  balance += amt;`
	`}`
	`public bool Withdraw(decimal amt) {`
	`  bool result = false;`
	`  return result;             // just to make it compile`
	`}`

21	The Deposit method is straightforward – it will work, there is no upper balance.
	
	Withdraw however is a bool method as very soon you will want to 'listen' to see if a withdrawal has been successful.
	
	Here is the algorithm. If the amount that is being withdrawn is less than or equal to the balance on the account then the balance should be adjusted downwards and result set to true. Implement that code now.
	`public bool Withdraw(decimal amt) {`
	`  bool result = false;`
	`  if (?? ? ??) {`
	`    ?? ? ?                 // change the balance`
	`    ?? ? ?                 // ensure method returns true`
	`  }`
	`  return result;             // to make it compile`
	`}`
	This would work fine if there was no concept of an overdraft. But these are 'Student Accounts' – there is a £500 overdraft limit and it is the same for all Accounts so this should be held in a static (belonging to the class) variable.
	
	Declare a static decimal overdraftLimit with a value of 500.
	
	Also adjust the Withdraw(decimal amt) method to allow a student to withdraw as long as they do not go beyond their overdraft limit as opposed to below 0.

22	Perform a Deposit into ‘ac1’ and a successful WithDraw from ‘ac2’.
	
	Make these method calls before the display of the details. Check the methods have produced the correct values.
	
	Now change the parameter passed to Withdraw() so that it would take the student beyond their overdraft limit and check the balance does not change.
	
	You probably coded your Withdraw method as follows?
	`public bool Withdraw(decimal amt) {`
	`  bool result = false;`
	`  if (amt <= (balance + overdraftLimit)) {`
	`    balance -= amt;          // change the balance`
	`    result = true;           // ensure method returns true`
	`  }`
	`  return result;`
	`}`

23	It works fine but it is not obvious that overdraftLimit is ‘shared’ between all instances. Not clear that it belongs to ‘Account’ rather than ‘this’. Make the statement clearer perhaps by changing it to. `if (amt <= (this.balance + Account.overdraftLimit)) {`

24	Now we wish to enable transfers between accounts so that one student could do a little internet banking and ‘loan’ money to a friend.
	
	It could be written as an instance method and if it was its signature might be
	`public bool Transfer(decimal amt, Account to) {`
	`   ... calls to this.Withdraw and to.Deposit`

<table>
<tr><td></td><td>

```
}
```

The code that would invoke it might look like this:

```
ac1.Transfer(100M, ac2);  // 100 from ac1 to ac2
```

Alternatively, it could be written as a class (static) method with this signature.

```
public static bool Transfer(Account from, Account to,
                                         decimal amt) {

}
```

This latter version is perhaps more 'appropriate' for a method which by definition affects 2 accounts, but it is a matter of personal preference.

Write this method signature now, it will not compile yet of course.
</td></tr>
<tr><td>25</td><td>
Now for the implementation.

Because it is boolean it should declare a boolean result variable and set it to false (just being pessimistic).

The final statement should return the result.

In the main body of the method you should do a Withdraw from the 'from' account and only if that is a successful Withdraw() should it then do a Deposit() into the 'to' Account.

After the Withdraw & Deposit (or neither) operations have completed you should always display the result of attempting the transfer in the format

"Transfer Successful: Yes"
</td></tr>
</table>

	or
	
	“Transfer Successful: No”
	
	You should be able to implement that very easily without looking at the code block below that shows you the final code.
	
	Have an attempt now. It is a good chance to test yourself out on whether you understood how the conditional operator (?:) works as you should use it to produce the “Yes” or “No” strings.
	
	`public static bool Transfer(Account from, Account to,`
	`                                            decimal amt) {`
	`  bool result = false;`
	`  if (from.Withdraw(amt))`
	`  {`
	`    to.Deposit(amt);`
	`    result = true;`
	`  }`
	
	`  Console.WriteLine(`
	`    $"Transfer Successful: {(result ? "YES" : "NO")}");`
	`  return result;`
	`}`

26	Now go back to Main() and perform a Transfer of a modest sum from ‘ac1’ to ‘ac2’ or the other way round, depends which student ‘Fred’ or ‘Susy’ you think has blown their ‘terms’ allowance during ‘freshers week’. Remember the method you are calling is static not instance.
27	Before we wrap up note that although we can display the ‘Details’ of any account object we do not have the ability to simply find out the balance on the account or in fact the name of the holder of an account.

	Author and code now a simple public decimal GetBalance() method and a public string GetHolder() method. They will be needed in the challenges that follow and in the next chapter's lab.

If You Have Time: Coding Challenge 1

28	Add the 2 account references 'ac1' and 'ac2' to a List of Account objects, add a 3rd and a 4th if you want.
	
	Author a method (in Program) called ProcessAccounts with the following signature.
	`public static void ProcessAccounts(List<Account> accs) {`
	`}`
	This method should loop through the accounts it receives and add a £10 bonus (call Deposit()) to each of them.
	
	Call the method from Main and after getting control back loop through the array yourself displaying the account holders name and the account balance to check that they have all gone up by £10.

If You Have Time: Coding Challenge 2

29	Declare and create a List of string called names exactly as follows
	`List<string> names = new List<string>`
	`{"Ann","Anne","Annie","Anneka","Annabel"};`

30	Declare and create a List of Account objects called studentAccs

	Declare and create an instance of class Random, it has a useful Next() method that you soon will use repeatedly.

31	Write a simple for loop that runs 'the right number of times' that creates an account for each of the names and adds the object to the studentAccs collection.
	
	The account holders name should be the 'next' name from the names array.
	
	Use instance method Next() of the Random object you created earlier to generate a random number of £ in the range 10-99 inclusive to be used as opening balance.
	`for(int i = 0; i < ???; i++) {`
	`  ??.??(new ??(??,??));`
	`}`

32	Write a foreach loop that steps through the list of accounts displaying the details of each account. Run the code. The 5 names should now have an account each with a balance in the range £10 - 99 inclusive. Now you are going to write code in a loop so that a Transfer happens from each account holder to the 'next' (5 transfers in total). i.e. it will transfer money from 'Ann's account to 'Anne's account.

	From ‘Anne’s account to ‘Annie’s account etc and lastly (the tricky bit) from the final person ‘Annabel’ to the first person ‘Ann’s account. The amount of money that each person transfers out is the length of their own name. It is easy to write code to determine the length of the account holders name as class String has a ‘Length’ property. So Ann will give £3 to Anne. Anne will give £4 to Annie. Annie will give £5 to Anneka… finally Annabel will give £7 to Ann. Each person’s balance will drop by £1 except Ann whose balance will go up by £4 (£3 out and £7 in). Write this code now without hard coding any names or amounts.

If You Still Have Time: Coding Challenge 3

Add an xUnit test project to the solution and create a series of tests that ensure the code in the Account class is comprehensively behaving as expected.

Properties and Constructors

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

The Car Library and Console Projects

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.

If You Have Time - Coding Challenge: Enhance the Finance ClientExe Projects

1	Either return to your solution to the previous exercise (08 OOP Concepts) or open the one in the Begin folder.
2	Rework the Account class so: • The holder field is private with access being given to it via a public property called Holder. The property should be both Gettable and Settable with the following validation being applied: o If the passed in value is null the value of holder should be set to "*** Anon ***". o If the passed in value is **not** more than 2 character the value of holder should be set to the passed in value surrounded by 3 sets of asterixes. E.G. If the passed in value is "Mo" the value of holder should be set to "*** Mo ***". • The private balance filed should be replaced by a gettable but **not** settable property called Balance. You will need to rework other elements of code in both the Account and Program classes in order to make the code compile. Add code to the Program class that fully tests the new features.

3	We want to be able to create an account object allowing the balance to default to 0. i.e. a single parameter constructor of string name only. Give the Account class this as a second constructor. Use constructor chaining (see relevant slides in the PowerPoint appendix) so the single parameter constructor calls the two-parameter constructor passing the name and a zero for the balance. Add code to the Program class that fully tests the new features.

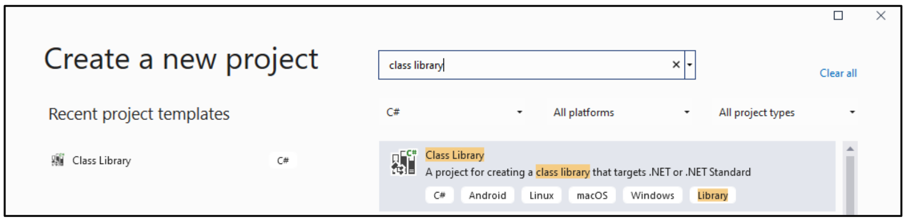
If You Still Have Time

Add an xUnit Test project to the solution and create a set of unit tests that ensure the code in the Account class is working correctly.

Inheritance

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

1	Locate the **'Labs\03_Inheritance_and_Abstract_Classes\Begin'** folder and, inside it, create a new Visual Studio project of type Class Library called **LendingLibrary**.



Create a new project

class library

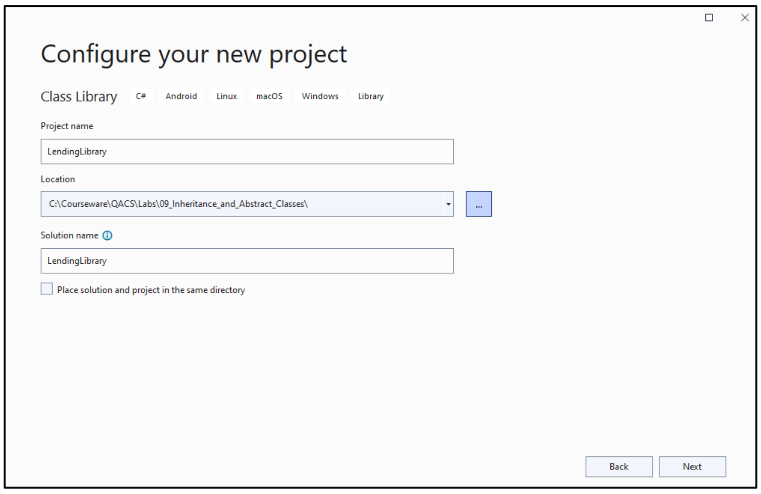
Clear all

Recent project templates

Class Library C#

C# · All platforms · All project types

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library



Configure your new project

Class Library C# Android Linux macOS Windows Library

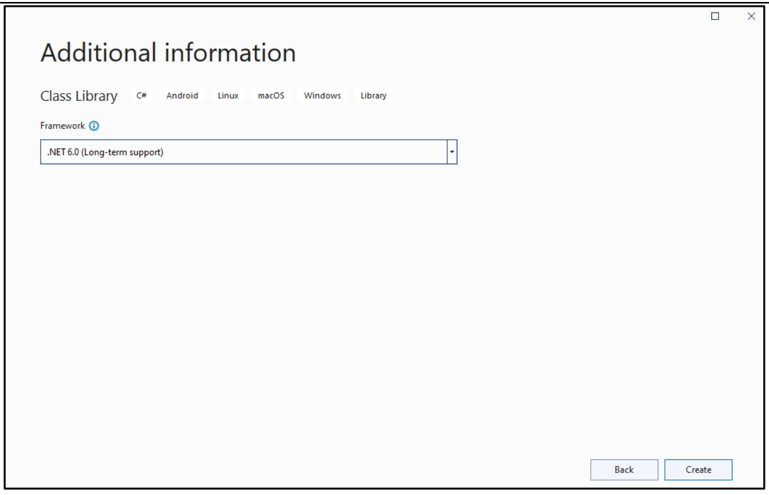
Project name
LendingLibrary

Location
C:\Courseware\QACS\Labs\09_Inheritance_and_Abstract_Classes\

Solution name
LendingLibrary

☐ Place solution and project in the same directory

Back Next



Additional information

Class Library C# Android Linux macOS Windows Library

Framework

.NET 6.0 (Long-term support)

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 **‘Labs\03_Inheritance_and_Abstract_Classes\Assets’** folder (copy just the first part of the file, up to the end of the test).

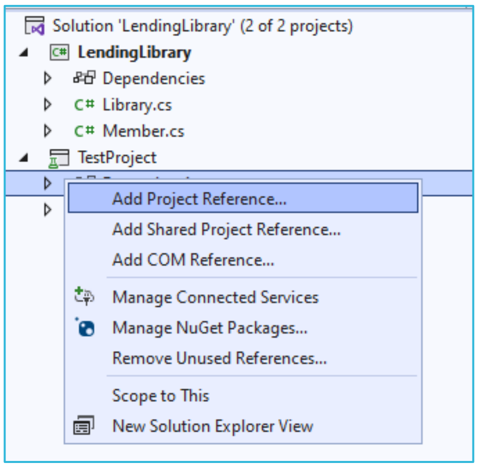
```
[Fact]
public void Create()
{
  Library library = new Library();
  Member ittzy = library.Add(name: "Ittzy Child", age: 15);
  Member irma = irma = library.Add(name: "Irma Adult", age: 73);

  Assert.Equal(2, library.NumberOfMembers);
  Assert.Equal(1, ittzy.MembershipNumber);
  Assert.Equal(2, irma.MembershipNumber);
}
```

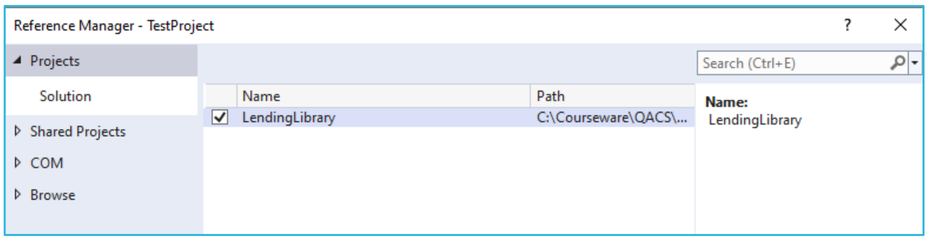
3 Create **Library** and **Member** classes in the **LendingLibrary** project, then copy in the code listed just after the 'Create' test.

```
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;
  }
}



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.


Solution 'LendingLibrary' (2 of 2 projects)
LendingLibrary
Dependencies
Library.cs
Member.cs
TestProject
Add Project Reference...
Add Shared Project Reference...
Add COM Reference...
Manage Connected Services
Manage NuGet Packages...
Remove Unused References...
Scope to This
New Solution Explorer View

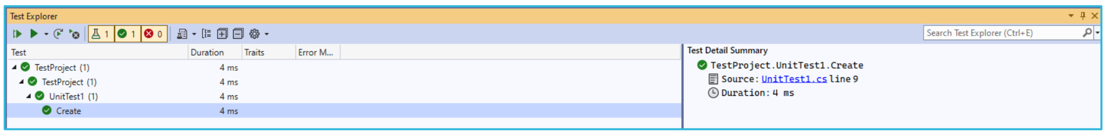

Reference Manager - TestProject

	Name	Path
✓	LendingLibrary	C:\Courseware\QACS\...

Name: LendingLibrary

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

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



```
Test Explorer
1  1  0                                              Search Test Explorer (Ctrl+E)
Test                    Duration  Traits  Error M...   Test Detail Summary
◢ TestProject (1)         4 ms                           TestProject.UnitTest1.Create
  ◢ TestProject (1)       4 ms                             Source: UnitTest1.cs line 9
    ◢ UnitTest1 (1)       4 ms                             Duration: 4 ms
        Create            4 ms
```

4 We are going to need the **library** instance and **iittzy** and **irma** 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.

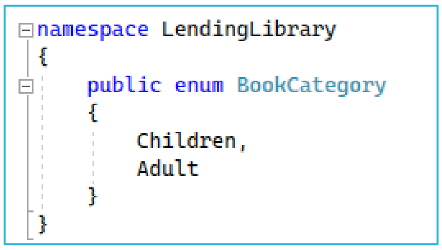
Note: If you get stuck, we've shown it in the Assets folder.

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();
  ittzy = library.Add(name: "Ittzy Child", age: 15);
  ittzy.Street = "1 the High Street";
  ittzy.City = "Buddlington";
  ittzy.OutstandingFines = 25M;
  irma = irma = library.Add(name: "irma Adult", age: 73);
  irma.Street = "2 the High Street";
  irma.City = "Cruddlington";
  irma.OutstandingFines = 2500M;
}
```

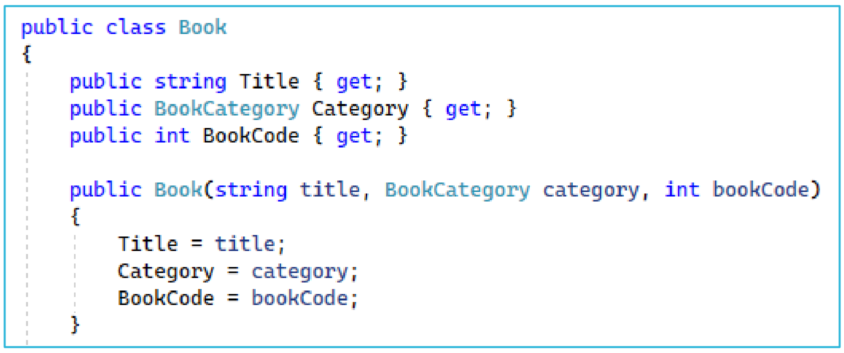
6 In this library we have different rules for borrowing a book, dependent on whether the member is over 16 or not.

We are going to need this enum. Add it to your LendingLibrary, in a file called BookCategory.cs



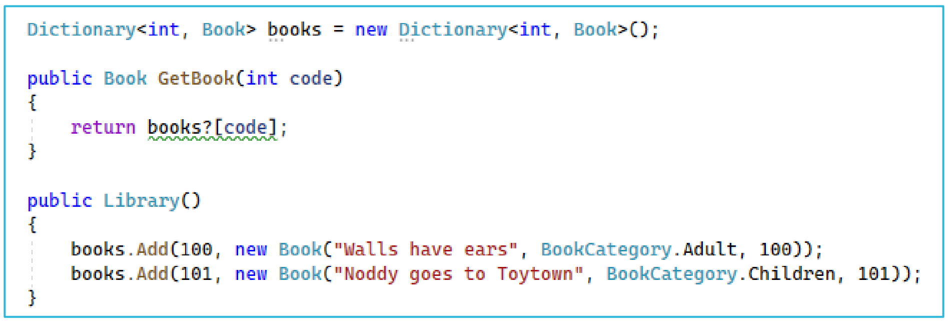
```
namespace LendingLibrary
{
    public enum BookCategory
    {
        Children,
        Adult
    }
}
```

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;
    }
```

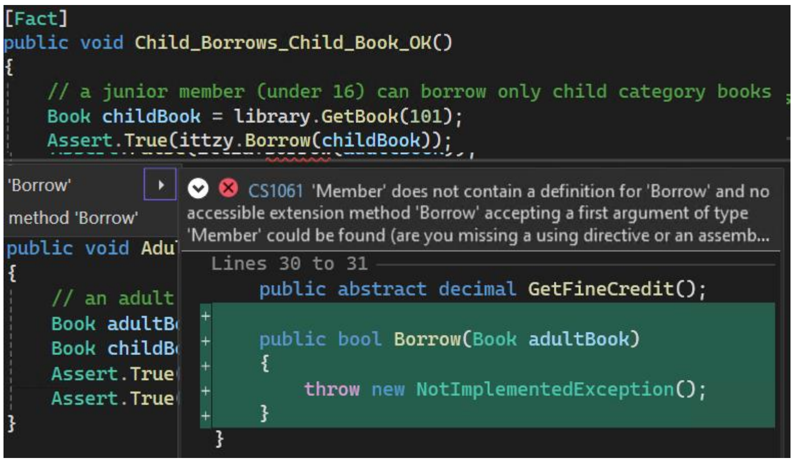
We will also need some Book code in the **Library** class:



```
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:



```
[Fact]
public void Child_Borrows_Child_Book_OK()
{
    // a junior member (under 16) can borrow only child category books
    Book childBook = library.GetBook(101);
    Assert.True(ittzy.Borrow(childBook));
```

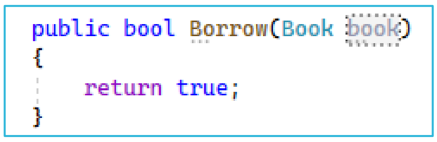
'Borrow'
method 'Borrow'

```
public void Adu
{
    // an adult
    Book adultB
    Book childB
    Assert.True
    Assert.True
}
```

CS1061 'Member' does not contain a definition for 'Borrow' and no accessible extension method 'Borrow' accepting a first argument of type 'Member' could be found (are you missing a using directive or an assemb...

```
Lines 30 to 31
        public abstract decimal GetFineCredit();
+
+       public bool Borrow(Book adultBook)
+       {
+           throw new NotImplementedException();
+       }
    }
```

Populate the method like this:

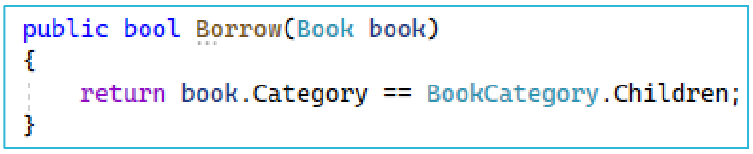


```
public bool Borrow(Book book)
{
    return true;
}
```

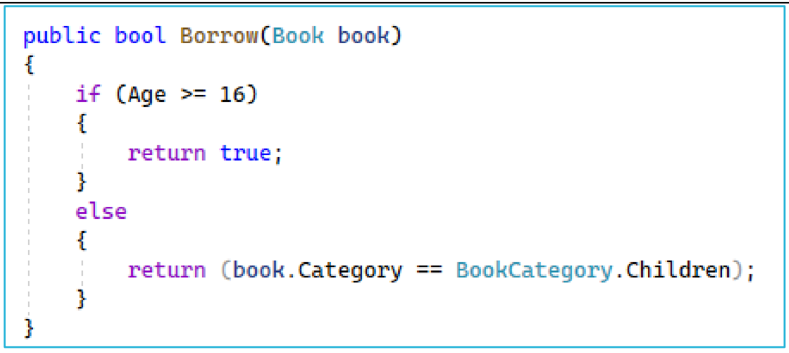
Run the Test and confirm it passes.

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(ittzy.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(irma.Borrow(adultBook));`
	`  Assert.True(irma.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);
    }
}
```

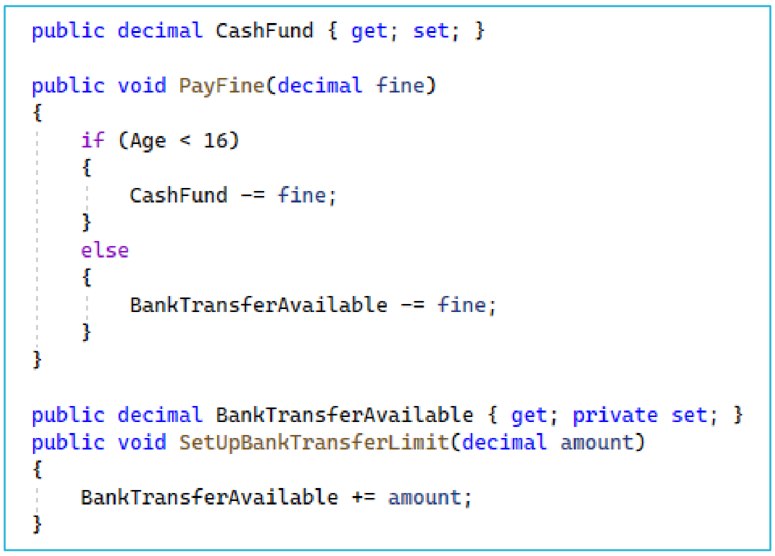
Run the Tests. All tests should now pass.

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()
{
  ittzy.SetFineLimit(20M);
  ittzy.PayFine(7M);
  Assert.Equal(13M, ittzy.GetFineCredit());
}


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

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 two 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.

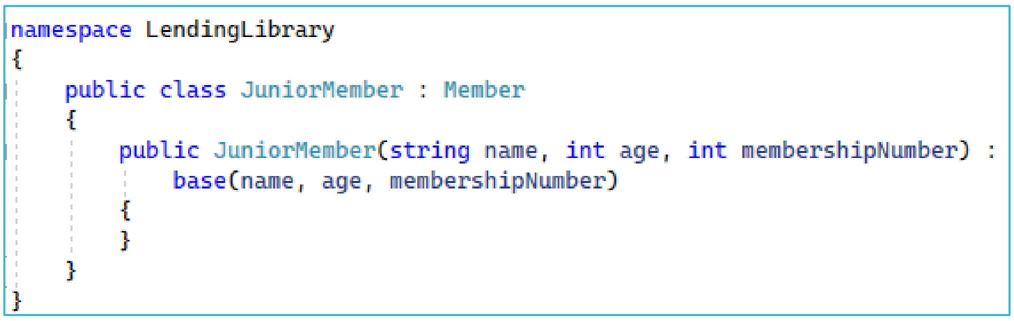
All tests will pass as no code has been changed.

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**.

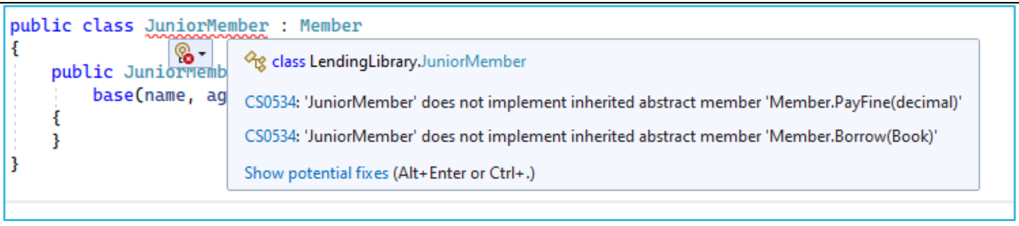
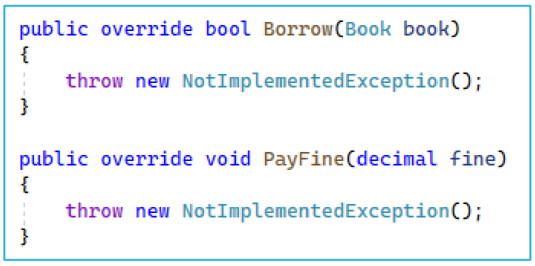
14 We need to modify **Library.Add** to create either a *Junior* or an *Adult* member. Change Library.Add() to this:

	```
	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:  ```public override bool Borrow(Book book) { throw new NotImplementedException(); } public override void PayFine(decimal fine) { throw new NotImplementedException(); }```
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 iittzy is a JuniorMember and Irma is an AdultMamber:
	
	 `public class UnitTest1`
	
	 `{`
	
	 `  Library library;`

```
  Member ittzy;

  Member irma;
```

There are two ways of fixing this:

1) Make iittzy a Junior and irma 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;

Member Iittzy;

Member irma;


public UnitTest1()

{

  library = new Library();

  Iittzy = (JuniorMember) library.Add(

      name: "Iittzy Child", age: 15);

  Iittzy.Street = "1 the High Street";

  Iittzy.City = "Buddlington";

  Iittzy.OutstandingFines = 25M;

  irma = (AdultMember) library.Add(

      name: "irma Adult", age: 73);

  irma.Street = "2 the High Street";

  irma.City = "Cruddlington";

  irma.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()`
	`{`
	`  Iittzy.SetFineLimit(20M);`
	`  Iittzy.PayFine(7M);`
	`  Assert.Equal(13M, Iittzy.GetFineCredit());`
	`}`

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

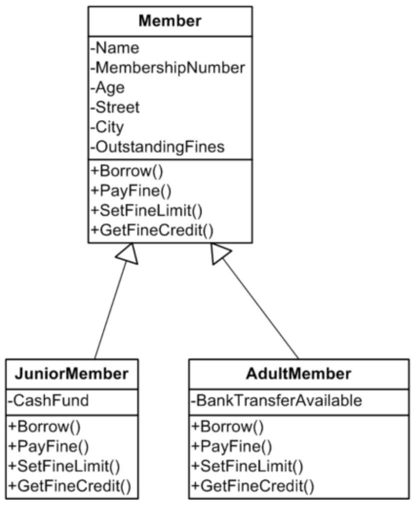
What happens when Iittzy turns 16?

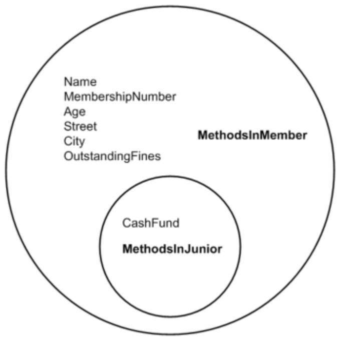
Easy, you say – you just create her as an AdultMember

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. Iittzy will feature in various collections. She would need to be removed from these and the new Iittzy would need to be inserted silently (probably breaking the rules we have established).

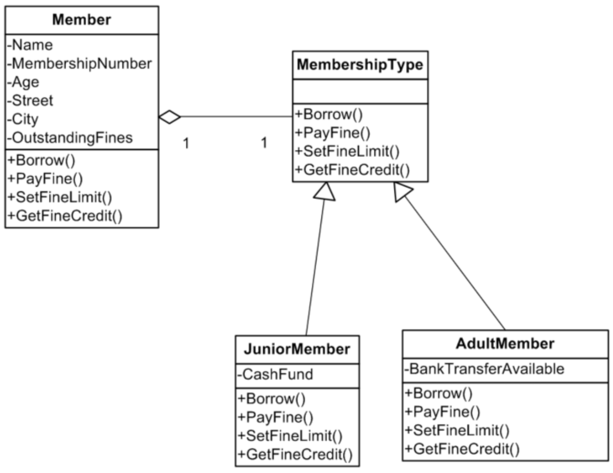
Our current class diagram and a representative object is shown below:

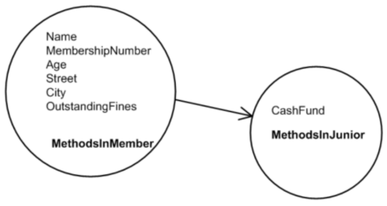




It's like we have one object built from two recipes – the Member recipe and the JuniorMember recipe. And when we destroy this object, we not only throw away the CashFund and MethodsInJunior (which is fine), we also throw away all the data and methods in the Member bit of the object.

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.





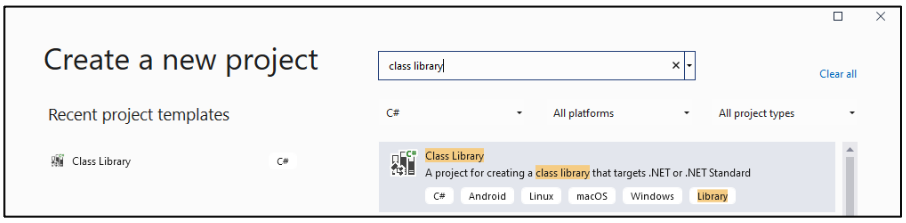
If you have plenty of time and feel very confident, then go ahead and have a go at changing your solution to be the State Pattern.

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 Locate the **'..\Labs\04_Interfaces\Begin'** folder and, inside it, create a new Visual Studio project of type Class Library called **InterfaceProject**.



Create a new project

class library

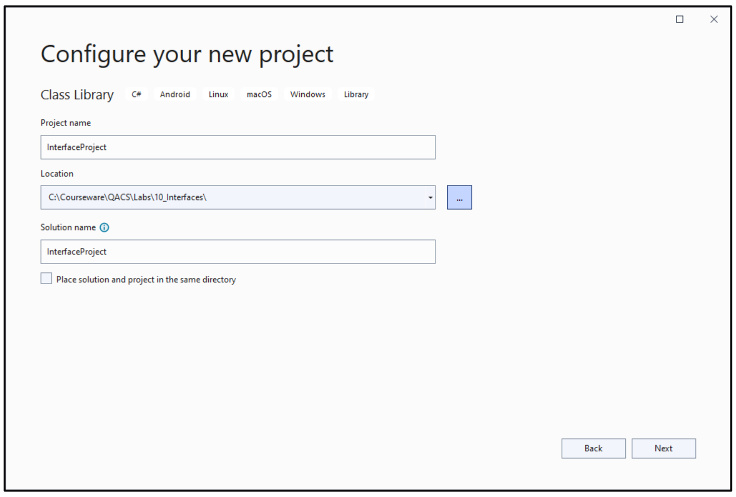
Clear all

C# | All platforms | All project types

Recent project templates

Class Library C#

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library



Configure your new project

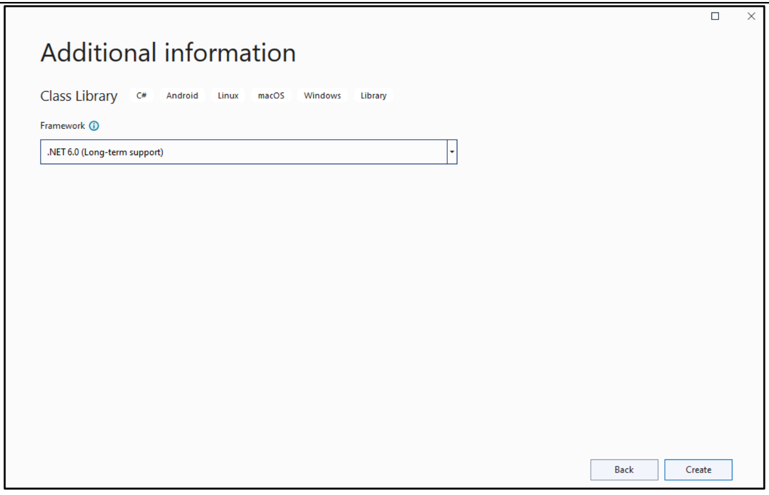
Class Library C# Android Linux macOS Windows Library

Project name: InterfaceProject

Location: C:\Courseware\QACS\Labs\10_Interfaces\

Solution name: InterfaceProject

☐ Place solution and project in the same directory



Additional information

Class Library C# Android Linux macOS Windows Library

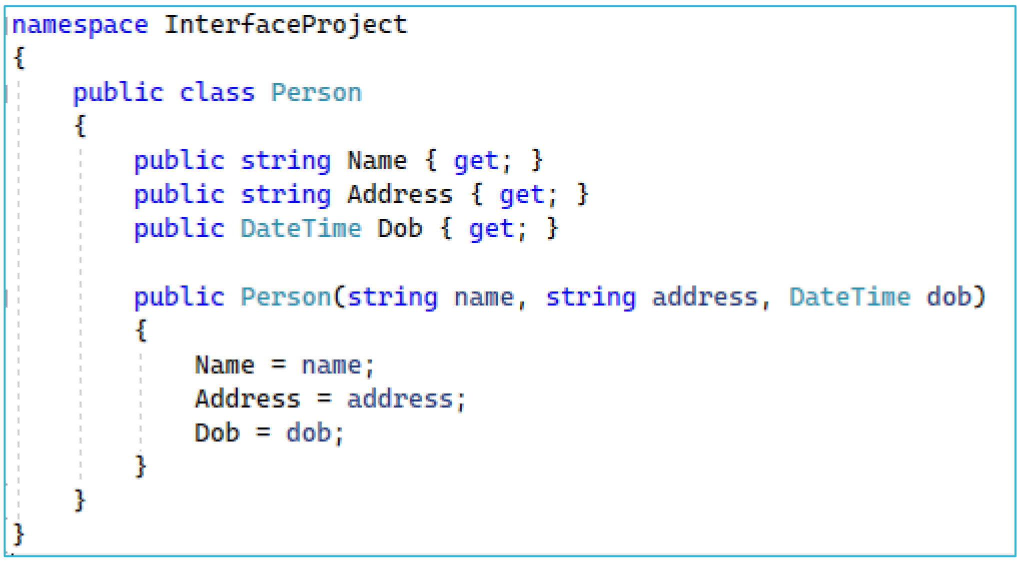
Framework

.NET 6.0 (Long-term support)

Delete **Class1**.

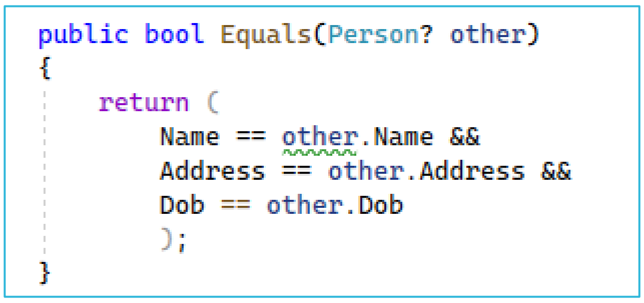
Add to this solution an XUnit Test project called **TestProject**.

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;
        }
    }
}
```

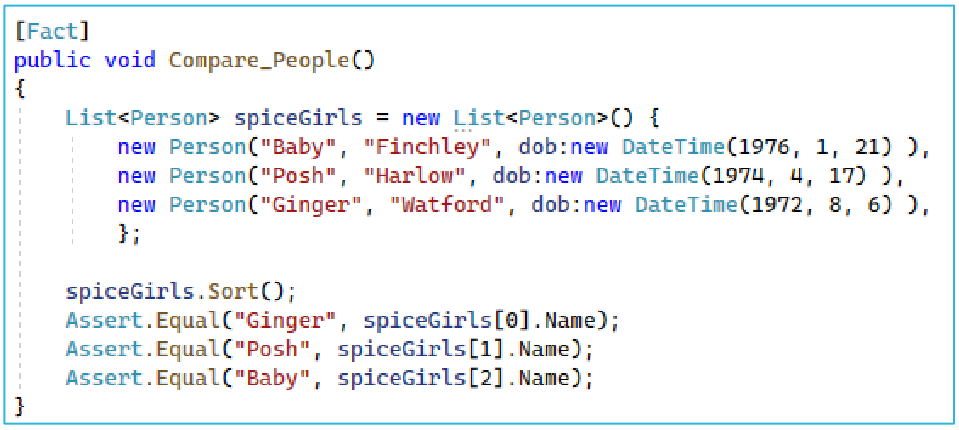
3	Replace the provided Test1() with the two tests in **IEquatable.txt** from the **'..\Labs\04_Interfaces\Assets'** folder and resolve the red squiggles.
4	Run the tests. The first test fails, the second test passes by accident. The problem is that the compiler doesn't know how to figure out if two Persons 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
        );
}
```

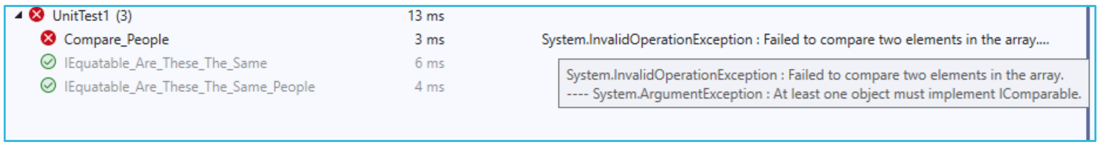
6	You will now compare some people, the Spice Girls, by putting them in order based on their DateOfBirth. We'll put these in DateOfBirth order.

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:



```
◢ ✖ UnitTest1 (3)                        13 ms
    ✖ Compare_People                      3 ms    System.InvalidOperationException : Failed to compare two elements in the array....
    ✓ IEquatable_Are_These_The_Same       6 ms
    ✓ IEquatable_Are_These_The_Same_People 4 ms

System.InvalidOperationException : Failed to compare two elements in the array.
---- System.ArgumentException : At least one object must implement IComparable.
```

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.

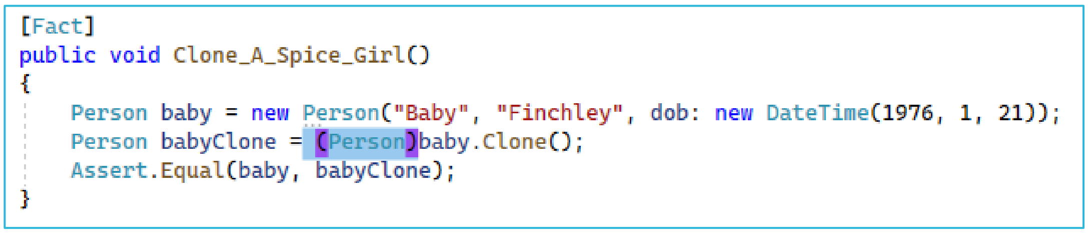
The interface you need is **ICloneable**. Make Person implement the interface **ICloneable**.

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.

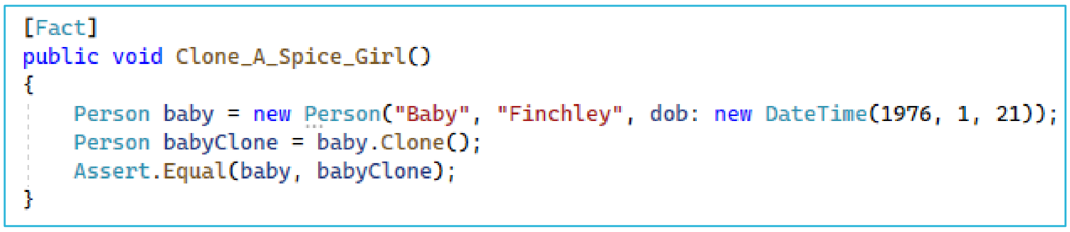
It’s a pity that the framework does not offer a generic version of **ICloneable.**

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:

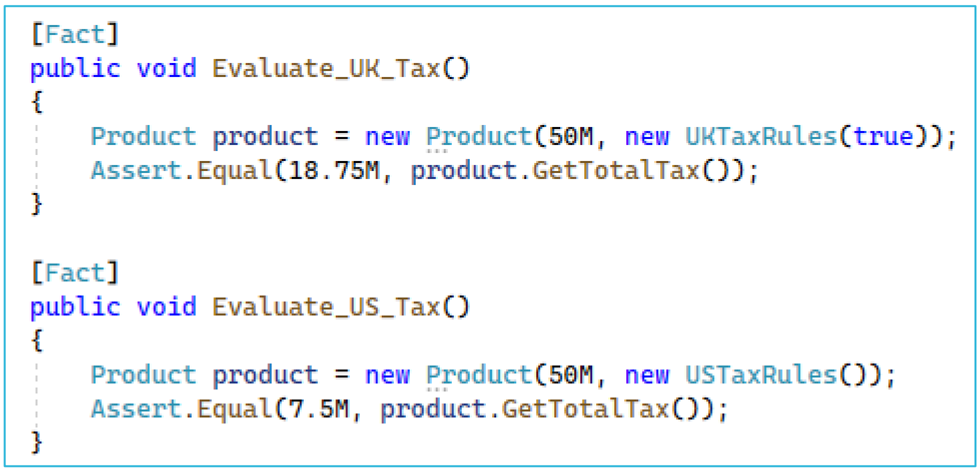


```
[Fact]
public void Clone_A_Spice_Girl()
{
    Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));
    Person babyClone = baby.Clone();
    Assert.Equal(baby, babyClone);
}
```

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.

If you have time: Explicit interfaces

NOTE, This part of the lab makes use of concepts discussed on slides 13 to 15 of the appendix

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

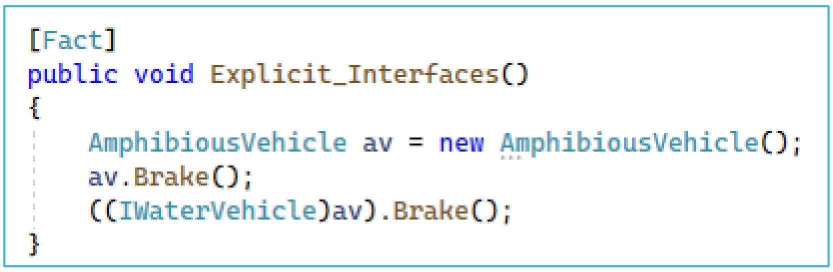
K=all-wheel drive

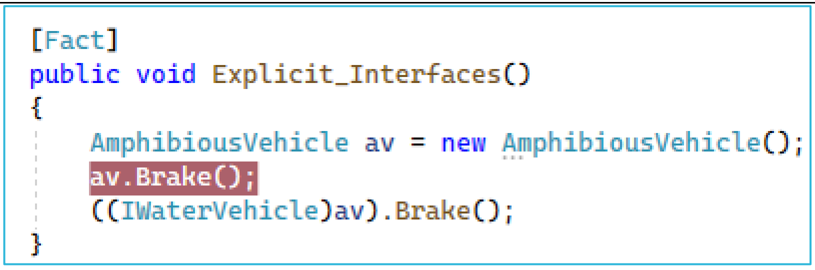
W=2 powered rear axles

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();
}
```

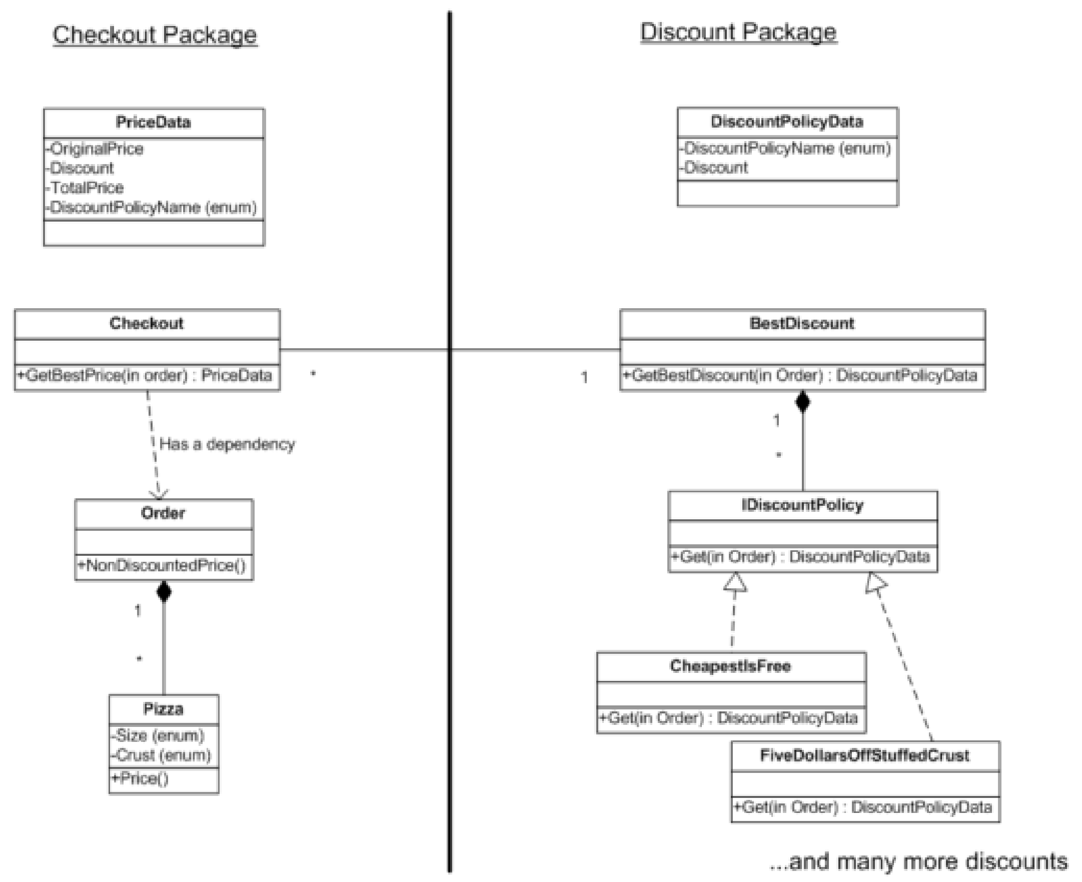
Right-click in the test and choose **Debug Tests.** Step into the code using and **Step Into (F11)** to confirm that both brake methods are called.

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.



Checkout Package
Discount Package
PriceData
-OriginalPrice
-Discount
-TotalPrice
-DiscountPolicyName (enum)
DiscountPolicyData
-DiscountPolicyName (enum)
-Discount
Checkout
+GetBestPrice(in order) : PriceData
*
1
BestDiscount
+GetBestDiscount(in Order) : DiscountPolicyData
Has a dependency
Order
+NonDiscountedPrice()
1
*
Pizza
-Size (enum)
-Crust (enum)
+Price()
IDiscountPolicy
+Get(in Order) : DiscountPolicyData
CheapestIsFree
+Get(in Order) : DiscountPolicyData
FiveDollarsOffStuffedCrust
+Get(in Order) : DiscountPolicyData
...and many more discounts

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 **'..\Labs\06_Delegates_and_Lambdas\Pizza'** 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’.

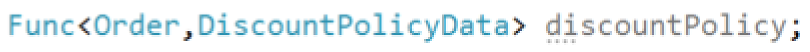
In this instance, we could have a neater syntax that doesn’t care about the name of the discount policy method and just cares about the signature.

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**:

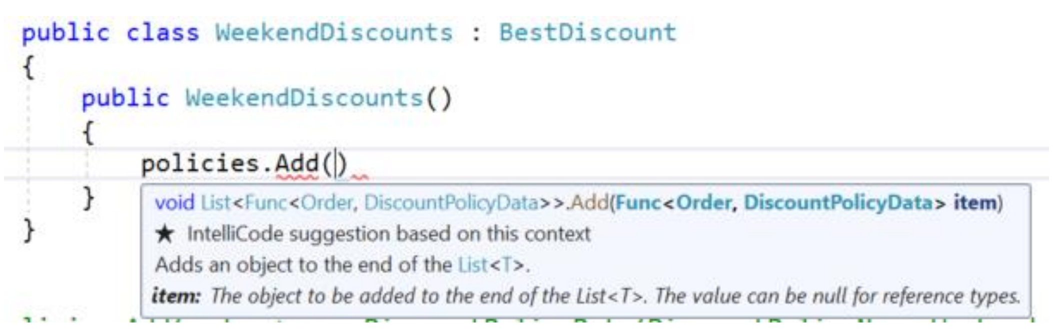
```
Weekend_10_Percent_Off
```

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:



```
public class WeekendDiscounts : BestDiscount
{
    public WeekendDiscounts()
    {
        policies.Add(|)
    }
}
```

void List<Func<Order, DiscountPolicyData>>.Add(Func<Order, DiscountPolicyData> item)
★ IntelliCode suggestion based on this context
Adds an object to the end of the List<T>.
item: The object to be added to the end of the List<T>. The value can be null for reference types.

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.

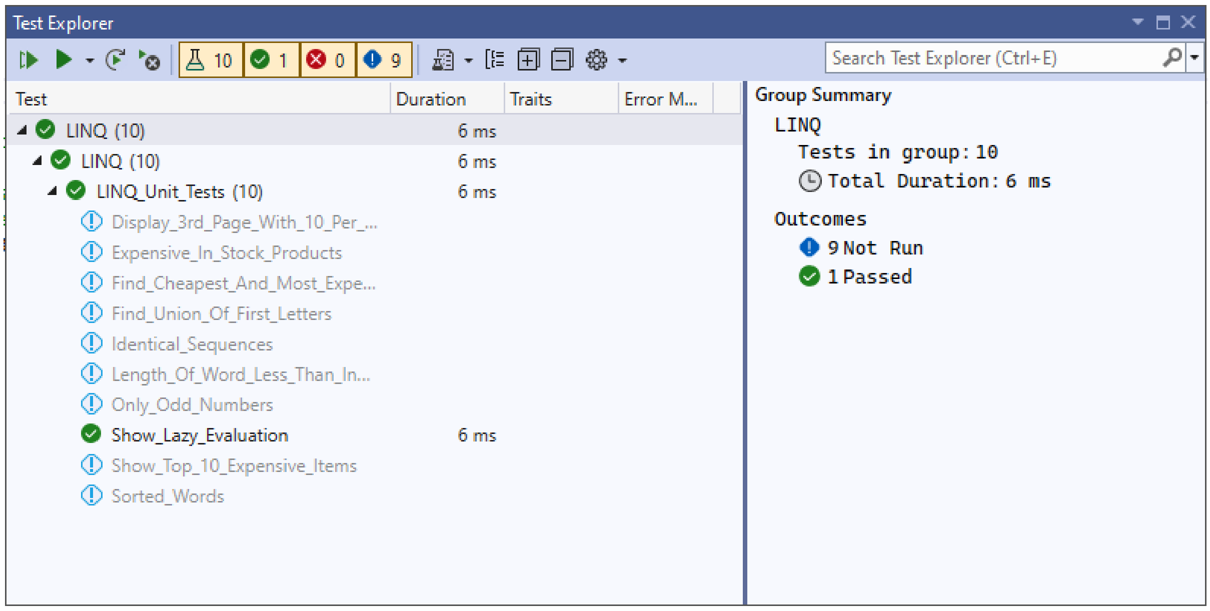
```
policies.Add(order=> new
  DiscountPolicyData(DiscountPolicyName.Weekend_10_Percent_Off,
                                            order.NonDiscountedPrice * 0.1M) );
```

Language Integrated Query (LINQ)

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

Writing LINQ Queries

1. Open **'..\Labs\07_LINQ\Begin\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.



Test	Duration	Traits	Error M...
LINQ (10)	6 ms		
LINQ (10)	6 ms		
LINQ_Unit_Tests (10)	6 ms		
Display_3rd_Page_With_10_Per_...			
Expensive_In_Stock_Products			
Find_Cheapest_And_Most_Expe...			
Find_Union_Of_First_Letters			
Identical_Sequences			
Length_Of_Word_Less_Than_In...			
Only_Odd_Numbers			
Show_Lazy_Evaluation	6 ms		
Show_Top_10_Expensive_Items			
Sorted_Words			

```
Group Summary
  LINQ
    Tests in group: 10
    Total Duration: 6 ms

  Outcomes
    9 Not Run
    1 Passed
```

Exception Handling

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

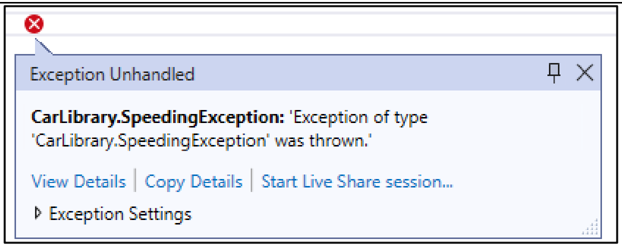
1	Open the **CarLibrary** solution in: **'..\Labs\08_Exception_Handling\Begin'**
2	Comment out the existing code in **Program.cs**
3	Open **Car.cs** and override the **ToString** method to display detailed car information: `return $"Car Make is {Make}, Model is {Model}, Colour is {Colour}, Speed is {Speed} 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:

	 Exception Unhandled **CarLibrary.SpeedingException:** 'Exception of type 'CarLibrary.SpeedingException' was thrown.' View Details | Copy Details | Start Live Share session... ▷ Exception Settings
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");`
	`}`

1.1. 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.

90 Working with Data and Files

Working with JSON data

1	Create a console app called JSON_Films.
2	Use NuGet to add the Newtonsoft.Json package
3	Add a class to the project that is suitable for holding film information (film_id, title, synopsis, director, release_date).
4	Using the file called Films.json in the Assets folder deserialize a JSON file containing films into a List<Film> object.
5	Display the film information on a console screen.
6	Add another film to the list.
7	Serialize the list back out to the JSON file.
8	If you have time, duplicate the code but use dynamic / anonymous types rather than purpose-built classes.

Reading and Writing CSV files

9	Create a console app called MoviesCSVFile..
10	Use NuGet to add the CSVHelper package
11	Add a class to the project that is suitable for holding film information (Title, ReleaseYear, Genres, Revenue and StreamedOn).

11	Write code to read the CSV file "streaming_movies.csv" into a list using a custom class suitable for the data within each row. You will find the file in the Assets folder.
12	Filter the list to get just movies on the Netflix platform, and sort it by movie title.
13	Write the filtered list of movies to a new CSV file, but only including the columns: Title, ReleaseYear, and Revenue.

Streams

14	Open the Visual Studio solution called Streams.sln located in the Begin folder.
15	Review the code that already exists. You will notice there are functions called CompressIntoByteArray, DecompressByteArrayIntoString, EncryptString and DecryptString. Three of these functions contain ready written (and functioning code) but the DecompressByteArrayIntoString currently does nothing with the passed in byte array.
16	Using the CompressToByteArray function as a guide try to add appropriate code to the DecompressByteArrayIntoString to make the decompression process work.

Before moving on don't forget to commit and push your work to GitHub.

Learn. To Change.

QA.com