Practical Task 6.1

(Pass Task)

Submission deadline: 10:00am Monday, May 4
Discussion deadline: 10:00am Saturday, May 16

General Instructions

This practical task introduces you to ***polymorphism***, one of the four basic concepts of Object-Oriented Programming (OOP) along with ***abstraction***, ***encapsulation***, and ***inheritance***. Polymorphism is the ability of an object to take on "***many forms***". It occurs when we have several classes that are related to each other by ***inheritance***; that is, when they implement the "parent-child" type of class relationship. While inheritance lets one class to inherit attributes and methods from another class, polymorphism helps us to select specific class (a child class or the parent class itself) at the runtime, and call the methods associated with the selected class. Thus, it allows us to perform a single action in different ways.

For example, think of a parent class called `Animal` that has a method called `AnimalSound()`. Child classes of the `Animal` class could be `Pig`, `Cat`, `Dog`, and `Bird`, which all may also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.). Polymorphism allows us to treat objects of `Pig`, `Cat`, `Dog`, and `Bird` as objects of the parent `Animal` class that makes the parent and the child classes compatible. Therefore, we can refer to their objects as objects of the `Animal` class. This allows us to deal with a reference variable of type `Animal`, but play sound of the class object that the reference points to currently at the runtime.

Polymorphism implies that any child class object can take any form of a class in its parent hierarchy and of course itself as well. What this technically means in code is that the child class object can be assigned to any class reference in its parent hierarchy. For example, consider a `Student` class as a child of a `Person` class. Then the following assignments are both valid and represent the essence of inheritance: A reference to a child class is type-compatible with a reference to its parent class.

```
Student student = new Student();
Person person = new Student();
```

Polymorphism is the art of taking advantage of this simple but powerful and versatile feature. In our example, the second line guarantees that the methods associated with the `Student` class will be executed when called on the `person` reference variable.

The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Actually, here we are talking about ***dynamic polymorphism***. But there is also another type, which is known as ***static polymorphism***.

Dynamic polymorphism is often mentioned as run-time polymorphism, dynamic binding, run-time binding, late binding, and method overriding. Here, having many forms is associated with using parent-child related classes. This is what is generally referred as "polymorphism". To state it differently, consider the following two facts:

- Let's assume there are methods with a same method signature in classes in a class hierarchy (parent-child relationships). These methods have different implementation (this is known as ***method overriding***).
- Then when an object is assigned to a ***parent class reference*** and when a method of the object is called, **the method in the object's class is executed, not the method of the reference class**.

Because objects are created at runtime, the method which should be executed also can be only decided at runtime.

Static polymorphism is known as compile-time polymorphism, static binding, compile-time binding, early binding and ***method overloading***. This is actually what you have already seen earlier in Task 5.1. Method overloading is having more than one method with a same method name, but with different arguments (though return type may or may not be the same). Here, when calling a method, the compiler chooses which method to call depending on the passed parameters. This happens at ***compile-time***. Though one may say that method overloading is not true polymorphism, it is often categorized as a type of polymorphism since in method overloading there is a method acting in "many forms".

1. In this exercise, you will further practice class inheritance and work on a program code to hold details of several birds. Specifically, you will need to create three classes: one `Bird` base class and two sub-classes, `Penguin` and `Duck`, which inherit from the `Bird`. You will then use polymorphism to print out a list of `Bird` objects consisting of `Bird`, `Penguin` and `Duck` objects.

2. **STEP 1:** Create the `Bird` base class, which has a `Name` property, a ***virtual*** `fly()` method, and the `ToString()` method. `ToString()` is actually inherited from the `Object` superclass, and therefore must be ***overridden***. Add the code given below and save it to a new `Bird.cs` file.

```
public String name { get; set; }

public Bird()
{

}

public virtual void fly()
{
    Console.WriteLine("Flap, Flap, Flap");
}
```

Since all objects inherit from the C# base `Object` class, every object will have a `ToString()` method. We can ***override*** this method so that when the `Bird` object's `ToString()` method is called, we can output a specific message or perform a certain action. We will override the method to return the bird's `Name` when it is called.

```
public override String ToString()
{
    return "A bird called " + name;
}
```

STEP 2: You will now need to create two `Bird` objects to see if the `Bird` class has been implemented correctly. Navigate to the `Program` class and add the following code.

```
Bird bird1 = new Bird();
Bird bird2 = new Bird();

bird1.name = "Feathers";
bird2.name = "Polly";

Console.WriteLine(bird1.ToString());
bird1.fly();

Console.WriteLine(bird2.ToString());
bird2.fly();

Console.ReadLine();
```

Now, when the base class is complete, we will need to create a `Penguin` and a `Duck` class that both inherit from the `Bird`. Create the `Penguin` sub-class first. The class will override the `fly()` method inherited from `Bird` and will also override the `ToString()` method inherited from the `Object`. To add the new class, type the following code.

```
class Penguin : Bird
{

    public override void fly()
    {
        Console.WriteLine("Penguins cannot fly");
    }

    public override string ToString()
    {
        return "A penguin named " + base.name;
    }
}
```

Now that the Penguin class is complete, you will create a Duck class that also inherits from the Bird. The Duck class will also have two attributes: size and kind. The class will override the ToString() method inherited from the Object. To add the new class, write the code as given below.

```
class Duck : Bird
{
    public double size { get; set; }
    public String kind { get; set; }

    public override string ToString()
    {
        return "A duck named " + base.name + " is a " + size + " inch " + kind;
    }
}
```

We will now create two Penguin objects and two Duck objects to check if the both classes override methods and have been implemented correctly. Navigate to the Program class and add the following code.

```
Penguin penguin1 = new Penguin();
Penguin penguin2 = new Penguin();

penguin1.name = "Happy Feet";
penguin2.name = "Gloria";

Console.WriteLine(penguin1.ToString());
penguin1.fly();

Console.WriteLine(penguin2.ToString());
penguin2.fly();
```

```
Duck duck1 = new Duck();
Duck duck2 = new Duck();

duck1.name = "Daffy";
duck1.size = 15;
duck1.kind = "Mallard";

duck2.name = "Donald";
duck2.size = 20;
duck2.kind = "Decoy";

Console.WriteLine(duck1.ToString());
Console.WriteLine(duck2.ToString());
```

Compile the program and run it to check if the produced output is as expected.

Now, you will create a List collection of type Bird and assign to the list objects of the Bird, Penguin and Duck classes. You can do this as Penguin and Duck inherit from Bird, i.e. you can assign a subclass to an object of the base (parent) type. Let's first create a List of type Bird (the base class), and tidy up the code in the Program's main method as shown below.

```
List<Bird> birds = new List<Bird>();
Bird bird1 = new Bird();
bird1.name = "Feathers";
Bird bird2 = new Bird();
bird2.name = "Polly";


Penguin penguin1 = new Penguin();
penguin1.name = "Happy Feet";
Penguin penguin2 = new Penguin();
penguin2.name = "Gloria";
```

```
Duck duck1 = new Duck();
duck1.name = "Daffy";
duck1.size = 15;
duck1.kind = "Mallard";
Duck duck2 = new Duck();
duck2.name = "Donald";
duck2.size = 20;
duck2.kind = "Decoy";

birds.Add(bird1);
birds.Add(bird2);
birds.Add(penguin1);
birds.Add(penguin2);
birds.Add(duck1);
birds.Add(duck2);
```

Note that you can add a new instance of an object straight to the list and use the object initialiser to populate the applicable fields, e.g. `Name`:

```
birds.Add(new Bird{Name = "Birdy"});
```

Use a loop to call the `ToString()` method of each bird or simply call the object using the variable name in the `List`, for example:

```
foreach (Bird bird in birds)
{
    Console.WriteLine(bird);
}
```

Compile the program and run it to check if the produced output is as expected.

3. ***Covariance*** is a way to add an entire list of `Duck` objects to a list of `Bird` objects. Covariance allows an object that is instantiated with a more derived type argument (i.e. the `Duck`) to be assigned to an object with a less derived type argument (i.e. `Bird`) and the assignment compatibility is preserved.

   You will achieve covariance using the `IEnumerable<T>` interface. To do this, first create a `List` of `Duck` objects in the `Main` method.

```
Duck duck1 = new Duck();
duck1.name = "Daffy";
duck1.size = 15;
duck1.kind = "Mallard";
Duck duck2 = new Duck();
duck2.name = "Donald";
duck2.size = 20;
duck2.kind = "Decoy";

List<Duck> ducksToAdd = new List<Duck>()
{
    duck1,
    duck2,
};
```

   Next, create an `IEnumerable` of type `Bird` and assign your list of ducks to it. A reference of the `Duck` list is assigned to the `IEnumerable<Bird>` interface variable.

```
IEnumerable<Bird> upcastDucks = ducksToAdd;
```

   Covariance is really useful when you have a collection of objects and want to add them to a more general list. Let's first create a list of `Bird` objects.

```
List<Bird> birds = new List<Bird>();
birds.Add(new Bird() { name = "Feather" });
```

   Now add the `IEnumerable<Bird>` collection to the `Bird` list using the `AddRange` method which accepts an `IEnumerable<T>` interface variable.

```
birds.AddRange(upcastDucks);
```

   Use a loop to call the `ToString()` method of each bird or simply call the object using the variable name in the `List` to see if all duck objects were added to the list.

```
foreach (Bird bird in birds)
{
    Console.WriteLine(bird);
}
```

   Compile the program and run it to check if the produced output is as expected.

4. Examine the following code and represent the classes and their relationship as a UML class diagram.

```
abstract public class Animal
{
    abstract public void Greeting();
}
public class Cat : Animal
{
    override public void Greeting() {
        Console.WriteLine("Cat: Meow!");
    }
}
public class Dog : Animal
{
    override public void Greeting() {
        Console.WriteLine("Dog: Woof!");
    }

    public void Greeting(Dog another) {
        Console.WriteLine("Dog: Wooooooooooof!");
    }
}
public class BigDog : Dog
{
    override public void Greeting() {
        Console.WriteLine("BigDog: Woow!");
    }

    new public void Greeting(Dog another) {
        Console.WriteLine("Wooooooowwwww!");
    }
}
```

Explain the outputs (or existing errors) for the following test program.

```
public class TestAnimal
{
    public static void Main(String[] args) {
        // Using the subclasses
        Cat cat1 = new Cat();
        cat1.greeting();
        Dog dog1 = new Dog();
        dog1.greeting();
        BigDog bigDog1 = new BigDog();
        bigDog1.greeting();
        // Using Polymorphism
        Animal animal1 = new Cat();
        animal1.greeting();
        Animal animal2 = new Dog();
        animal2.greeting();
        Animal animal3 = new BigDog();
        animal3.greeting();
        Animal animal4 = new Animal();
        // Downcast
        Dog dog2 = (Dog)animal2;
        BigDog bigDog2 = (BigDog)animal3;
        Dog dog3 = (Dog)animal3;
        Cat cat2 = (Cat)animal2;
        dog2.greeting(dog3);
        dog3.greeting(dog2);
        dog2.greeting(bigDog2);
        bigDog2.greeting(dog2);
        bigDog2.greeting(bigDog1);
    }

}
```

Further Notes

- Explore Section 6.2 of the SIT232 Workbook to learn the concept of polymorphism, how it is achieved, and how to apply it in solving problems. Section 6.3 explains how abstract methods and classes provide the ability to express the desirable semantics of inheritance and how to use them. The Workbook is available in CloudDeakin → Resources.
- The following links give some additional notes on polymorphism, abstract methods and classes, and the mechanism of method overriding in C#:
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism
  - https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm
  - https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/
  - https://docs.microsoft.com/en-us/dotnet/api/system.object
  - https://www.c-sharpcorner.com/UploadFile/0c1bb2/ienumerable-interface-in-C-Sharp/
  - If you are looking for a free tool to build UML diagrams, then Lucidchart can be a good option.
  - https://www.lucidchart.com

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your program implements the required functionality. It must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit the expected code files as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their work and solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.
- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.
- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.