School of Science, Computing and Engineering Technologies

Object-Oriented Programming

Week 1: Preparing for Object-Oriented Programming

Overview

We have designed this unit assuming that you have already been exposed to some fundamental programming concepts. While you may not have been exposed to object-oriented programming specifically, we do expect that you have a solid grasp of fundamental programming concepts as acquired in COS10009.

Purpose:	Demonstrate that you have the pre-requisite knowledge required for successful completion of COS20007 – Object-Oriented Programming (OOP).
Non-assessable Tasks:	I. Install Visual Studio (VS) Code and NUnit test on your machine II. Demonstrate 'Hello world' console program with VSCode III. Discuss with your tutors about OOP Preparation Sheet IV. Review existing C# programming reference sheets in Week 0

Note: If you are unable to complete this task, you will struggle in this unit. Your tutor may be able assist you with suggestions on how increase your knowledge. You can also participate in HelpDesk sessions

I. Installations

1. VSCode Installation Guide
2. NUnit Installation Guide

II. 'Hello word' console program

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.

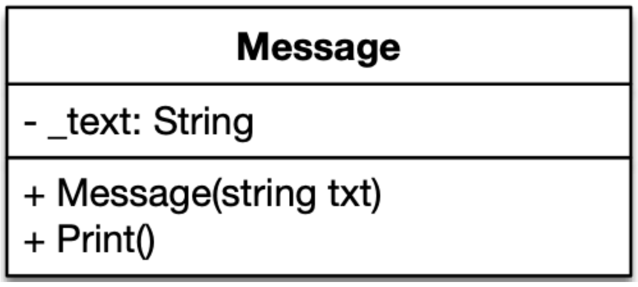
The "*Hello, World!*" program is technically not "object-oriented", however, it uses a class to pro-duce output, namely *Console* which represents standard input/output operations.

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 per-forming.

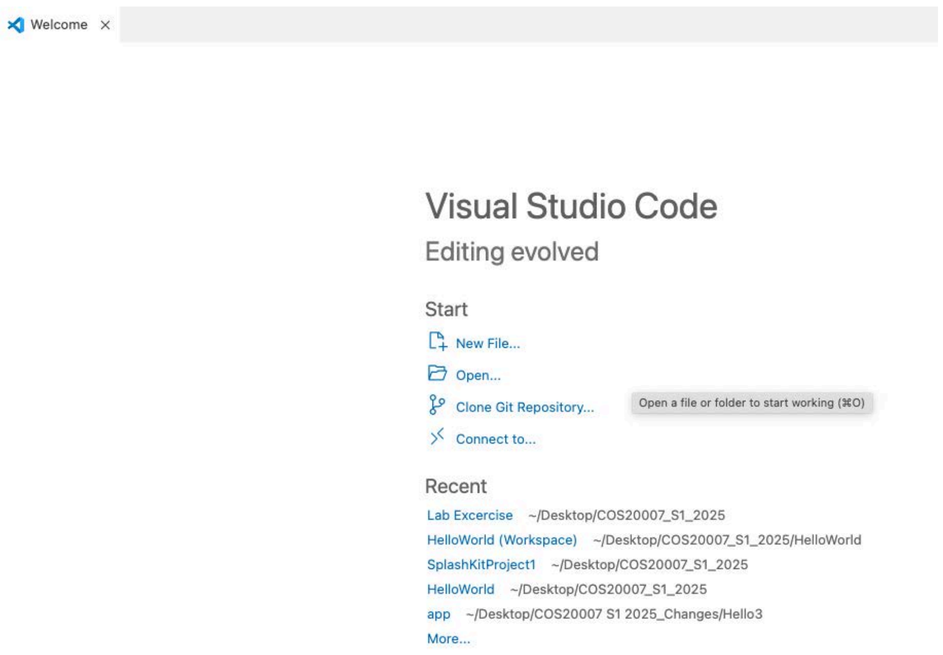
In C#, objects are created from classes. Classes provide initial values for instance variables and the bodies for methods. All objects generated from the same class share the same methods, but contain separate copies of the instance variables. New objects are created from a class by ap-plying the ***new***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

1. Read the UML Class Diagrams Tutorial by Robert C. Martin and ensure that you fully un-derstand 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. Open Visual Studio Code.
   - Select Open… to browse to an existing local folder or creating a new folder in your local machine



Welcome

Visual Studio Code

Editing evolved

Start

New File...

Open...

Clone Git Repository...

Open a file or folder to start working (⌘O)

Connect to...

Recent

Lab Excercise ~/Desktop/COS20007_S1_2025

HelloWorld (Workspace) ~/Desktop/COS20007_S1_2025/HelloWorld

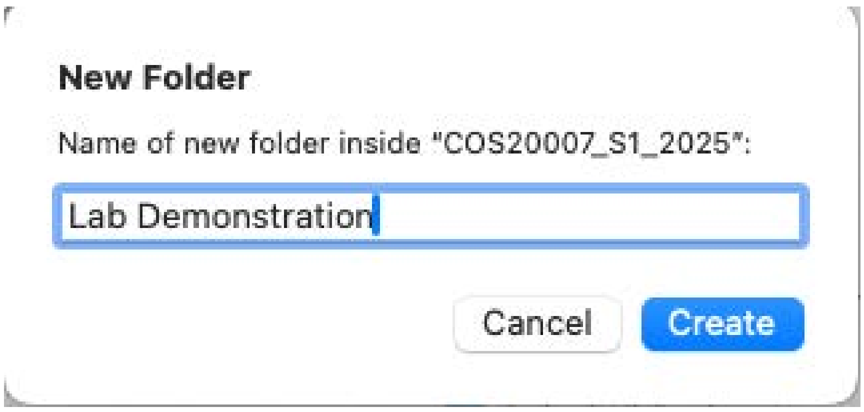
SplashKitProject1 ~/Desktop/COS20007_S1_2025

HelloWorld ~/Desktop/COS20007_S1_2025

app ~/Desktop/COS20007 S1 2025_Changes/Hello3

More...

   - Select Create a new folder… For example, name a folder as 'Lab Demonstration'



New Folder

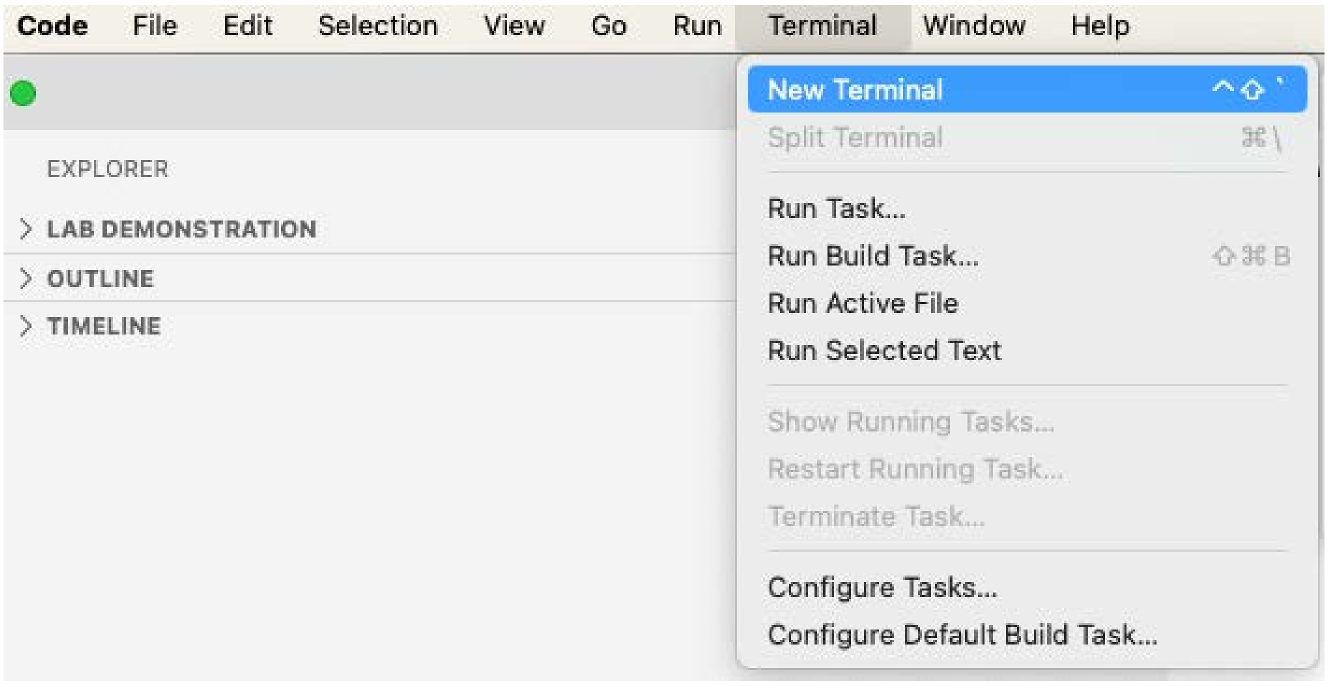
Name of new folder inside "COS20007_S1_2025":

Lab Demonstration

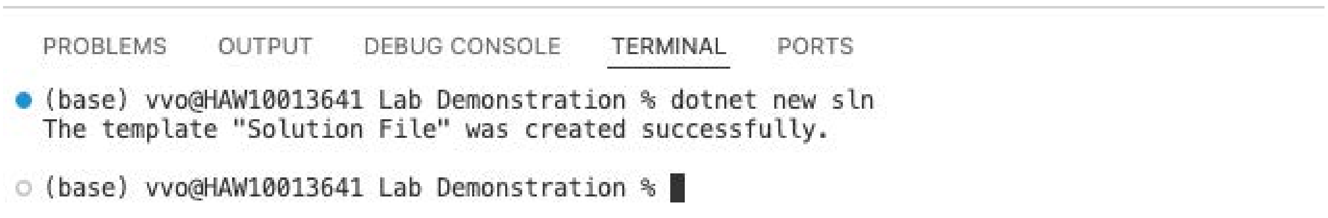
Cancel Create

- Select Create a new folder… For example, name a folder as 'Lab Demonstration'

- Under the 'Explorer window' in Visual Studio Code, you can see the folder 'Lab Demonstration'

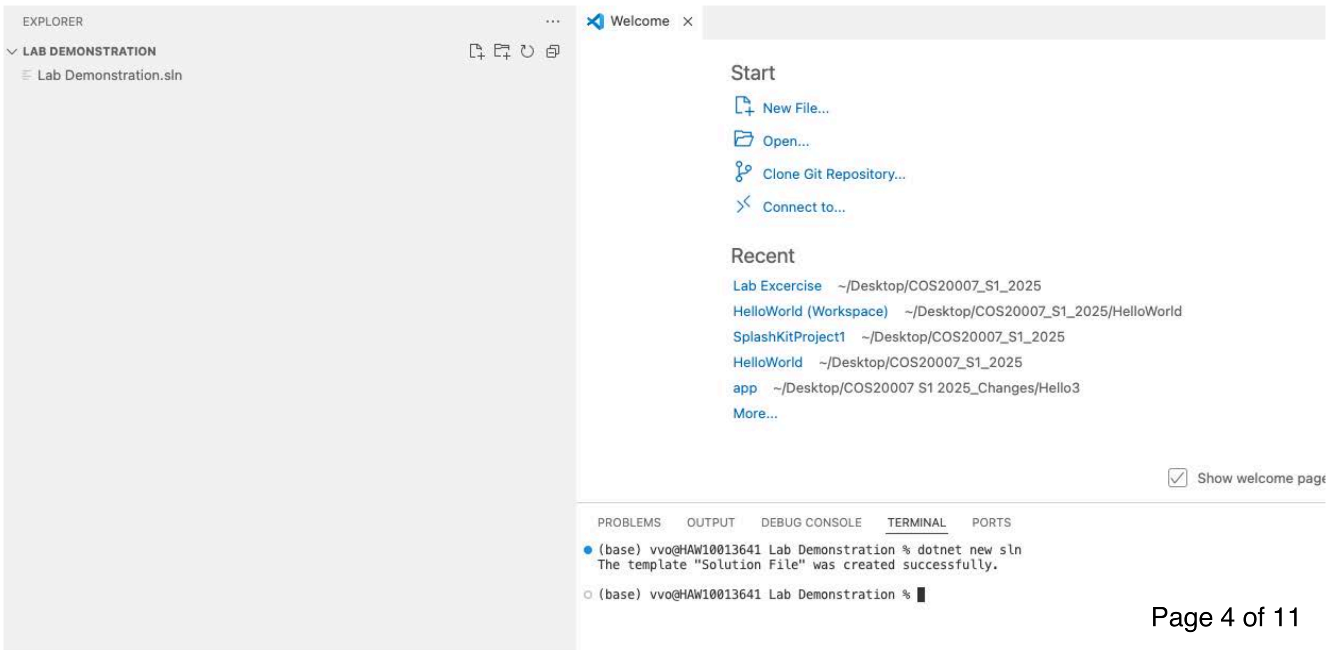


- From the Menu Bar, select Terminal>New Terminal. This will open a new console terminal at your current working directory, i.e., Lab Demonstration



```
(base) vvo@HAW10013641 Lab Demonstration % dotnet new sln
The template "Solution File" was created successfully.

(base) vvo@HAW10013641 Lab Demonstration %
```

- From the terminal, you can type pwd to retrieve the full path of your current working directory. Then, type dotnet new sln in the terminal as in the below figure

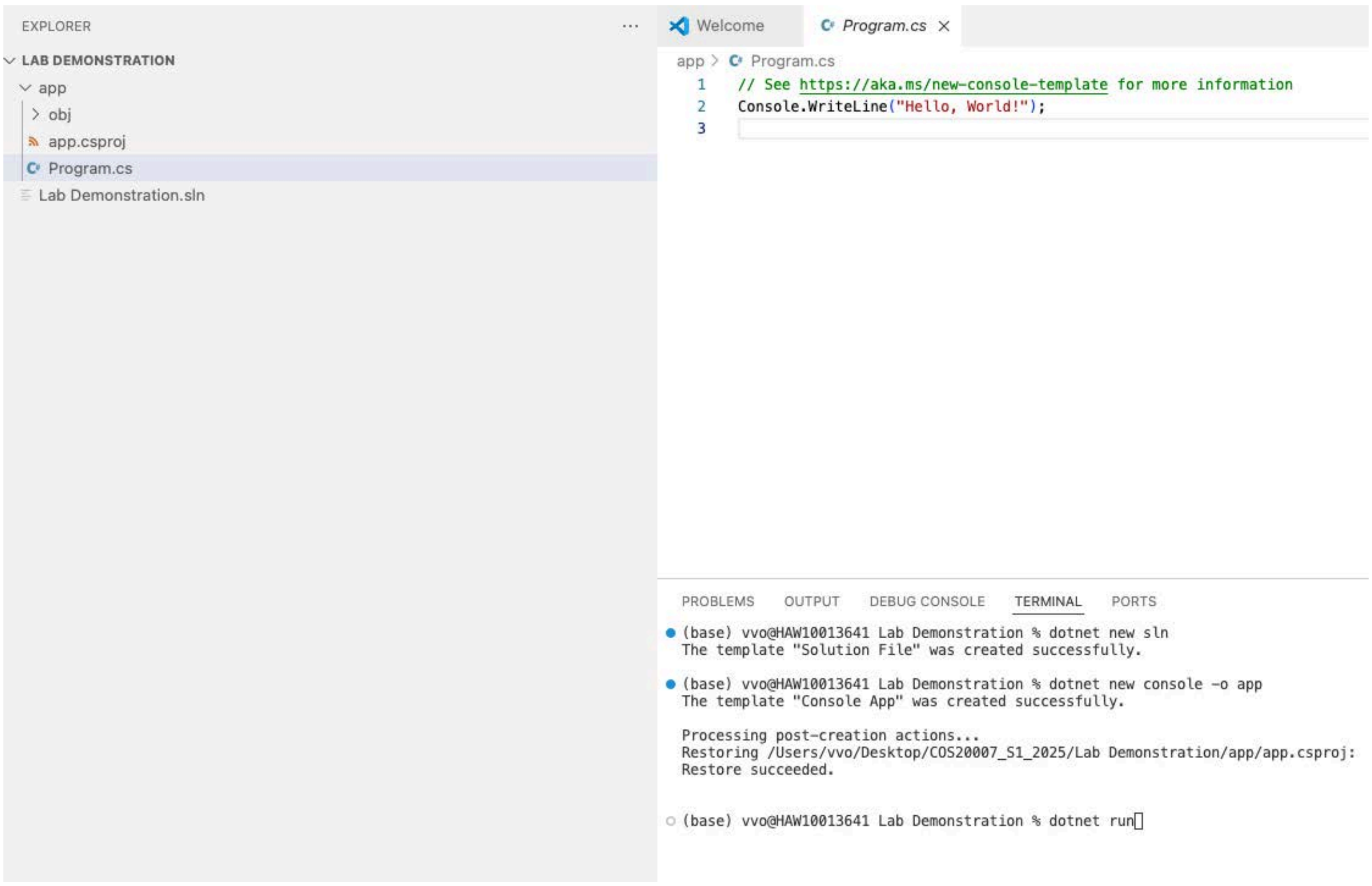

EXPLORER
LAB DEMONSTRATION
Lab Demonstration.sln
Welcome
Start
New File...
Open...
Clone Git Repository...
Connect to...
Recent
Lab Excercise ~/Desktop/COS20007_S1_2025
HelloWorld (Workspace) ~/Desktop/COS20007_S1_2025/HelloWorld
SplashKitProject1 ~/Desktop/COS20007_S1_2025
HelloWorld ~/Desktop/COS20007_S1_2025
app ~/Desktop/COS20007 S1 2025_Changes/Hello3
More...
Show welcome page
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) vvo@HAW10013641 Lab Demonstration % dotnet new sln
The template "Solution File" was created successfully.

(base) vvo@HAW10013641 Lab Demonstration %
```

- The command dotnet new sln creates a .NET solution file in your current working directory. If you have a single solution file (.sln file) in the workspace, the Solution Explorer will detect that file and automatically load it after the workspace is loaded. The file is a structure for organizing projects linked to this .sln file. See this link for more details. https://code.visualstudio.com/docs/csharp/project-management

- For simplicity, we may have multiple projects under the same solution. For example, you may have a developing project and a test project under the same solution. The test project can be developed with NUnite test to test the functionalities of the developing project.

- For example, we can develop a developing project called app, by typing following command in the console dotnet new console -o app . The command will create a new console application with such output filename under your solution folder.



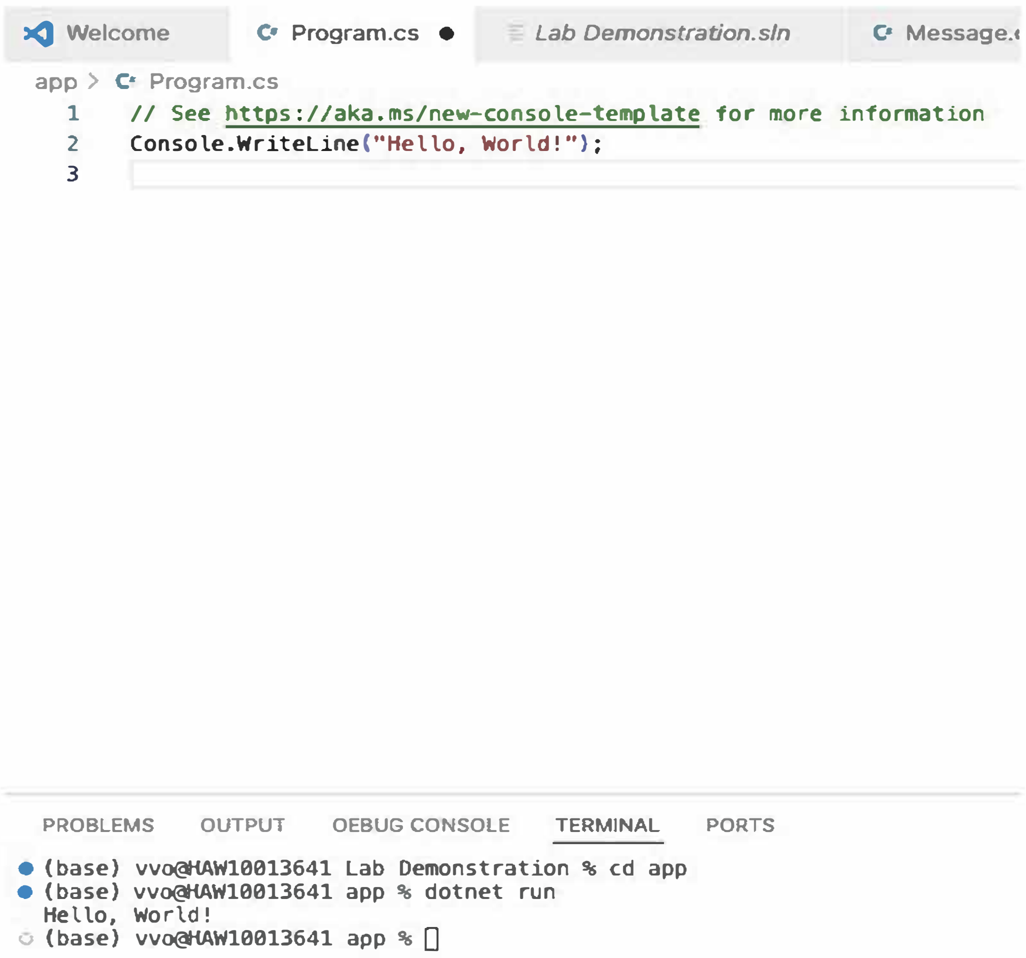
```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

```
(base) vvo@HAW10013641 Lab Demonstration % dotnet new sln
The template "Solution File" was created successfully.

(base) vvo@HAW10013641 Lab Demonstration % dotnet new console -o app
The template "Console App" was created successfully.

Processing post-creation actions...
Restoring /Users/vvo/Desktop/COS20007_S1_2025/Lab Demonstration/app/app.csproj:
Restore succeeded.


(base) vvo@HAW10013641 Lab Demonstration % dotnet run
```

- From the Solution Explorer window, you will see the 'app' folder with an 'obj' subfolder (will contain compiled objects after execution), an app.csproj (C sharp project configuration and description file), and the main Program.cs file.

- From the terminal, you can use the Unix command cd app to navigate to the app folder inside the folder Lab Demonstration.

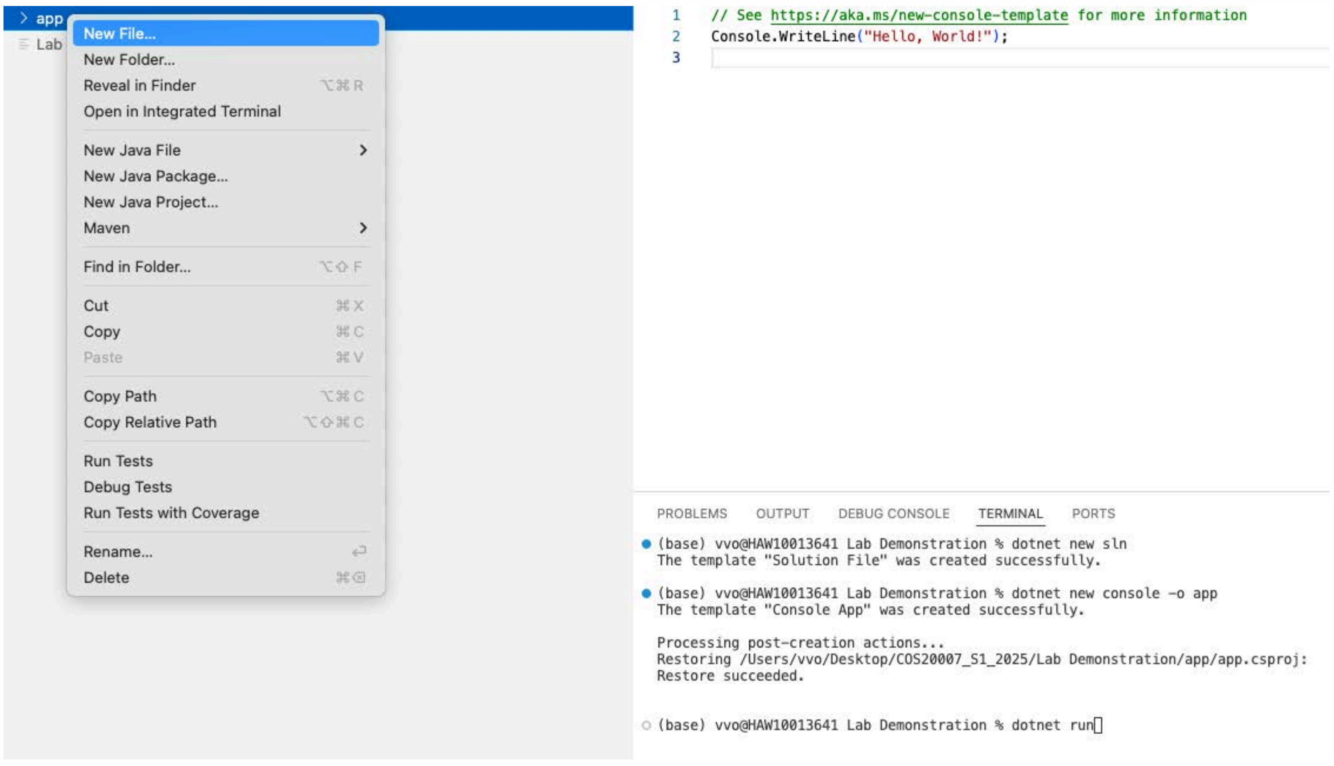
- See the screenshot next page after typing 'the command "dotnet run'



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

```
(base) vvo@HAW10013641 Lab Demonstration % cd app
(base) vvo@HAW10013641 app % dotnet run
Hello, World!
(base) vvo@HAW10013641 app % 
```

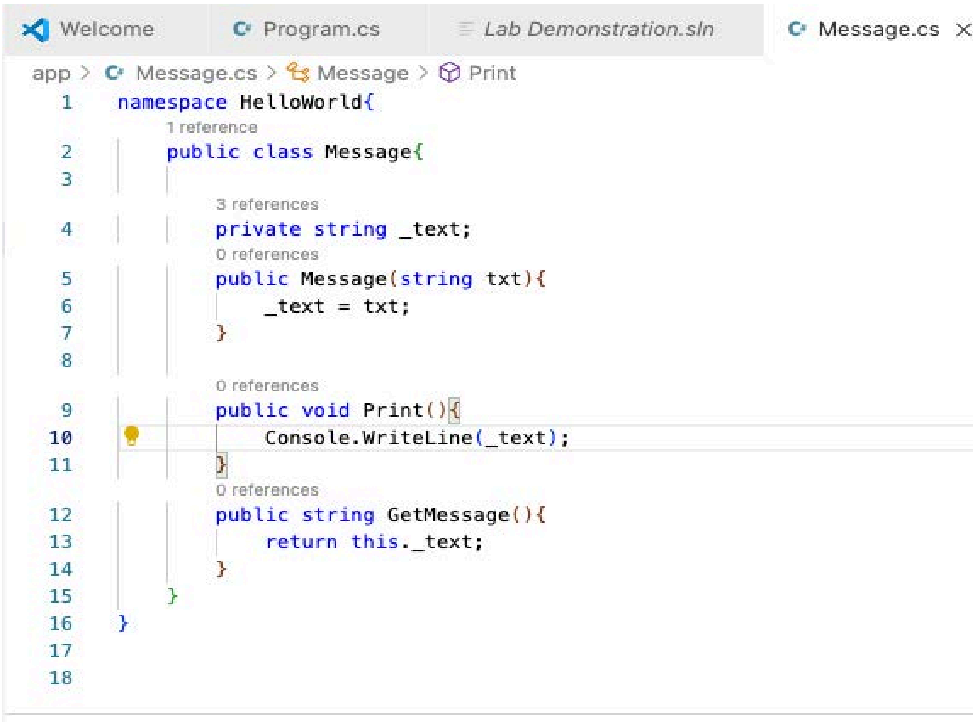
- From the Solution Explorer window, **right click on the 'app' folder** to create another Message.cs file within this folder.



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

```
(base) vvo@HAW10013641 Lab Demonstration % dotnet new sln
The template "Solution File" was created successfully.

(base) vvo@HAW10013641 Lab Demonstration % dotnet new console -o app
The template "Console App" was created successfully.

Processing post-creation actions...
Restoring /Users/vvo/Desktop/COS20007_S1_2025/Lab Demonstration/app/app.csproj:
Restore succeeded.

(base) vvo@HAW10013641 Lab Demonstration % dotnet run
```

- Copy the following code into the **Message.cs** file



```
1   namespace HelloWorld{
        1 reference
2       public class Message{
3
            3 references
4           private string _text;
            0 references
5           public Message(string txt){
6               _text = txt;
7           }
8
            0 references
9           public void Print(){
10              Console.WriteLine(_text);
11          }
            0 references
12          public string GetMessage(){
13              return this._text;
14          }
15      }
16  }
17
18
```

  - **Line 1 create a** namespace **Hellworld.** The namespace keyword is used to declare a scope that can contain a set of related objects. **Lines 5-7** in the above figure is the constructor implementation 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 vari-ables, either using constructor parameters or default initializers. The UML Diagram (see page 3) 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.

  - **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.

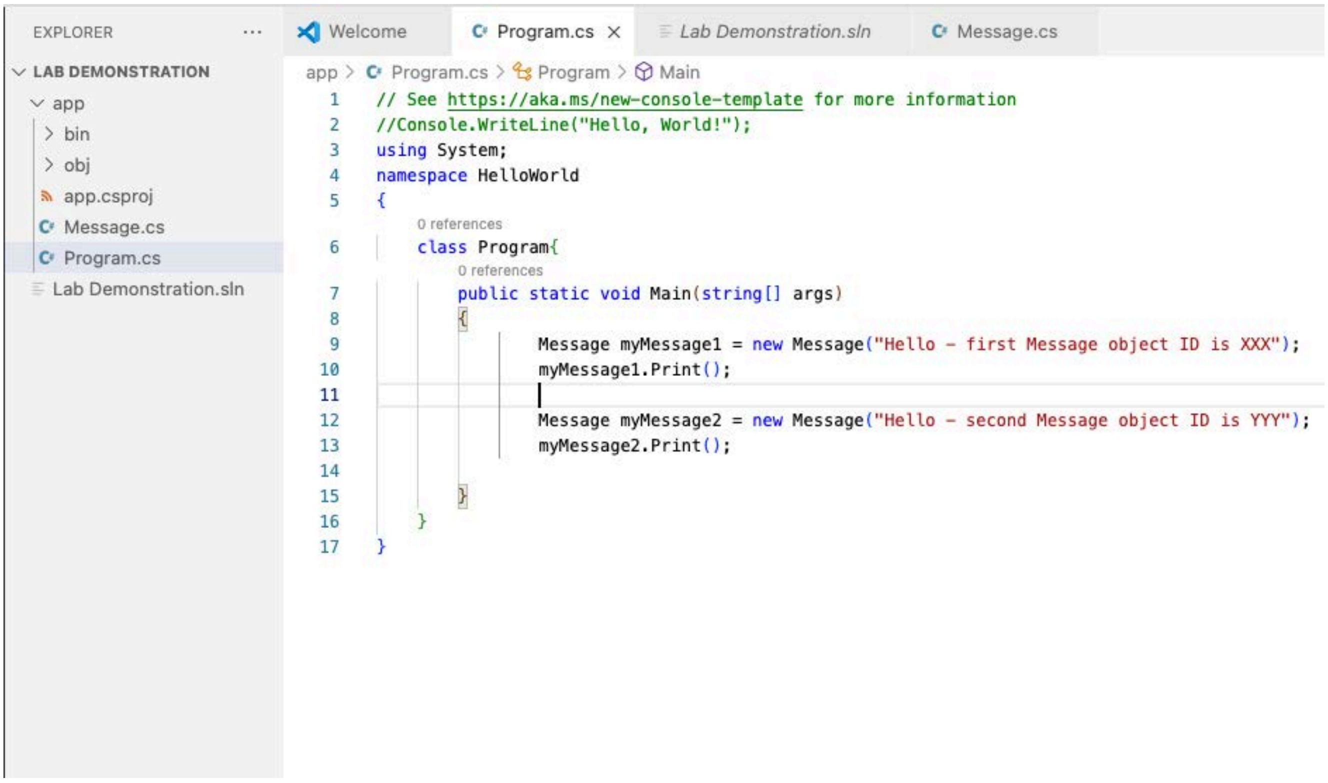
  - **Lines 9-11** in the above figure is the implementation of the Print method to the Message class. Print uses the WriteLine method of class *Console* (which represents standard input/output operations) to output the value of instance variable ***_text***.

  - **Lines 12-14** is a Get property to return the value of instance variable _text. Noting that line 12 has a return datatype 'string' which matches to the data type of the variable (see line 4).

  - **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.

- Return to the *Program.cs* file and modify the code as below.



```
// See https://aka.ms/new-console-template for more information
//Console.WriteLine("Hello, World!");
using System;
namespace HelloWorld
{
    class Program{
        public static void Main(string[] args)
        {
            Message myMessage1 = new Message("Hello - first Message object ID is XXX");
            myMessage1.Print();

            Message myMessage2 = new Message("Hello - second Message object ID is YYY");
            myMessage2.Print();

        }
    }
}
```

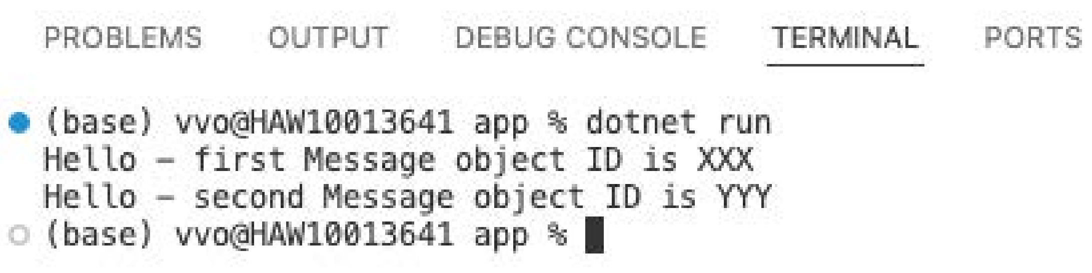
- Comment the first two lines using the syntax //.

- Define the namescape 'Helloworld' as used in the Message.cs file

- In the class ***Program***, we define Main, the main entry point of application “app”. 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.

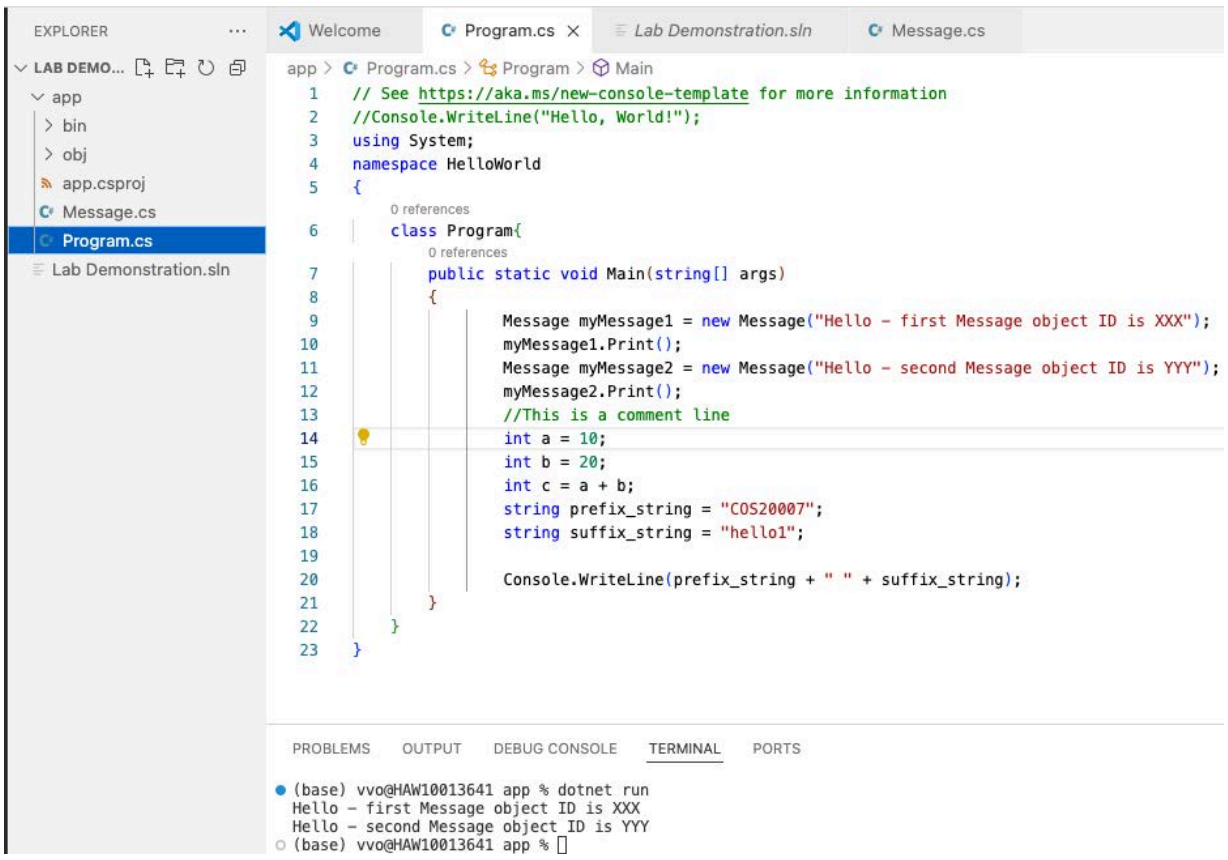
- The method Main has a special purpose. It is called at most once. It is the first method that is called and when Main finishes, the application terminates.

- Inside the ***Main*** method, create a local *Message* object named ***myMessage1*** and initialize it with the text “*Hello - first Message object* ***ID is <xxx>***”. Replace the "<xxx>" with your actual Student ID. You can create a local Message object named ***myMessage2.*** Bothe the two objects are instantiated from the same Message class.
- Ask the ***myMessage1*** object to ***Print*** itself.
- Ask the ***myMessage2*** object to ***Print*** itself.
- See the result from Terminal as below after using the command 'dotnet run'



```
● (base) vvo@HAW10013641 app % dotnet run
  Hello - first Message object ID is XXX
  Hello - second Message object ID is YYY
○ (base) vvo@HAW10013641 app % 
```

- Now, modify the Program.cs as below to exercise other data types like int, float, and string.

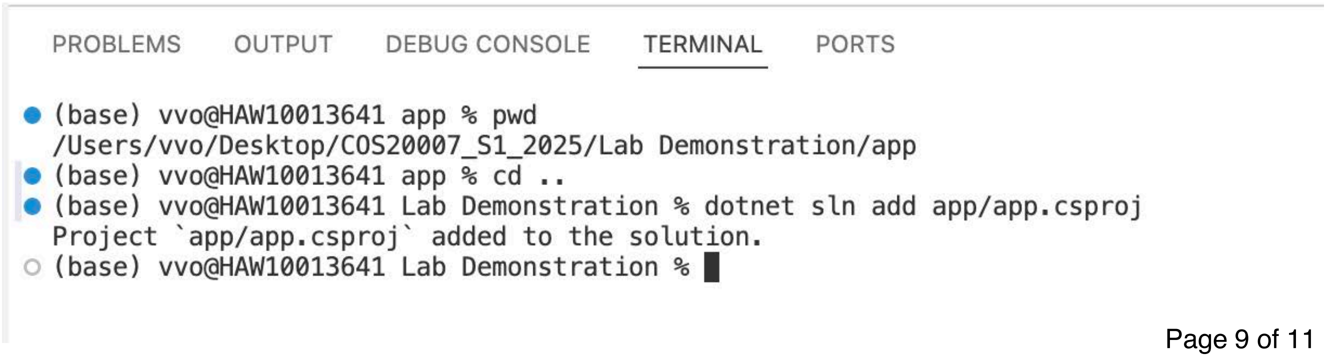


```csharp
// See https://aka.ms/new-console-template for more information
//Console.WriteLine("Hello, World!");
using System;
namespace HelloWorld
{
    class Program{
        public static void Main(string[] args)
        {
            Message myMessage1 = new Message("Hello - first Message object ID is XXX");
            myMessage1.Print();
            Message myMessage2 = new Message("Hello - second Message object ID is YYY");
            myMessage2.Print();
            //This is a comment line
            int a = 10;
            int b = 20;
            int c = a + b;
            string prefix_string = "COS20007";
            string suffix_string = "hello1";

            Console.WriteLine(prefix_string + " " + suffix_string);
        }
    }
}
```

```
● (base) vvo@HAW10013641 app % dotnet run
  Hello - first Message object ID is XXX
  Hello - second Message object ID is YYY
○ (base) vvo@HAW10013641 app % 
```

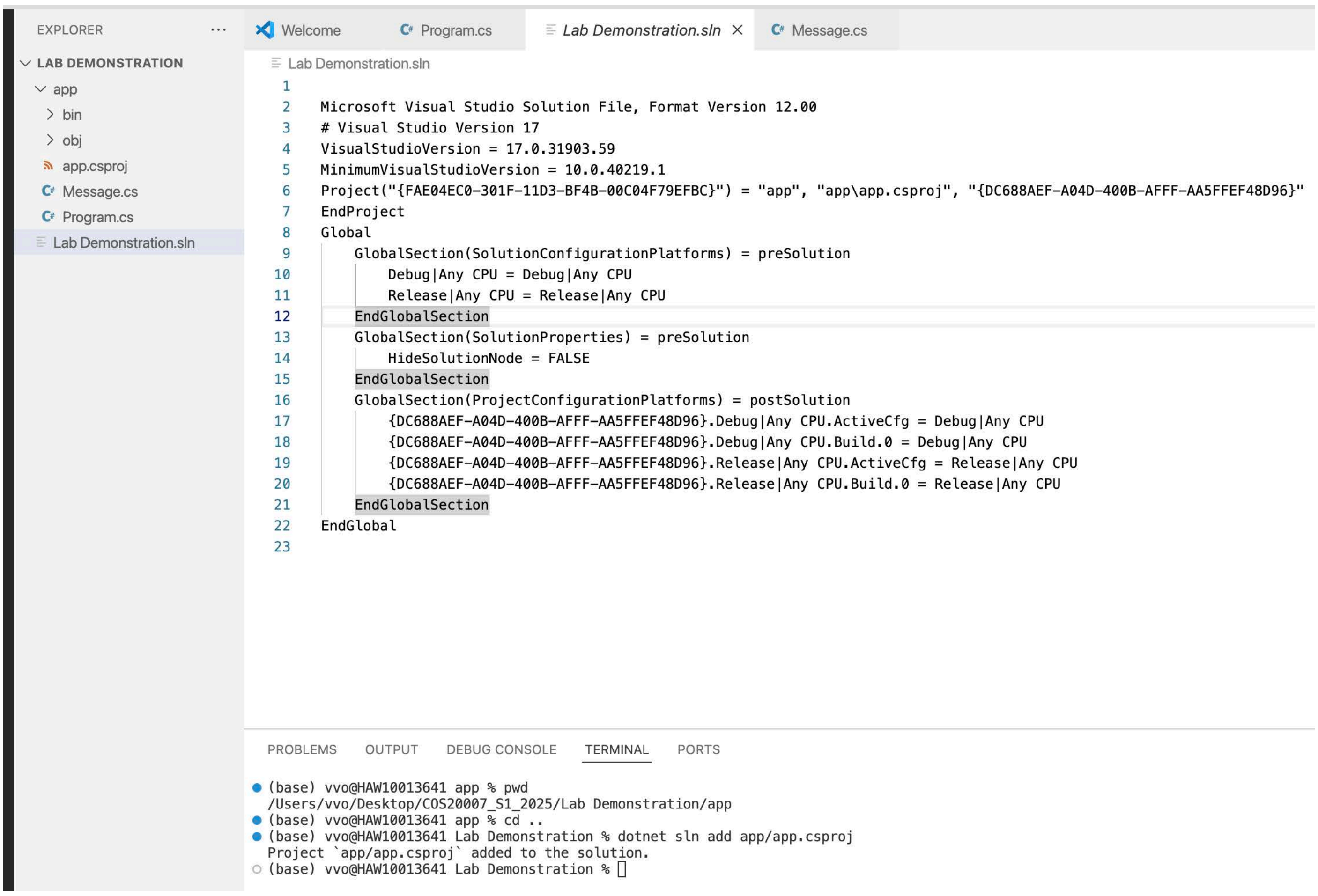
- The current project 'app' have not been linked into the 'Lab Demonstration.sln' yet. This can create compiling errors if the solution contains multiple projects inside. The best practice is to add the project into the sln using the commands as in the below figure. Assuming you are at the 'app' working directory in the terminal (can be verified by using the command 'pwd', please type cd .. to move back to the parent folder (the double dots are important).

- Then, type dotnet sln add app/app.csproj. This will add the app project into the solution. Next time, when you open the Lab Demonstration folder, the app project will be automatically linked.



```
● (base) vvo@HAW10013641 app % pwd
  /Users/vvo/Desktop/COS20007_S1_2025/Lab Demonstration/app
● (base) vvo@HAW10013641 app % cd ..
● (base) vvo@HAW10013641 Lab Demonstration % dotnet sln add app/app.csproj
  Project `app/app.csproj` added to the solution.
○ (base) vvo@HAW10013641 Lab Demonstration % 
```

- Open the Lab Demonstration.sln with Visual Studio Code, you can see the app project now has been added as in the below figure



```

Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio Version 17
VisualStudioVersion = 17.0.31903.59
MinimumVisualStudioVersion = 10.0.40219.1
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "app", "app\app.csproj", "{DC688AEF-A04D-400B-AFFF-AA5FFEF48D96}"
EndProject
Global
    GlobalSection(SolutionConfigurationPlatforms) = preSolution
        Debug|Any CPU = Debug|Any CPU
        Release|Any CPU = Release|Any CPU
    EndGlobalSection
    GlobalSection(SolutionProperties) = preSolution
        HideSolutionNode = FALSE
    EndGlobalSection
    GlobalSection(ProjectConfigurationPlatforms) = postSolution
        {DC688AEF-A04D-400B-AFFF-AA5FFEF48D96}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
        {DC688AEF-A04D-400B-AFFF-AA5FFEF48D96}.Debug|Any CPU.Build.0 = Debug|Any CPU
        {DC688AEF-A04D-400B-AFFF-AA5FFEF48D96}.Release|Any CPU.ActiveCfg = Release|Any CPU
        {DC688AEF-A04D-400B-AFFF-AA5FFEF48D96}.Release|Any CPU.Build.0 = Release|Any CPU
    EndGlobalSection
EndGlobal

```

```
(base) vvo@HAW10013641 app % pwd
/Users/vvo/Desktop/COS20007_S1_2025/Lab Demonstration/app
(base) vvo@HAW10013641 app % cd ..
(base) vvo@HAW10013641 Lab Demonstration % dotnet sln add app/app.csproj
Project `app/app.csproj` added to the solution.
(base) vvo@HAW10013641 Lab Demonstration %
```

Extra activity at home.

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()
myMessage := new Message with text "Hello World…"
Tell myMessage to Print
let messages be a list of Message objects
Add first greeting to messages
Add second greeting to messages
Add third greeting to messages
Add fourth greeting to messages
Add standard greeting to messages
Tell Console to Write "Enter name: "
name := Tell Console to ReadLine
// test name with a sequence of 4 if-else statements
if Tell name to ToLower == "wilma"
    Tell messages[0] to Print
else if Tell name to ToLower == "fred"
    Tell messages[1] to Print
…
else
    // standard greeting
    Tell messages[4] to Print
```