SIT771 Object Oriented Development

Pass Task 3.1: Name Tester

Overview

In this task you will create a simple name testing program in order to explore the different kinds of control flow statements. The program will have a small menu to allow the user to choose between several options: testing a name, guess that number, and quit. The name tester will read in the user's name, and output a custom message. Guess that number will ask the user to guess a number between 1 and 100, letting them know if their guess was smaller or larger than their goal.

The material in Course 2, Week 1 will help you with this task.

This task has lots of details, but by the end you will have worked through the use of each kind of control flow structure. We have tried to add guidance so that you will see how to put this program together.

Submission Details

Submit the following files to Doubtfire.

- The Program code (*Program.cs*)
- A screenshot of the running program

You want to focus on the different control flow statements, and think about how they work when you run your program. The program will be built in multiple steps so that you can see each part working as you go.

Instructions

We are going to build this program over a number of steps. This is always a good idea, and in this case it will let us focus on the different control flow statements one at a time.

The following UML class diagram shows the class and enumeration that we will build in this task.

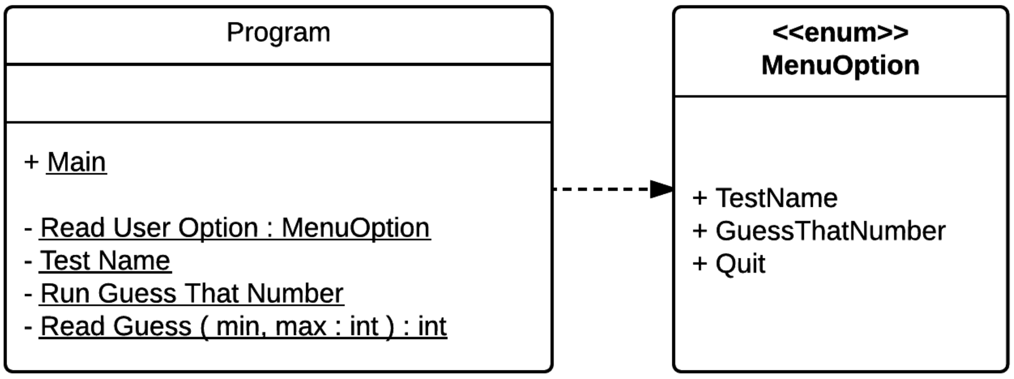

Figure: UML class diagram for the name tester program

Reading a Menu options

To get started lets show a menu to the user and read the option they want to perform. This program will use the Terminal to interact with the user, so we can use `Console.WriteLine` and `Console.ReadLine` to show and read values from the user.

1. Create a new project folder named `NameTester`, and use `dotnet` to create and restore the settings in this folder.
2. Open the project/folder in Visual Studio Code.
3. Open the **Program.cs** file.
4. Create a new **enumeration** called `MenuOption`, with the options `TestName`, `GuessThatNumber`, and `Quit`.

   Place this outside of the Program class, either before or after the Program class code. Keep the options in that order, this way we know that `TestName` will map to the integer value 0, `GuessThatNumber` will map to 1, and `Quit` will map to 2.

   Here is a start for this enum.

   ```
   public enum MenuOption
   {
       TestName,
       //... continue here
   }
   ```

5. Create a new `ReadUserOption` method that returns a `MenuOption` **inside** the **Program** class.

   This method will show the menu to the user and read in their selection.

   The `ReadUserOption` method should perform the following steps. These are expressed here as *pseudocode* (which is code like text, often used to express how an algorithm should work). Remember to add the `static` keyword to this method, as we will run it directly on the `Program` class itself.

   ```
   private static MenuOption ReadUserOption()
   {
       // steps go here...
   }
   ```

   1. Start the method by declaring an `option` integer variable. We will use this to store the value of the option the user selects.

   2. Use `Console.WriteLine` to output a menu showing the three options. "1 will run Test Name, 2 will play Guess That Number, and 3 will Quit". Add a header and use "*" or "-" to put borders around things to make them look a little nicer.

   3. Start a **do ... while** loop...

      1. Use `Console.Write` to show the user a prompt, something like "Choose an option [1-3]: "

      2. Use `Console.ReadLine` and `Convert.ToInt32` to read in the user's selection and convert it to an integer. Store this in a variable for later use.

   4. End the **do ... while** loop, having the above code loop **while** the value read in by the user is less than 1 or larger than 3.

      In this way we can force the user to choose a valid option. We ask them to enter the value, then loop while it is not a valid option.

   5. Lastly, we can return the matching `MenuOption`. We can make use of our understanding of enumerations, and the fact that each enumeration value is actually an integer. If you named the variable *option*, then the following code will return the matching MenuOption value.

      ```
      return (MenuOption)(option - 1);
      ```

      This works because each of the enumeration labels is associated with an integer value internally. When option is 1, the user wanted to `TestName`. This is the first value of the enumeration, so it has the integer value 0. Subtracting 1 from the value the user entered will convert the value they entered into the matching enumeration value.

      The `(MenuOption)` code is a **cast**. This tells the computer to re-interpret the value `(option - 1)` as a `MenuOption` value. This converts the `0` to

`MenuOption.TestName` .

Finally, the result of this cast can then be returned to the caller. They will get `MenuOption.TestName` if the user entered 1, `MenuOption.GuessThatNumber` if they entered 2, or `MenuOption.Quit` if they entered 3.

6. Change `Main` to have the following code.

```
public static void Main()
{
    MenuOption userSelection;

    userSelection = ReadUserOption();

    Console.WriteLine(userSelection);
}
```

7. Make sure everything is saved, then switch to the Terminal and build and run your program.

   To test this, try entering a value that less than 1 and a value larger than 3. Its good to test the next larger/smaller number to make sure you got it right. It is common to have an *off by one* type issue.

Think about how the **do while** loop is allowing you to get this behaviour. Why do you think we chose a post test loop here?

Actioning the Menu

Now that the user can enter an option from the menu, lets get the program to respond to that selection.

1. Switch back into VS Code, and open the **Program.cs** file.
2. We can now edit the `Main` method, and have it respond to the option the user selected. Locate the `Main` method.
3. Add in a `switch` statement, with a `case` for each of the `MenuOption` values. For the moment, just add a message indicating the option.

   Here is a start:

```
switch(userSelection)
{
    case MenuOption.TestName:
        Console.WriteLine("Test Name...");
        break;
    // ...
}
```

4. Switch to the Terminal and build and run your program.

   Check that you get different messages for each menu option.

5. Now lets add a loop to `Main` so that these instructions are repeated.

   - Use a **do while** loop.

- Start the loop after declaring the `userSelection` variable, but before assigning it a value.
- Have the while after the end of the switch.
- Remember to adjust your indentation!

```
MenuOption userSelection;

do
{
    userSelection = ReadUserOption();
    switch(userSelection)
    {
        case MenuOption.TestName:
            Console.WriteLine("Test Name...");
            break;
        // ...
    }
} while (userSelection != MenuOption.Quit);
```

6. Switch to the Terminal and build and run your program.

   Check that you can run the instructions until the user chooses to quit.

At this point focus on the **switch** statement and think about how the computer executes these instructions to get this output.

Testing the Name

Now lets play with the **if statement** and use this to implement the Test Name functionality.

1. In the `Program` class, create a new `TestName` static method. This method will read in the user's name, then output one of a number of custom messages.
2. In `TestName` :

   1. Declare a variable to accept the user's name.
   2. Use WriteLine to output a message asking them to enter their name.
   3. For the moment, just output the message "Hello " with the user's name.

3. Locate the `Main` method, and the `MenuOption.TestName` case of the `switch` statement.
4. Change this case to call your new `TestName` method. For example:

```
switch(userSelection)
{
    case MenuOption.TestName:
        TestName();
        break;
    // ...
}
```

5. Build and run your program.

   Check that at this point you can choose to test a name, and that this reads and outputs the appropriate details.

6. Now lets add custom messages for some different names. Switch back to the `TestName` method. After writing out the hello message add an if statement that will check if you are the user. Have it output a custom method if it is your name. For example, I would use the following:

   ```
   if ( name == "Andrew" )
   {
       Console.WriteLine("Welcome my creator!");
   }
   ```

7. Build and run your program, and test the new `TestName` functionality. Try your name, another name, but then also try your name with different cases, for example `andrew` or `anDreW`.

8. To allow the program to select your name, regardless of the case, we can ask the `String` to give you a lowercase version. For example:

   ```
   if ( name.ToLower() == "andrew" )
   {
       Console.WriteLine("Welcome my creator!");
   }
   ```

   Notice that we also need to change the string we are testing. It also needs to be lower case.

9. Test this again, and check it now works regardless of the case you use in your name.

10. Now lets add a message for other names. Firstly, we can add an `else` branch, add a standard message for uses that do not have a name that matches one of the custom messages (so anything other than your name at this stage). The structure will be something like this:

    ```
    if ( name.ToLower() == "andrew" )
    {
        //...
    }
    else
    {
        //...
    }
    ```

11. Test that you get the correct messages at this stage.

12. Now lets add a test for another name. In this case, check that the name matches one of your friends. Add a custom message for that user.

    To do this you need to add an **else** branch, and have within this have a single **if statement**. See the following example.

```
if ( name.ToLower() == "andrew" )
{
    //...
}
else
{
    if ( name.ToLower() == "jake" )
    {
        //...
    }
    else
    {
        //...
    }
}
```

Now, because it is common to have this kind of **if...else if ... else ...** statement, nesting these as shown above would be cumbersome. As a result, it is common to lay this out as following. This is the format you would use for these *if...else if ... else ...* blocks.

```
if ( name.ToLower() == "andrew" )
{
    //...
}
else if ( name.ToLower() == "jake" )
{
    //...
}
// more else if ( ) { ... } here
else
{
    //...
}
```

13. Add at least one other name test, and check that the name testing part of the program is working correctly.

At this point think about how the **if** statement works, and how the **else** branch can be used to provide different options. In this instance, you could use a **case statement**, but leave this with the if statements as we want to focus on that at this stage.

Guess that Number

Lets now wrap this up with a *Guess that Number* game. Here is a reminder of the structure for the program.

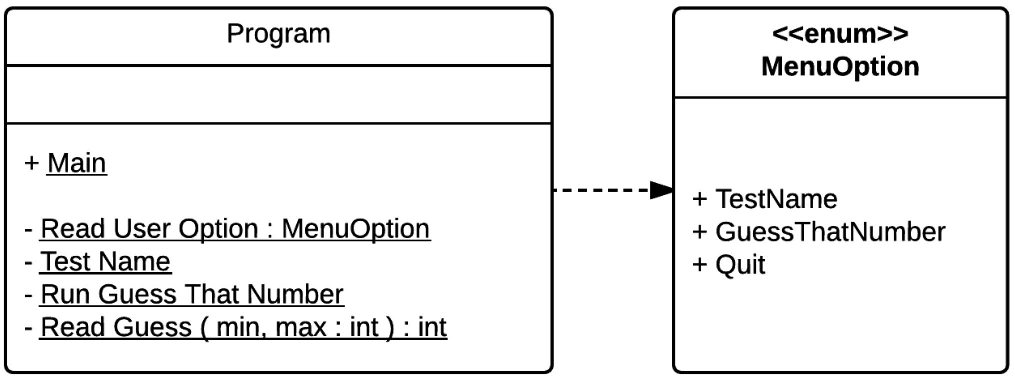

Figure: UML class diagram for the name tester program

1. Create a new `RunGuessThatNumber` static method in the `Program` class.
2. Call this method in `Main` when the user selects that option.
3. Before we write the details, lets create the `ReadGuess` method. This will be a static method that returns the user's guess. We can put some smarts in this so that it checks that the value is between a min and max, which will change as the user makes different guesses.

   The start of this method looks like this:

   ```
   private static int ReadGuess(int min, int max)
   {
       // ...
   }
   ```

4. Within this method, declare a local variable to store the integer value the user enters.

5. Use `Console.WriteLine` to output the message "Enter your guess between" the min and max values.

6. Use a **do while** loop to read a value between `min` and `max`, looping *while* the value read is less than `min` **or** larger than `max`.

   Read the value from the user using `Console.ReadLine` and `Convert.ToInt32` as before, this time storing it in the variable you created in this method.

7. Finally, return the entered value after it has been validated. (so outside of the do while loop)

   At this point you now have a method that can read in a guess from the user. It will make sure that the value is within the range. To make use of this we need to edit the `RunGuessThatNumber`.

8. Now return to `RunGuessThatNumber` method, and we can make use of this to start the same.

9. You will need four variables to help track this program:

   - A variable for the **target**
   - A variable for the user's **guess**
   - A variable to store the **lowest guess** and another for the **highest guess**

10. Start by initialising, **lowest guess** to 1 and **highest guess** to 100.

11. Assign **target** a random value. We can do this using C#'s `Random` class.

    ```
    target = new Random().Next(100) + 1;
    ```

    This creates a `Random` object and asks it to generate a random number between 0 and 99 (100 random values). Adding one to this gives a range from 1 to 100.

12. Output a message "Guess a number between 1 and 100".

13. Add the start of a **while** loop that runs *while* `guess` does not equal `target`. Inside the loop:

    1. Use `ReadGuess` to read in the user's guess, and then store this in the `guess` variable. You will need to pass in *lowest guess* as `min` and *highest guess* as `max`.

    2. *If* the guess is less than the target:

        1. Output a message to indicate this

        2. Change **lowest guess** to store `guess`.

    3. *If* the guess is larger than the target:

        1. Output a message to indicate this

        2. Change the **highest guess** to store `guess`.

    4. Otherwise,

        1. Output a message to indicate that they guessed the number!

14. To test this... lets add a cheat. Output the value of the target, just so we know what it is while we make sure this works.

15. Now switch to the Terminal and build and run your program.

Test that things work as expected. Try putting in values outside the asked range, guessing higher and lower than the target. Then make sure it works when you guess the number.

Handling Errors

Lastly we need to add some error handling to

1. Locate your `ReadUserOption` method.

2. Add a **try catch** block around the convert call. This way you can handle any errors where the user does not enter a number. In the catch block output a message, and set the `option` value to -1 (as this will ensure that the loop runs again).

```
try
{
    // Call to Convert in here...
}
catch
{
    // output message
    option = -1;
}
```

3. Add similar code to the **ReadGuess** method.

4. Now switch to the Terminal and build and run your program.

Now test that the program does not crash when enter something other than an integer at either of these prompts.

When it is all working comment out the code that prints out the target, and you are done.

Grab a screenshot, backup your work, and submit to Doubtfire.