School of Science, Computing and Engineering Technologies

Object-Oriented Programming

Pass Task 1.2: Object-Oriented "*Hello, World!*"

Overview

"*Hello, World!*" is often the first program that you write in a new programming language or with a new set of programming tools. In this task you will create an object-oriented version of this classic program.

Purpose:	Demonstrate that you have successfully set up Visual Studio and the .NET framework for programming in C#.
Task:	Create a "*Hello, World!*" program, and extend it to output custom messages for different user names. **The tasks contain personalized requirements**.
Deadline:	Due by end of week two, **Friday, 09 August 2024, 23:59 AEDT (Firmed)**.

Submission Details

All students have access to the Adobe Acrobat tools. Please print your solution to PDF and combine it with the screenshots of the console output and a screenshot of Visual Studio with an active breakpoint.

Instructions

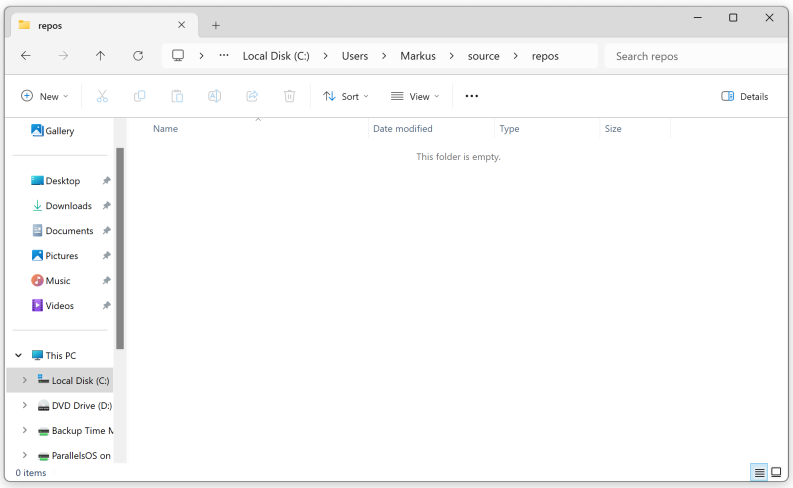
The first task includes the steps needed for you to install the tools you will need in this unit. You will then use these tools to create the classic "*Hello, World!*" program.

1. Install the tools you need for your platform:

   - Windows:
     Refer to the installation *Guide Environment Setup – Windows* on Canvas for details regarding the installation of Visual Studio 2022 on your Window computer. The Community suffices. Please note that you need to sign in into you Microsoft account when using the Community edition. If you have a license key for Visual Studio, do not forget to register Visual Studio. Visual Studio stops working after a trial period of either 15 or 30 days without sign-in or registered license key.
   - macOS:
     Refer to the installation *Guide Environment Setup – macOS* on Canvas for details regarding the installation of Visual Studio 2022 or Visual Studio Code on your Mac. Visual Studio and Visual Studio Code works with both Apple Silicon and Intel. Please note that you need to sign in into you Microsoft account when using the Community edi-tion.

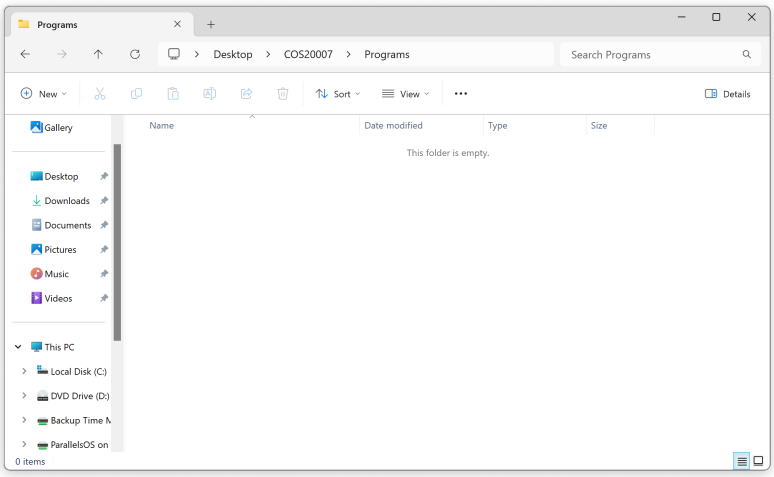
Note: If using the computers in labs, Visual Studio 2022 Enterprise may be available.

- In order to develop a program, you need a working directory. In Visual Studio 2022, this working directory is called repos and it is located in the current user's folder source\repos:



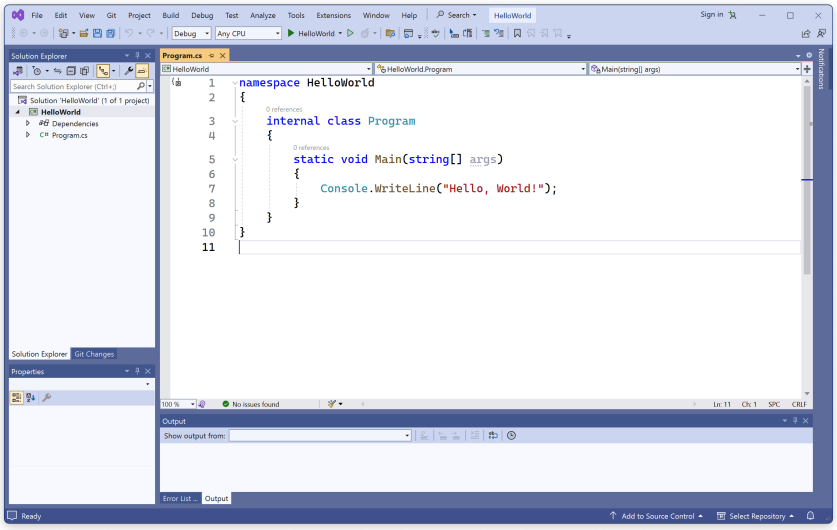
You can use this location, but in a shared environment like the university lab computers there is no guarantee that the contents of this folder will be retained between sessions. Lab machines get routinely reimaged. Also, you may want to separate your

projects based on course work. For example, you may want to assemble all your files (i.e., lecture notes, assignments, and projects) in a semester-specific subject directory. For example, you could use Desktop\COS20007\Programs (or another suitable location) as your working directory.



2. Open **Visual Studio** and follow the instructions in *Guide Environment Setup – Windows* or *Guide Environment Setup – macOS* to create a new project.

   The generated project is a fully functional C# "Hello, World!" application:



   Run the application. Initially, you should use Debug/Start Without Debugging. This guarantees that the Debug Console remains open after the program terminates. At this stage, you only verify that the program produces the expected output. Later when you debug your program, you will set breakpoints at specific code lines and the debugger will stop at those lines, allowing you to inspect the state of your program more closely (i.e., review or change the value of variables, or step through the program line-by-line).

   Review every language construct used in "*Hello, World!*".

Object-oriented programs work differently to procedural programs. An object-oriented program consists of *objects* that *know*, and *can do things*. When creating an object-oriented program, you design the kinds of objects you want, the things they know, and the things they can do. The program then coordinates actions of objects by "*sending messages*", asking receiver objects to *do things* or to return *things they know* in response to receiving messages.

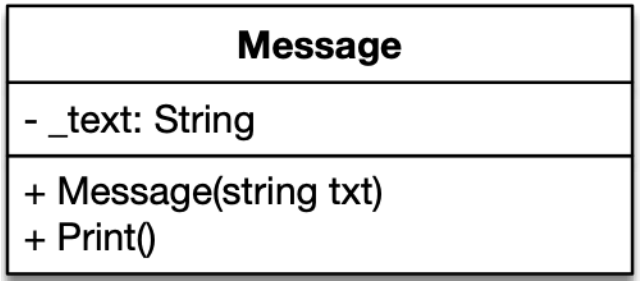
The "*Hello, World!*" program is technically not "object-oriented", however, it uses a class to produce output, namely *Console* which represents standard input/output operations.

C# is an object-oriented, class-based, general purpose programming language. In class-based languages *classes* and *objects* separate important concerns. Classes form extensible templates that can be used to create objects. Objects are the fundamental components of computation. More precisely, objects consist of a collection of *instance variables*, representing the state of the object, and a collection of *methods*, representing the behavior that the object is capable of performing.

In C#, objects are created from classes. Classes provide initial values for instance variables and the bodies for methods. All objects generated from the same class share the same methods, but contain separate copies of the instance variables. New objects are created from a class by applying the ***new*** operator to the name of the class.

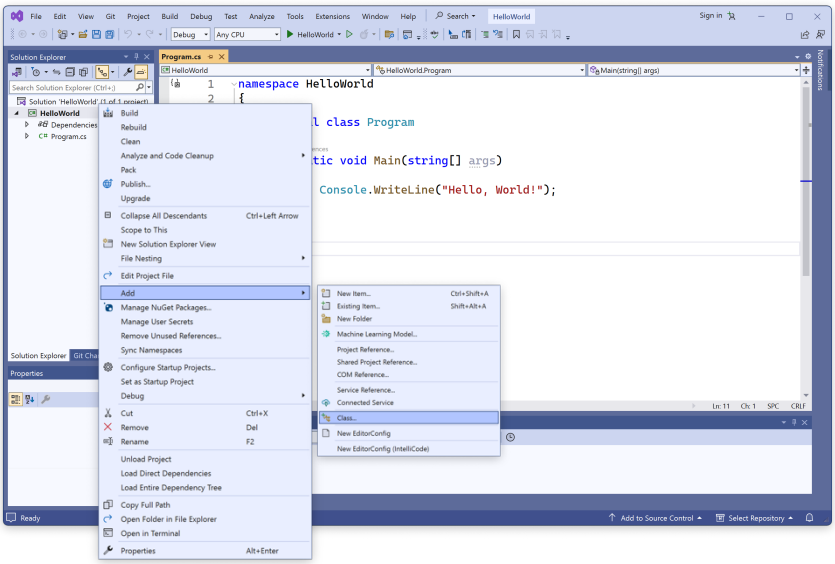
To create your own objects you first need to create a class, and then use that class to create an object for you.

3. Read the UML Class Diagrams Tutorial by Robert C. Martin and ensure that you fully understand the following *UML Class Diagram*. It describes a class and the features you need to implement for it:
    - The overall rectangle represents class *Message*
    - The top part has the name of the class
    - The middle part contains the things the object *knows*. These become the instance variables (or *state*) within the object, much like the fields of a record or struct. Class *Message* has one instance variable, named ***_text***, that stores a ***String*** object.
    - The lower part contains the things the object *can do*. These become *methods* (i.e., behavior) within the object, much like functions and procedures. Class *Message* has two methods, the first is a special method called *constructor* (constructors are named using the class name, here ***Message***) and the second is a ***Print*** method. We just specify the signature of message, not the behavior itself.

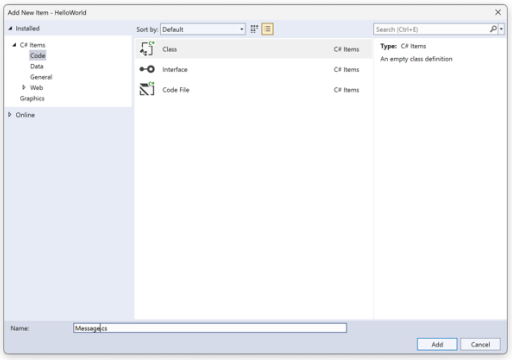


4. Create a new file for your C# class.

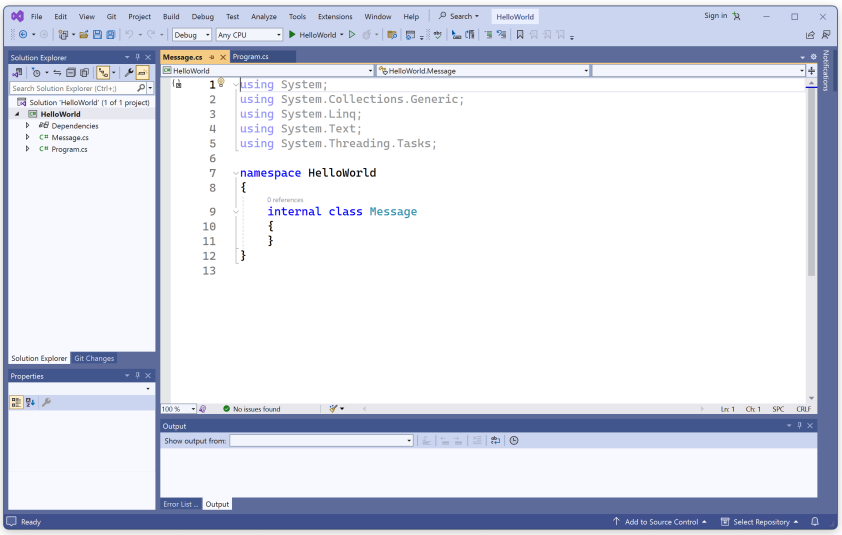
- Select project HelloWorld, right-click mouse and select Add/Class…



- Choose C# Items/Class and name the class file *Message.cs*.



- Visual Studio creates class ***Message*** and opens the associated file *Message.cs* in the editor.

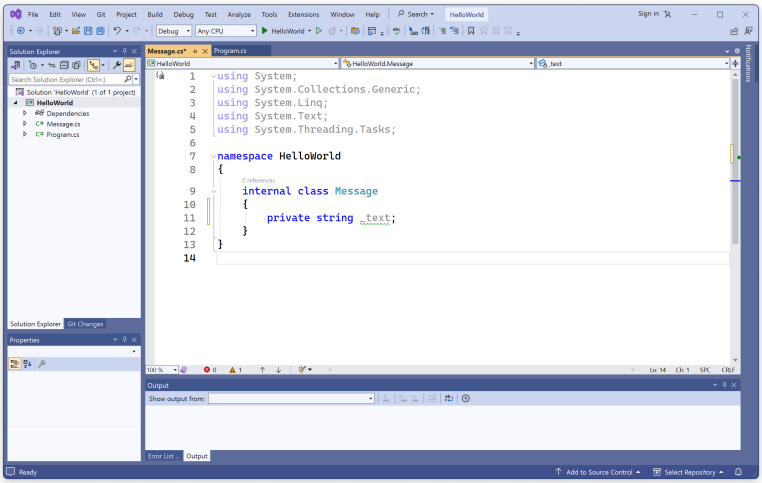


We have created a skeleton for class Message. It is marked internal to signify that this class is only accessible within the application "*Hello, World!*". This is the default and it works perfectly for this task and others in COS20007.

Next, we need to add the instance variable ***_text*** to store the *text* that the object *knows*. Note, the *scope of an instance variable* is its enclosing class. That is, all objects generated from the same class have access to the instance variables defined within the class.

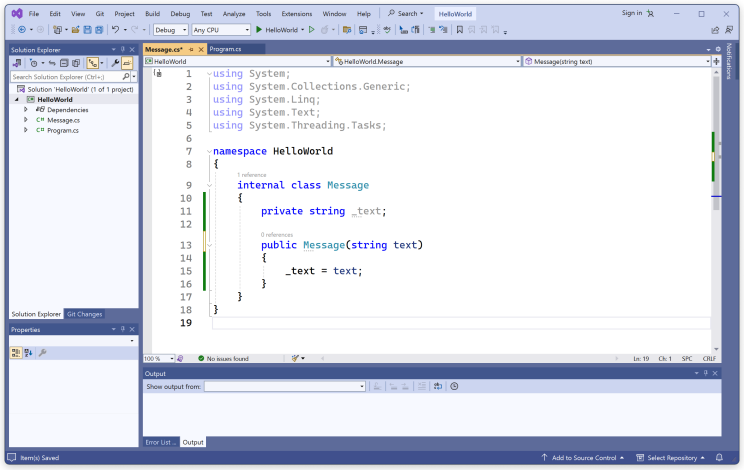
> **Tip**: Store the things the object knows (its instance variables) at the top of the class. This helps match the UML, and means it is easy to locate this when you need it.

5. Add instance variable ***_text*** the *Message* class. It should appear as shown below in your code. This tells the class *Message* that objects generated from it *know* a string that can be accessed via name ***_text***:



You may noticed a green wave underlining ***_text***. This is Visual Studio's way of telling you that ***_text*** is currently unused and that you have further options to control access to this instance variable. No actions need to be taken at this moment.

6. Add the constructor for *Message* objects. The constructor is called when we create new objects for a class. The purpose of a constructor is to initialize an object with sensible variables, either using constructor parameters or default initializers. The UML Diagram indicates that the constructor for class *Message* takes a string parameter. This parameter is used to initialize the instance variable ***_text***.

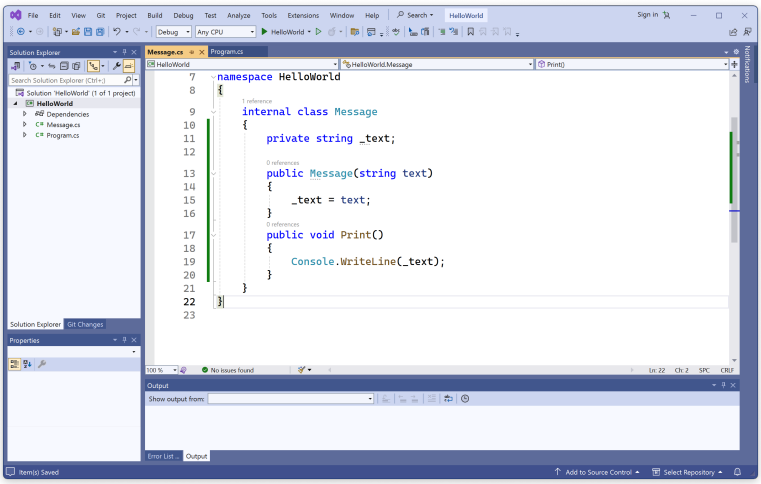


In C#, constructors unlike other methods cannot be called directly. It is the ***new*** operator that calls the constructor.

You may notice green dots underlining the name Message. This is Visual Studio's way of telling you that there may be an alternative way to initialize ***_text***. No actions need to be taken at this moment.

Note: Within the object's methods you can access the object's instance variables and other methods directly. Here ***_text*** refers to the *Message* object's instance variable ***_text***. The underscore is a naming convention we use in C# to indicate that an instance variable is private. You may have seen a similar convention being used in Python, for example.

7. Add the ***Print*** method to the *Message* class. ***Print*** uses the ***WriteLine*** method of class *Console* (which represents standard input/output operations) to output the value of instance variable ***_text***.



Tip: Picture a *Message* object as a capsule that contains a ***_text*** instance variable and a ***Print*** method. When you ask it to print, the object run's the steps inside the ***Print*** method. ***Print*** is inside the capsule so it can access the object's ***_text*** instance variable.

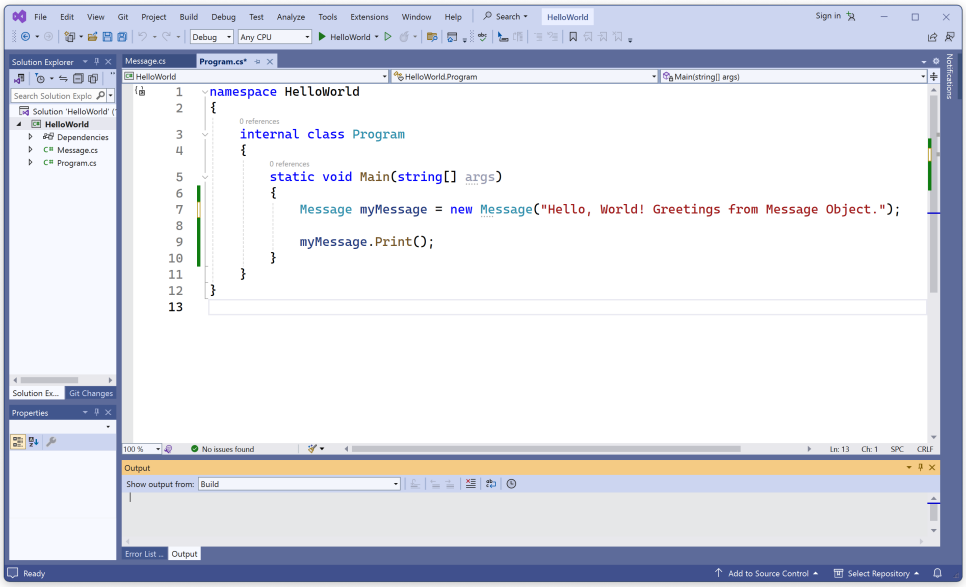
You have successfully created a *Message* class. Now, we can create new *Message* objects that can print messages (i.e., the value of instance variable ***_text***) to the Terminal.

8. Return to the *Program.cs* file.

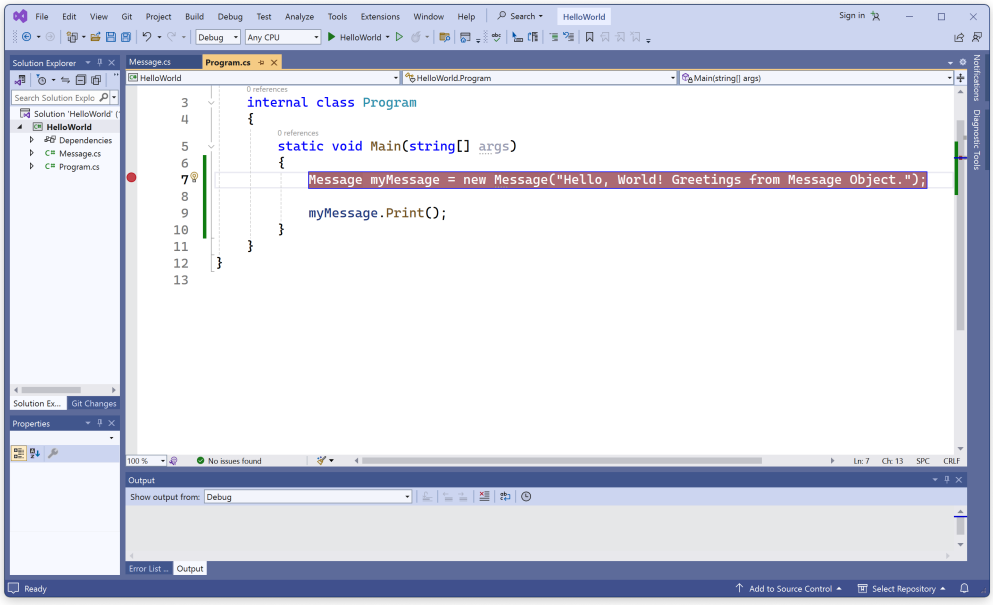
Note: Notice the program is run from a class *HelloWorld*, which is a C# class just like *Message*. However, it defines ***Main***, the main entry point of application "*Hello, World!*". Method ***Main*** is marked *static* that tells C# that this method is a class member function. Class members are shared by all objects of a class. We have already seen other class members. For example, ***WriteLine*** is a static member function of class *Console*.

The method ***Main*** has a special purpose. It is called at most once. It is the first method that is called and when ***Main*** finishes, the application terminates.

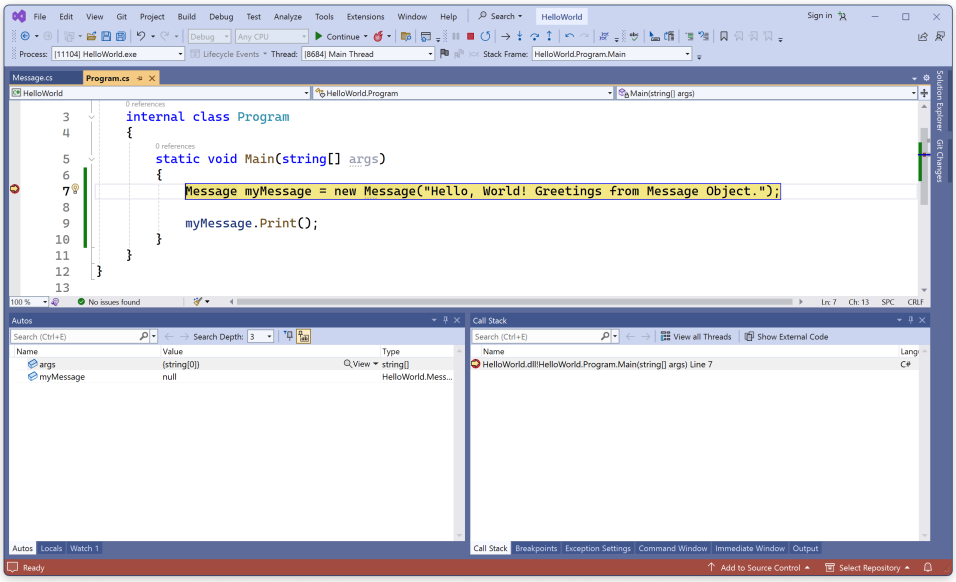
9. Inside the ***Main*** method, create a local *Message* object named ***myMessage*** and initialize it with the text "*Hello, World! Greetings from Message Object.* ***My Student ID is <xxx>***". Replace the "<xxx>" with your actual Student ID.
10. Ask the ***myMessage*** object to ***Print*** itself.
11. Delete the line Console.WriteLine("Hello, Word!"). The following screenshot shows the resulting structure method ***Main***:



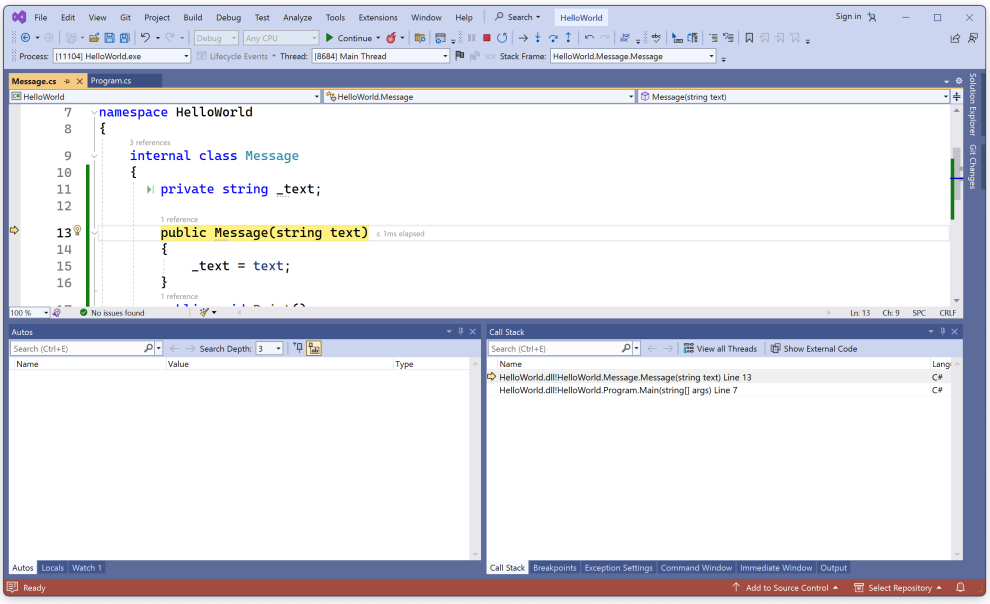
12. Run your program using Debug/Start Without Debugger or press the play button (▶) in the tool bar.
13. Now try the following features of the debugger:
    - Add a *breakpoint* by clicking in the gray margin next to the code that creates your ***myMessage*** object in ***Main*** (line 7). If you have clicked the correct location, then a red dot appears. A breakpoint tells the debugger to stop at this line (before the code is run).This allows you to inspect the program manually.



    - Start the program using Debug/Start Debugging or press the play button (▶) in the tool bar. The program should stop when it reaches the breakpoint. You should be able to see the *Call Stack* and the values of *Locals*. Watch the values of these change as the program runs. You can also hover over variables, or enter your own expressions to *Watch*.



- Press the *Step Into* button (or choose Debug/Step Into). This advances the program one statement at a time. You can also try stepping over and out of a method, and continuing when you no longer want to step. Using *Step Into* at this stage instructs the debugger to proceed to the start of the *Message* constructor.



You now have an object-oriented "*Hello, World!*" program.

14. Modify the program to have it ask users for their name and if the program recognizes the name it returns a predefined greeting. If the program does not know the name, it responds with a standard greeting.

Your program must support 3 (three) types of greetings for the following types of inputs:
A. Three names of your friends, or of your family, or from your favourite movies/novels (e.g. our "James Bond");
B. Name of the System Administrator (i.e. you); and
C. The entry of any unrecognised name (not one of the above).

Type A should have three personalised greetings similar to our example "Hi James Bond, how are you?". Type B should have the special greeting of "Welcome Admin". And finally Type C should have one standard greeting for an unknown name, i.e. "Welcome, nice to me you.".

Or you may alter the following pseudocode to allow for a single test run with an exit case (e.g. the user supplies the name of "Exit") or you keep processing names until "Exit" is provided. In which case there would only be one screenshot containing the five names and their greetings, followed by "Exit".

Tip: You can read a value into a string variable using Console.ReadLine().
Eg: `name = Console.ReadLine();`

See the following pseudocode for the above example. Change the example to use your own names and messages.

```
   Main()
 1: myMessage := new Message with text "Hello World…"
 2: Tell myMessage to Print
 3: let messages be a list of Message objects
 4: Add first greeting to messages
 5: Add second greeting to messages
 6: Add third greeting to messages
 7: Add fourth greeting to messages
 8: Add standard greeting to messages
 9: Tell Console to Write "Enter name: "
10: name := Tell Console to ReadLine
11: // test name with a sequence of 4 if-else statements
12: if Tell name to ToLower == "wilma"
13:     Tell messages[0] to Print
14: else if Tell name to ToLower == "fred"
15:     Tell messages[1] to Print
16: …
17: else
18:     // standard greeting
19:     Tell messages[4] to Print
```

Complete the program.

Once your program is complete you can prepare it for your portfolio. This can be placed in your portfolio as evidence of what you have learnt.

1. Review your code and ensure it is formatted correctly.
2. Run the program and use your preferred screenshot program to take a screenshot of the Terminal showing the program's output.
3. Insert a breakpoint within your program and run the debugger. Take a screenshot of the Visual Studio showing the call stack and the code paused within the *Print* method of the *Message* class.
4. Save and backup your work to multiple locations, if possible.
    - Once you your program is working you do not want to lose your work.
    - Work on your computer's storage device most of the time, but backup your work when you finish each task.
    - You may use a cloud storage provider to safely store your work.
    - USB and portable hard drives are good secondary backups, but there is a risk that the drive gets damaged or lost.

Assessment Criteria

Make sure that your task has the following in your submission:

- The "Universal Task Requirements" (see Canvas) have been met.
- Screenshot of running the debugger.
- Your program (as text or screenshot).
- Screenshot of output.