Unit 3: Classes, Objects and Methods

ABOUT YOUR LOGBOOK
For the class problems 1, 3 and 5 only put the C# source code and sample outputs into your logbook. For the problems 2 and 4 , enter in your logbook:

- **Input-Output Diagram**
- **UML Class Diagram (see next page for example)**
- **Algorithms for all Methods (see next page for example)**
- **Source Code and screenshots**
- **Test Plan (with results)**

Classwork (4 Tasks)

In this section of our work we shall start to look at the world of Object Oriented Programming (OOP for short)

Introduction to OOP

- The world around us is made up of objects .. e.g. students, classes, cars, restaurants etc.
- A **class** is like a template or design for an **object**.
- For example, we may have a design for a car class, but my own red/white Citroen rusting away in the car park is an actual object from this class .. it is a specific car with its own colour, registration number, etc. …. we sometimes say this is an **instance** of the car class.
- My car is different from your car and all the others in the car park but they are similar in many ways (number of wheels, engine, windscreen, etc.). This is because they are all objects belonging to the same class .. the car class.

C# Classes

- C# is an Object Oriented programming language .. which means that it allows us to program using classes and objects.
- There must always be at least one **class** in a C# project and there must also always be a **Main()** method (or function) because program execution always starts here.
- So far in this course all your projects have used one class .. which was called **Program**
- When you ran a project, C# looked for and then executed its **Main()** method.
- Note: you can't execute a class .. you must first create an object from it (but see Note below)
  Just as you can't drive a car's design .. you can only drive a real car built from the design!
- In real OOP programming:
  - a class can have any reasonable **name** (usually starting with a Capital letter)
  - a class can have **data** (e.g. variables) that define the class properties or attributes
  - a class can also have **methods** (or functions) that define the class behaviour
  - once a class has been defined, you can create any number of **objects** from it.

Note:

```
class Program
{
  static void Main()
  {
  }
}
```

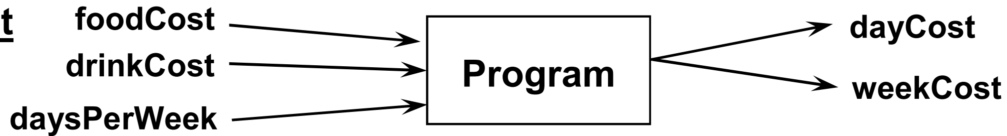
The word **'static'** here means that the Main() method belongs to the class -- NOT an object. So we can use Main() without first creating an object from the class

4.1 Meal Costs

Look at **Task4_1.csproj** which contains a class called **MealCosts**.

- First compile and **run** the project to see what it does.
- See that the program asks you to enter the cost of food and drink, then how many days per week you attend college. It then calculates your college daily and weekly costs (assuming you have 1 meal and 3 drinks per day)

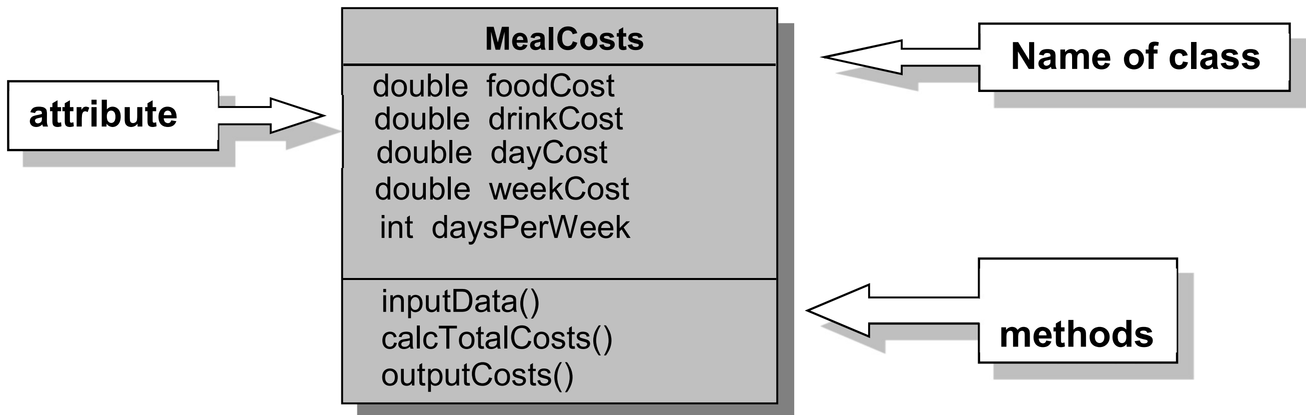
Input-Output Model



- If you examine the code (next page) you will see that there is one **class** (**MealCosts)**.
- Also note that inside the MealCosts class there are 5 variables (sometimes called class attributes or fields) and then there are 4 **methods**
- **Methods** (or functions) are used to perform tasks for a class and in this case these 4 methods are named **inputData()**, **calcTotalCosts()** and **outputCosts()** (plus of course **Main()**)

Class diagram

In OOP programming we often draw a UML Class diagram to show the basic class structure:



Detailed Algorithm (this gives the detail for each method)

1. **method: Main**
   a. Create a new object called **myMeals** from the MealCosts class:
      MealCosts myMeals = new MealCosts();
   b. Call myMeals' inputData method : **myMeals.inputData()**;
   c. Call myMeals' calcTotalCosts method : **myMeals.calcTotalCosts()**;
   d. Call myMeals' outputCosts method : **myMeals.outputCosts()**;

2. **method: inputData**
   a. Input the cost of a meal (**foodCost**)
   b. Input the cost of one drink (**drinkCost**)
   c. Input the number of days attended per week (**daysPerWeek**)

3. **method: calcTotalCosts**
   a. Calculate the **dayCost** as **foodCost** + (**3** * **drinkCost**)
   b. Calculate the **weekCost** as **daysPerWeek** * **dayCost**

4. **method: outputCosts**
   a. Output **dayCost**
   b. Output **weekCost**

```
class MealCosts
{

    double dayCost, weekCost;          // define all class variables (attributes)
    double foodCost, drinkCost;
    int daysPerWeek;                   // number of days attending college

    static void Main()                 // program starts executing here
    {
        MealCosts myMeals = new MealCosts();   // create a new myMeals object
        myMeals.inputData();                   // call object's inputData method
        myMeals.calcTotalCosts();              // call object's calcTotalCosts method
        myMeals.outputCosts();                 // call object's outputCosts method
    }

    void inputData()                   // method to input data from keyboard
    {
        string input;                  // local input variable
        Console.Write("Enter the price of a meal: £");
        input = Console.ReadLine();
        foodCost = Convert.ToDouble(input);
        Console.Write("Enter the price of a drink: £");
        input = Console.ReadLine();
        drinkCost = Convert.ToDouble(input);
        Console.Write("Enter the number of days per week at college: ");
        input = Console.ReadLine();
        daysPerWeek = Convert.ToInt32(input);
    }

    void calcTotalCosts()
    {
        dayCost = foodCost + ( 3 * drinkCost );
        weekCost = dayCost * daysPerWeek;
    }

    void outputCosts()
    {
        Console.WriteLine("\nYour Final Costing Results");
        Console.WriteLine("=========================");
        Console.WriteLine("Total cost for one day = £" + dayCost.ToString("0.00"));
        Console.WriteLine("Total cost for one week = £" +
                                    weekCost.ToString("0.00"));
    }

} // end of MealCosts class
```

Test plan

Test	INPUTS			Expected Results		Actual Results	
No	foodCost	drinkCost	daysPerWeek	dayCost	weekCost	dayCost	weekCost
1	2.20	0.60	5	4.00	20.00		
2	1.50	0.50	5	3.00	15.00		
3	4.50	1.00	2	7.50	15.00		

Task 4.1

- Modify the program so it also enters the user's **name** and then outputs this name along with the results display.
- Add a new **method** to the MealCosts class and call it **introduction()** .. this should display suitable headings and user instructions before the program does its input. Get it to work.
- Not everyone is the same. Some people take more than one meal (or none at all) and not everyone has 3 drinks per day. Modify the program so that it includes another **method** called **getAmounts()** which asks the user how many meals and how many drinks on average they have each day and inputs these values.
- The rest of the program should then use this new data correctly in the calculations.

4.2 Game Score

You are now to write a new program for the following problem to calculate the final score in a computer game. The program has a similar structure to the previous program of this unit, but it should use a class called **GameScore**.
You can use the design and program of problem 4.1 as a guide.

The program has at least **four methods** … as well as **Main** of course.

- The first method inputs the player's name, the number of aliens destroyed, the value of treasure accumulated and the number of hours played.
- The second method calculates the raw score using the formula below:
  Raw Score = (aliens destroyed x 20) + (treasure value x 50)
  It also calculates the final score by applying a time penalty according to how long the game was played:
  - If the game lasted more than 10 hours .. score 50% of the raw score
  - Between 7 and 10 hours .. 70% of the raw score
  - Between 5 and 7 hours .. 80% of the raw score
  - Between 3 and 5 hours .. 100% of the raw score
  - Under 3 hours .. add an extra 50% to the raw score
- The third method outputs all details, including the player name, raw score, and final score.
- Add a fourth method to display a suitable congratulatory (or otherwise) end message that depends on the value of the final score achieved.

Note: For this exercise you **must** use separate methods within a class as described above. You should always use meaningful names for the class and methods.

4.3 Very Dicey

Look at project **Task4_3.csproj** and execute it several times. See that a random number between 1 and 6 is generated by the **Dice** class.

Examine the Dice Class code below:

```
class Dice
{

    private Random rand;            // define rand as a Random class object

    public static void Main()       // program starts executing here
    {
        Dice myDice = new Dice();         // create a new object called myDice
        myDice.rand = new Random();   // create a new Random object
        Console.Clear();                  // clear the console screen
        myDice.throwTheDice();            // call the  throwTheDice method
    }

    public void  throwTheDice()     // call the oneThrow() method
    {
        Console.WriteLine("I have thrown " + oneThrow());
    }

    public int oneThrow()
    {
        return  rand.Next(6) + 1;    // pick a number from 1 to 6 and return this
    }

} // end of Dice class
```

Public and Private

- This program example uses the words **public** and **private** which we have not used before
- It is common practice to use these words to limit access to parts of the program
    - Class variables (attributes) are generally made **private** so that access to them can be carefully controlled (limited to the class they are in)
    - If a method is **public** it is made available to the world outside this class.
    - There is also a **protected** mode (this limits access to the class and any subclasses derived from it). Note that you can create **child** classes that **inherit** the attributes and methods from another class .. this is known as **inheritance**)

Return

- The **oneThrow()** method is being used to **return** a random integer
- This is why it is defined with **public int** instead of **void** (void means nothing is returned)

Task 4.3

1. Add a new method called **throw20Dice** that uses **oneThrow()** in a loop so as to throw the Dice 20 times. Get it to work correctly
2. Change the display in **throw20Dice** so that it has this format:

   Throw No 1 is ….
   Throw No 2 is … etc.
3. Set up a new method called **manyThrows** … this should start by asking how many times you want to throw the dice and then produce a display like the one shown in 2 above

4.4 Dicey Behaviour

For this task you can modify task 4.3 : **Very Dicey** .

- You are to add a new method called **countEm**.
- This method should behave like the **manyThrows** method .. asking how many times you want to throw the dice.
- But It should also count how many times **one**, **two**, **three**, **four**, **five** and **six** appears.
- It should finish by displaying the results like this:

```
Dice Count
==========
Total Number of throws = <          >
==============================
Number of ones      = <        >
Number of twos      = <        >
Number of threes    = <        >
Number of fours     = <        >
Number of fives     = <        >
Number of sixes     = <        >
```

- Test the code by throwing the dice **1000** times.
- In your logbook for this exercise you can put the source code for this method (commented fully) and sample output results

4.5 Craps!

Now you are to try to program a simple version of the American dice game Craps which uses 2 dice.   Follow these instructions:

1. Create a new project with a class called **Craps**.
2. Create a **oneThow()** method -- like the one you had before
3. Create a **throw2Dice()** method that uses oneThrow() twice, printing and returning the result

    e.g.   You threw a 6 and a 4 -- making 10   (get this to work before progressing)
4. Create a **play()** method that uses throw2Dice() and checks the result :

    2, **3** or **12** is Craps   -- You lose! End of game!

    7 or **11**   -- You win! End of game!

    4, **5**, **6**, **8**, **9** or **10**   -- this is your Point (the game continues : see below)
5. Create a **throwPoint()** method which does the following:

    Calls throw2Dice() repeatedly until either your same Point is thrown again -- You Win!

    or **7** is thrown -- You Lose!

    This method is only called if a Point has been obtained in **play()**

Independent Study (2 Tasks)

The following exercises are to be done individually and independently, in your own time.

ABOUT YOUR LOGBOOK
For independent study problem 6 and 7 enter in your logbook:

- **UML Class Diagrams**
- **Source Code and sample outputs**

4.6 Nuclear Control

Look at project **Task4_6.csproj** .. and examine the code on the following pages.

You will find that there are 2 **classes in this project .. NuclearStation** and **Test**

- When you execute this project, the Main() method creates a new Test object called **myTest** and calls its testStation() method:

  Test myTest = new Test();
  myTest.testStation();

- When the new Test object is created, the constructor (a method with the same name as its Class) creates a new NuclearStation object called **myStation**:

  myStation = new NuclearStation();

- The myTest.**testStation()** method calls myStation's **display()** method and you will see the following screen:

  Nuclear WinterLand Station
  ==========================
  Main Menu
  ==========
  1: Lower Fuel Rods
  2: Raise Fuel Rods
  3: Activate Shields
  4: Deactivate Shields
  5: Quit

- It then calls myStation's **getChoice()** method which asks you to enter a choice of 1-5 and returns your entry back.
- If your choice is "1" the **lowerRods()** method is called. This asks you to enter the correct code and you're in trouble if you get it wrong!

Task 4.6

- The program is only partially completed. Your main task is to complete it!
- First examine the code and be clear how it all works.
- Add some new methods to the **NuclearStation** class to deal with the other menu options.
  - Choices 1 and 4 require authorisation before proceeding
  - Choices 2 and 3 are less dangerous .. users are informed about what they are doing and given the chance to change their minds
- The **testStation()** method should be modified to include these new options.
- The **testStation()** method should also use a loop to allow users to continually select choices from the display until the "5" option is chosen.
- Note that allowing any other number choices could be a disaster so you should **validate** the inputs and give suitable error messages if e.g. 7 is chosen.

```
class Test
{
    private string choice;
    private NuclearStation myStation ;

    public static void Main()                  // program starts executing here
    {
            Test  myTest = new Test();         // create new Test object called myTest
            myTest.testStation();         // call its testStation method
    }

    public Test()                              // the Test class constructor
    {
          myStation = new NuclearStation();   // create new object from other class
    }

    public void testStation()
    {
            myStation.display();                // call the station display method
            choice = myStation.getChoice();          // get choice returned
            if (choice == "1")
            {
               myStation.lowerRods();          // call lowerRods method
            }
    }
} // end of Test class
```

```
class NuclearStation
{
    private const string SYSTEMCODE = "NUKEME";    // set authorisation code constant

    public void display()
    {
             Console.WriteLine("Nuclear WinterLand Station");
             Console.WriteLine("==========================");
          Console.WriteLine("     Main Menu");
          Console.WriteLine("     ==========");
          Console.WriteLine("     1:  Lower Fuel Rods");
          Console.WriteLine("     2:  Raise Fuel Rods");
          Console.WriteLine("     3:  Activate Shields");
          Console.WriteLine("     4:  Deactivate Shields");
             Console.WriteLine(" 5:  Quit\n");
    }

    public string getChoice()
    {
            string choice;                     // local string variable
            Console.WriteLine("\nWhat do you want to do?");
            Console.Write("Enter your choice : ");
            choice = Console.ReadLine();
            return  choice;                    // return user choice as a string
    }
```

// NuclearStation class continues on next page

```
public  void  lowerRods()
{
        string code;
        Console.WriteLine("\nWARNING:DANGER:   Lowering Fuel Rods!");
        Console.WriteLine("You require an authorisation code to do this");
        Console.Write("Enter your code now:");
        code = Console.ReadLine();                    //enter the code
        if (code == SYSTEMCODE)                       // check the code
        {
           Console.WriteLine("\n** CODE CORRECT : Fuel rods being lowered");
        }
        else
        {
           Console.WriteLine("\n** INCORRECT : please stay where you are!");
           Console.WriteLine("You will now be escorted from the building!");
        }
}
```

```
}  // end of NuclearStation class
```

4.7 More Craps!

Start a new console application called Task4_7 with a class called **CrapsGame** which is to be used with a Craps class. You may want to read up on constructors before tackling this. Now follow these steps and look at the previous task 4.6 for some clues.

1. Looking at the **CrapsGame** class, select **Project** from the menubar, followed by **Add Class** and name this new class **Craps**. You should now have 2 classes in the same project.
2. Now put the code from your previous **Craps** class of Task 4.5 into this new Craps class (you can copy and paste if you want)
3. Create a Constructor method in the Craps class (it should have the same name as the class)
4. Cut and paste the code that creates a new Random object into this constructor from Main()
5. You probably will have 2 Main() methods in 2 classes .. this will confuse the program because it doesn't know where to start - so remove the Main() method from Craps and use it in CrapsGame instead (paste it over the top)
6. Now check that your project still works as it did before !
7. In the **CrapsGame** class .. create a new method called **manyPlay()** that will allow you to play the game as many times as you want.
8. Create an **updateScore()** method in CrapsGame that keeps a count of games won and games lost (you will need to get the **play()** method in Craps to return something).
9. Add a **rules()** method and a **finalScore()** method to CrapsGame which display the appropriate information.