104222099 – Phan Vu

TASK 2

1. **Describe the principle of polymorphism and how it was used in Task 1.**

- Polymorphism is a key concept in Object-Oriented Programming. It lets different types of objects be seen as the same, making the code more flexible. It simplifies coding for different objects without modifying it for each type. Extensibility allows you to add new features to the code by adding new classes without changing the code that depends on the superclass or interface.

- In Task 1, polymorphism is used in:

Abstract Base Class (SummaryStrategy): There is an abstract base class called **SummaryStrategy**. It has a method called **PrintSummary**. This class is a contract that all summary strategies must follow. It requires derived classes to have their own **PrintSummary** method.

Concrete subclasses (AverageSummary and MinMaxSummary): Two subclasses, **AverageSummary** and **MinMaxSummary**, inherit from the base class **SummaryStrategy.** The subclasses have their own **PrintSummary** method for their specific summary strategies (average and min-max). This shows polymorphism because both subclasses follow the rules of the base class but have their own particular actions.

Polymorphic Usage in DataAnalyser: The **DataAnalyser** class uses polymorphism with a private variable called **_strategy**. The **_strategy** variable can store instances of any class derived from **SummaryStrategy**. The **DataAnalyser** class can work with different summary strategies without knowing their specific implementations. When **Summarise** is called, it runs **PrintSummary** on the **_strategy** object. Depending on the strategy

(**AverageSummary** or **MinMaxSummary**), **PrintSummary** is executed to provide different summary outputs.

In summary, I could use the SummaryStrategy reference to get to both classes. Polymorphism was also shown by the fact that the **SummaryStrategy** class could look or act differently based on whether it was an **AverageSummary** or a **MinMaxSummary**. This also made it easy to show the **DataAnalyzer** class's values because I only had to call the **_strategy** method. No matter what subclass of **SummaryStrategy** was given to the **_strategy** field, the **Printsummary (_numbers)** method would always be called.

2. **Using an example, explain the principle of abstraction. In your answer, refer to how classes in OO programs are designed.**

- Abstraction in object-oriented programming simplifies complicated systems by concentrating on property roles and avoiding implementation specifics. It lets you make models or representations of things or processes that exist in the real world, with only the most important data and behaviors included for each situation. By hiding the details, the system is easier to handle and can be broken down into smaller parts. It also has a more straightforward way to deal with things. Here is an example:

```
public abstract class Vehicle
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("The car is moving.");
    }
}

public class Bike : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("The bike is moving.");
    }
}
```

- The Vehicle class in this case is an abstract parent class that defines the Move method. The Concrete class Vehicle is where the Car and Bike classes come from, and they both implement the Move method. Now I'm gonna make a list of Vehicle objects and move each one to show how abstraction works in more detail.

```
public static void Main(string[] args)
{
    List<Vehicle> vehicles = new List<Vehicle>();
    vehicles.Add(new Car());
    vehicles.Add(new Bike());

    foreach (Vehicle vehicle in vehicles)
    {
        vehicle.Move();
    }
}
```

- The Main method generates a Vehicle list of Car and Bike objects. Abstraction lets us invoke the Move method on any Vehicle object regardless of its kind. We can operate with a collection of objects without caring about their implementation.

3. What was the issue with the original design in Task 1? Consider what would happen if we had 50 different summary approaches to choose from instead of just 2.

- The code in the original design lacks OOP structure and has problems with scalability and redundancy, making the program insufficient and potentially becoming more complex as more approaches are added.
- If there were 50 alternative summary ways instead of just two, more memory would be needed at initially than if the OOP structure was employed. Also, the writer has to use the same if/else statements over and over, which can make the code hard to read and make it harder to fix bugs, which would reduce extensibility and maintainability. However, using OOP structure (ideas like inheritance, polymorphism, encapsulation, and abstraction) as we did in Task1 means that common methods can be used again and again, making it easier to add to or change code without worrying about how easy it is to read.