Object Oriented Programming In C#

This practical session will focus primarily on refreshing the knowledge of OOP and practicing relevant syntax.

Class declaration

To create a class:

```
public class Person { }
```

Here, the public is what we call an access modifier. It determines whether a class is visible to other classes or not. Here is a class with a field and a method:

```
public class Person

{

public string Name;

public void Introduce()

{

Console.WriteLine("My name is " + Name);

}

}
```

Here void means this method does not return a value. To create an object, we use the new operator:

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

A cleaner way of writing the same code is:

```
var person = new Person();
```

We use the new operator to allocate memory to an object. In C# you don't have to worry about de-allocating the memory.

CLR(**Common Language Runtime)** has a component called Garbage Collector, which automatically removes unused objects from memory.

Once we have an object, we can access its fields and methods with the dot notation:

```
Person person = new Person();

person.Name = "John";
```

person.Introduce();

Access Modifiers

In C# we have 5 access modifiers:

public

A class member declared with the **public is accessible everywhere.**

```
class Student {
   public string name = "Sheeran";

   public void print() {
     Console.WriteLine("Hello from Student class");
    }
 }

 class Program {
   static void Main(string[] args) {

     // creating object of Student class
     Student student1 = new Student();

     // accessing name field and printing it
     Console.WriteLine("Name: " + student1.name);

     // accessing print method from Student
     student1.print();
     Console.ReadLine();
   }
 }
```

private

A class member declared with private is accessible only from inside the class.

```
using System;

namespace MyApplication {

 class Student {
   private string name = "Sheeran";

   private void print() {
     Console.WriteLine("Hello from Student class");
    }
 }

 class Program {
   static void Main(string[] args) {

     // creating object of Student class
```

```
      Student student1 = new Student();

      // accessing name field and printing it
      Console.WriteLine("Name: " + student1.name);

      // accessing print method from Student
      student1.print();

      Console.ReadLine();
    }
  }
```

Protected

A member declared as protected is **accessible only from the class and its derived classes**.

Try this code

```
using System;

namespace MyApplication {

  class Student {
    protected string name = "Sheeran";
  }

  class Program {
    static void Main(string[] args) {

      // creating object of student class
      Student student1 = new Student();

      // accessing name field and printing it
      Console.WriteLine("Name: " + student1.name);
      Console.ReadLine();
    }
  }
}
```

Change above code as follows

```
using System;

namespace MyApplication {

  class Student {
    protected string name = "Sheeran";
  }
// derived class
  class Program : Student {
    static void Main(string[] args) {

      // creating object of derived class
      Program program = new Program();

      // accessing name field and printing it
      Console.WriteLine("Name: " + program.name);
      Console.ReadLine();
    }
  }
}
```

Constructors and Inheritance

- Constructors are not inherited and need to be explicitly defined in the derived class.
- When creating an object of a type that is part of an inheritance hierarchy, base class constructors are always executed first.

```
using System;
  public class Employee
   {
      public Employee()
      {
          Console.WriteLine("Default Constructor Invoked");
      }
      public static void Main(string[] args)
      {
          Employee e1 = new Employee();
          Employee e2 = new Employee();
      }
   }
```

Single Level Inheritance Example: Inheriting Fields

```
using System;
  public class Employee
   {
     public float salary = 40000;
  }
  public class Programmer: Employee
  {
     public float bonus = 10000;
  }
```

```
class TestInheritance{
    public static void Main(string[] args)
     {
        Programmer p1 = new Programmer();

        Console.WriteLine("Salary: " + p1.salary);
        Console.WriteLine("Bonus: " + p1.bonus);

.    }
.  }
```

Single Level Inheritance Example: Inheriting Methods

```
using System;
  public class Animal
   {
     public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal
  {
     public void bark() { Console.WriteLine("Barking..."); }
  }
  class TestInheritance2{
     public static void Main(string[] args)
      {
         Dog d1 = new Dog();
         d1.eat();
         d1.bark();
      }
   }
```

C# Method Overloading

Having two or more methods with same name but different in parameters, is known as method overloading in C#

can overload:

- methods,
- constructors,

can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

```
using System;
public class Cal{
   public static int add(int a,int b){
      return a + b;
   }
   public static int add(int a, int b, int c)
   {
      return a + b + c;
   }
}
public class TestMemberOverloading
{
   public static void Main()
   {
      Console.WriteLine(Cal.add(12, 23));
      Console.WriteLine(Cal.add(12, 23, 25));
   }
. }
```

C# Method Overriding Example

If derived class defines same method as defined in its base class, it is known as method overriding in C#

```
using System;
public class Animal{
   public virtual void eat(){
      Console.WriteLine("Eating...");
   }
}
public class Dog: Animal
{
   public override void eat()
   {
      Console.WriteLine("Eating bread...");
   }
}
```

```
public class TestOverriding
{
   public static void Main()
   {
      Dog d = new Dog();
      d.eat();
   }
}
```

C# Polymorphism(example)

```
using System;
public class Shape{
   public virtual void draw(){
      Console.WriteLine("drawing...");
   }
}
public class Rectangle: Shape
{
   public override void draw()
   {
      Console.WriteLine("drawing rectangle...");
   }

}
public class Circle : Shape
{
   public override void draw()
   {
      Console.WriteLine("drawing circle...");
   }

}
public class TestPolymorphism
{
   public static void Main()
   {
      Shape s;
      s = new Shape();
      s.draw();
      s = new Rectangle();
      s.draw();
      s = new Circle();
.       s.draw();
.
.    }
. }
```

C# Encapsulation

Encapsulation is the concept of wrapping data into a single unit. It collects data members and member functions into a single unit called class. The purpose of encapsulation is to prevent alteration of data from outside. This data can only be accessed by getter functions of the class.

A fully encapsulated class has getter and setter functions that are used to read and write data. This class does not allow data access directly.

Here, we are creating an example in which we have a class that encapsulates properties and provides getter and setter functions.

1. Usage of get and set Method
   a. private variables can only be accessed within the same class
   b. can access them using get and set method

Exercise 1

Automatic Properties (Short Hand)	

```
namespace First
{
    class Animal
    {
        public string Name
        { get; set; }
    }
    class Program
    {
        static void Main(string[]
args)
        {
            Animal A1 = new Animal();
            A1.Name = "Lion";

Console.WriteLine(A1.Name);
        }
    }
}
```

```
namespace First
{
    class Animal
    {
        private string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
    }
    class Program
    {
        static void Main(string[]
args)
        {
            Animal A1 = new Animal();
            A1.Name = "Lion";

Console.WriteLine(A1.Name);
        }
    }
}
```

Usage of this Keyword

```
using System;
  public class Employee
   {
      public int id;
      public String name;
      public float salary;
      public Employee(int id, String name,float salary)
      {
         this.id = id;
         this.name = name;
         this.salary = salary;
      }

      public void display()
      {
         Console.WriteLine(id + " " + name+" "+salary);
      }
  }
  class TestEmployee{
     public static void Main(string[] args)
      {
         Employee e1 = new Employee(101, "Sonoo", 890000);
         Employee e2 = new Employee(102, "Mahesh", 490000);
         e1.display();
         e2.display();
.
.     }
.   }
```

Questions

1. Consider the following C# code for a banking system involving a **`BankAccount`** class and a **`SavingsAccount`** class that inherits from **`BankAccount`**: and understand the code

```
using System;

public class BankAccount
{
    public string AccountNumber { get; set; }
    public double Balance { get; protected set; }

    public BankAccount(string accountNumber, double initialBalance)
    {
        AccountNumber = accountNumber;
        Balance = initialBalance;
    }
    public virtual void Deposit(double amount)
    { Balance += amount; Console.WriteLine($"Deposited {amount:C}. New balance: {Balance:C}");
    }

    public virtual void Withdraw(double amount)
    {
        if (amount <= Balance)
        { Balance -= amount; Console.WriteLine($"Withdrew {amount:C}. New balance: {Balance:C}");
        } else
        { Console.WriteLine("Insufficient funds.");
        }
    }
}

public class SavingsAccount : BankAccount
{
    public double InterestRate { get; set; }

    public SavingsAccount(string accountNumber, double initialBalance, double interestRate) :
base(accountNumber, initialBalance)
    {
        InterestRate = interestRate;
    }

    public void ApplyInterest()
    { double interest = Balance * InterestRate / 100;
        Deposit(interest);
    Console.WriteLine($"Interest applied at rate {InterestRate}%. New balance: {Balance:C}");
    }
}

public class Program
{
    public static void Main()
    {
        SavingsAccount mySavings = new SavingsAccount("12345", 1000, 2.50);
        mySavings.Deposit(500); mySavings.Withdraw(200);
        mySavings.ApplyInterest();
        Console.WriteLine($"Final balance: {mySavings.Balance:C}");
        Console.ReadLine();
    }
}
```

I. Try to Add BankAccount Class and Saving Account Class to the your Application as separate classes (Project ->Add Class)

II. Import BankAccount Class SavingAccount Class to the main program(`Using` {NameSpacename}).