03 Object-Oriented Programming

Test your knowledge

1. What are the six combinations of access modifier keywords and what do they do?

Public: member can be accessed anywhere

Protected: member can be accessed in the current classes and child classes

Internal: member can be accessed in the current assembly

Private: member can only be accessed in current class

Protected Internal: member can be accessed in the same assembly or in a derived class in other assemblies.

Private Protected: member can be accessed inside the containing class or in a class that derives from a containing class, but only in the same assembly.

2. What is the difference between the static, const, and readonly keywords when applied to a type member?

Constant and ReadOnly are used to make a field constant which value cannot be modified. The static keyword is used to make members static that can be shared by all the class objects.

3. What does a constructor do?

Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

4. Why is the partial keyword useful?

The partial keyword indicates that other parts of the class, struct, or interface can be defined in the namespace.

5. What is a tuple?

A tuple is a data structure that contains a sequence of elements of different data types. It can be used where you want to have a data structure to hold an object with properties.

6. What does the C# record keyword do?

Record keyword can be used to define a reference type that provides built-in functionality for encapsulating data.

7. What does overloading and overriding mean?

Overloading: multiple methods in the same class, they have same method signature, including access modifiers and method name. but different input and output parameters

Overriding: happens between base class and derived class. They have the same method signature, including access modifiers, method name, and input parameters

8. What is the difference between a field and a property?

A field is a variable of any type that is declared directly in the class, while property is a member that provides a flexible mechanism to read, write or compute the value of a private field.

9. How do you make a method parameter optional?

By Params keyword: It allows to pass any variable number of parameters to a method.

10. What is an interface and how is it different from abstract class?

An interface only allows to define functionality, not implement it. Abstract class can inherit another class using extends keyword and implement an interface. Interface can inherit only an interface.

11. What accessibility level are members of an interface?

Public declared accessibility. No access modifiers are allowed on interface member declarations.

12. True/False. Polymorphism allows derived classes to provide different implementations of the same method.

True

13. True/False. The override keyword is used to indicate that a method in a derived class is providing its own implementation of a method.

True

14. True/False. The new keyword is used to indicate that a method in a derived class is providing its own implementation of a method.

False

15. True/False. Abstract methods can be used in a normal (non-abstract) class.

False

16. True/False. Normal (non-abstract) methods can be used in an abstract class.

True

17. True/False. Derived classes can override methods that were virtual in the base class.

True

18. True/False. Derived classes can override methods that were abstract in the base class.

True

19. True/False. In a derived class, you can override a method that was neither virtual non abstract in the base class.

False

20. True/False. A class that implements an interface does not have to provide an implementation for all of the members of the interface.

False

21. True/False. A class that implements an interface is allowed to have other members that aren’t defined in the interface.

True

22. True/False. A class can have more than one base class.

False

23. True/False. A class can implement more than one interface.

True

Working with methods

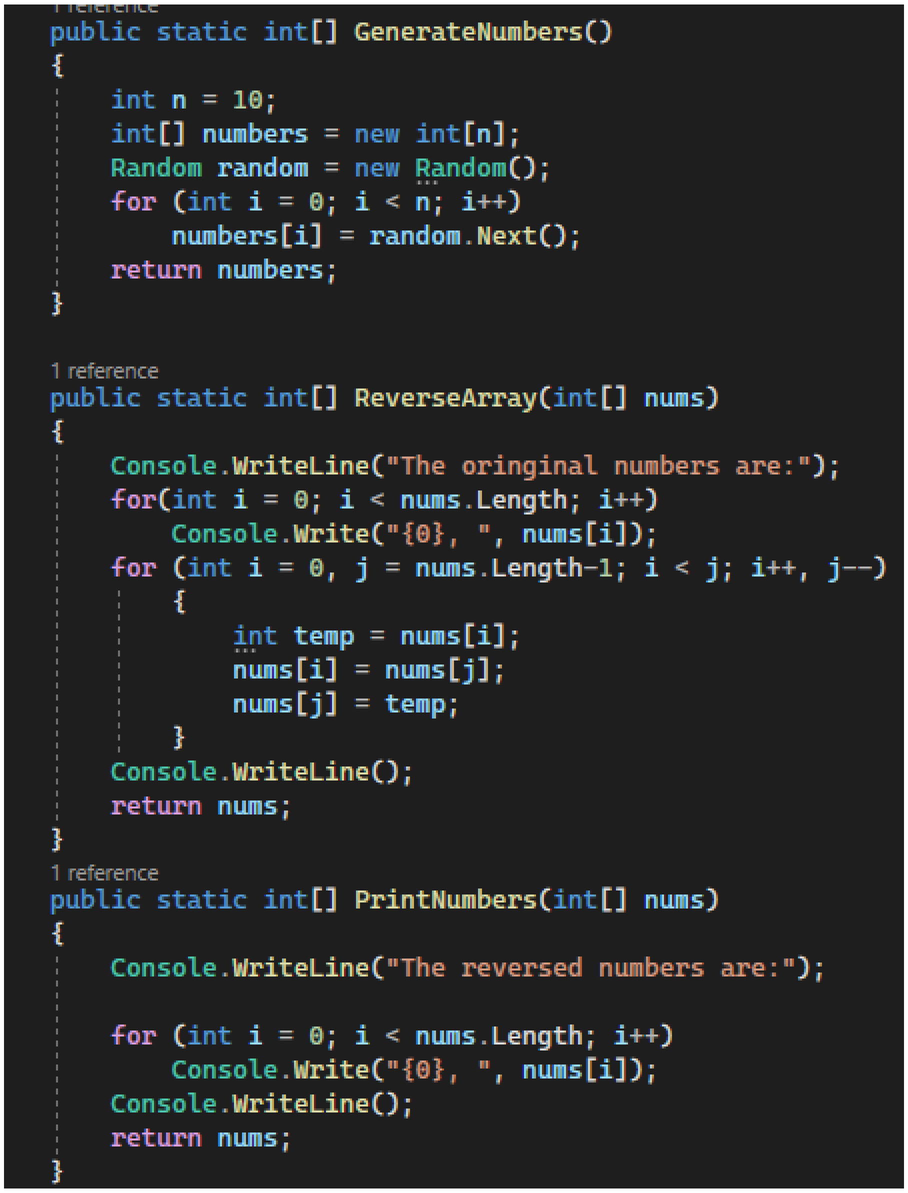
1. Let's make a program that uses methods to accomplish a task. Let's take an array and reverse the contents of it. For example, if you have 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, it would become 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.
To accomplish this, you'll create three methods: one to create the array, one to reverse the array, and one to print the array at the end.
Your Mainmethod will look something like this:

```
static void Main(string[] args) {

    int[] numbers = GenerateNumbers();

    Reverse(numbers);

    PrintNumbers(numbers);

}
```

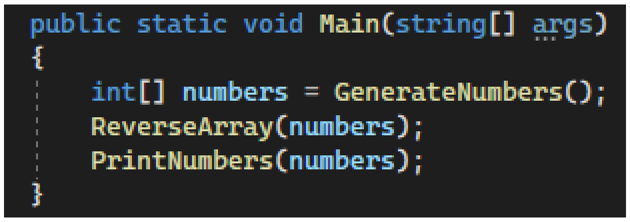
The GenerateNumbersmethod should return an array of 10 numbers. (For bonus points, change the method to allow the desired length to be passed in, instead of just always being 10.)
The PrintNumbersmethod should use a foror foreachloop to print out each item in the array. The Reversemethod will be the hardest. Give it a try and see what you can make happen. If you get stuck, here's a couple of hints:
Hint #1: To swap two values, you will need to place the value of one variable in a temporary location to make the swap:

```
// Swapping a and b.

int a = 3;

int b = 5;

int temp = a;

a = b;

b = temp;
```

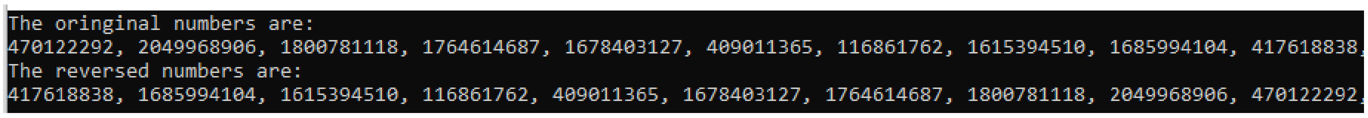
Hint #2: Getting the right indices to swap can be a challenge. Use a forloop, starting at 0 and going up to the length of the array / 2. The number you use in the forloop will be the index of the first number to swap, and the other one will be the length of the array minus the index minus 1. This is to account for the fact that the array is 0-based. So basically, you'll be swapping array[index]with array[arrayLength – index – 1].



```
public static int[] GenerateNumbers()
{
    int n = 10;
    int[] numbers = new int[n];
    Random random = new Random();
    for (int i = 0; i < n; i++)
        numbers[i] = random.Next();
    return numbers;
}

1 reference
public static int[] ReverseArray(int[] nums)
{
    Console.WriteLine("The oringinal numbers are:");
    for(int i = 0; i < nums.Length; i++)
        Console.Write("{0}, ", nums[i]);
    for (int i = 0, j = nums.Length-1; i < j; i++, j--)
        {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    Console.WriteLine();
    return nums;
}
1 reference
public static int[] PrintNumbers(int[] nums)
{
    Console.WriteLine("The reversed numbers are:");

    for (int i = 0; i < nums.Length; i++)
        Console.Write("{0}, ", nums[i]);
    Console.WriteLine();
    return nums;
}
```

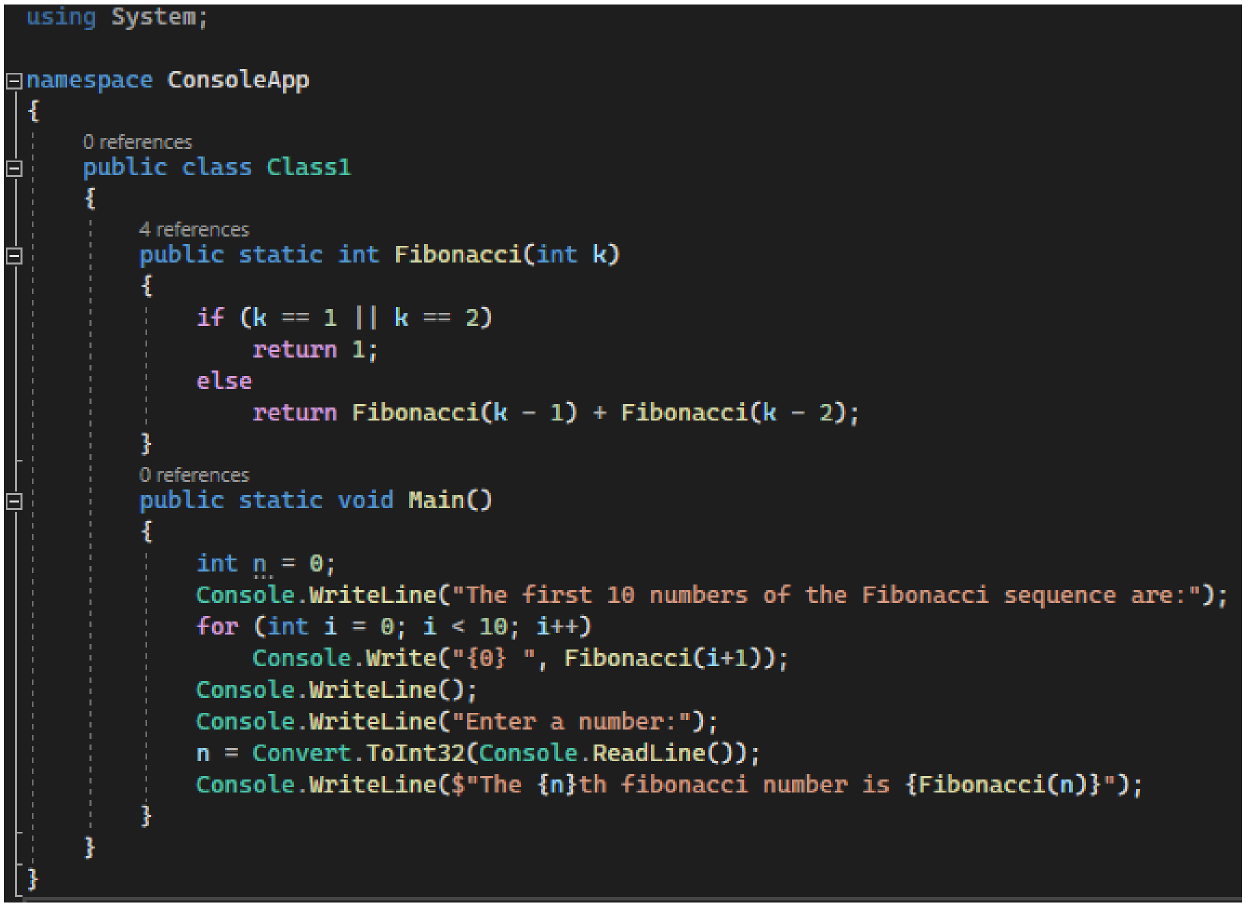


```
public static void Main(string[] args)
{
    int[] numbers = GenerateNumbers();
    ReverseArray(numbers);
    PrintNumbers(numbers);
}
```

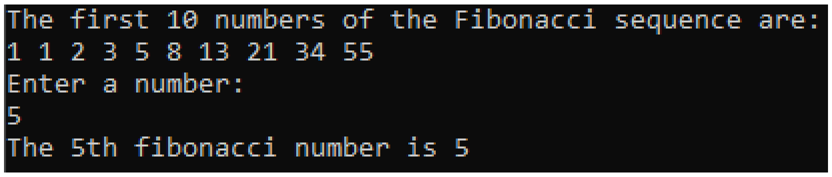


```
The oringinal numbers are:
470122292, 2049968906, 1800781118, 1764614687, 1678403127, 409011365, 116861762, 1615394510, 1685994104, 417618838
The reversed numbers are:
417618838, 1685994104, 1615394510, 116861762, 409011365, 1678403127, 1764614687, 1800781118, 2049968906, 470122292
```

2. The Fibonacci sequence is a sequence of numbers where the first two numbers are 1 and 1, and every other number in the sequence after it is the sum of the two numbers before it. So the third number is 1 + 1, which is 2. The fourth number is the 2nd number plus the 3rd, which is 1 + 2. So the fourth number is 3. The 5th number is the 3rd number plus the 4th number: 2 + 3 = 5. This keeps going forever.
The first few numbers of the Fibonacci sequence are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
Because one number is defined by the numbers before it, this sets up a perfect opportunity for using recursion.
Your mission, should you choose to accept it, is to create a method called Fibonacci, which takes in a number and returns that number of the Fibonacci sequence. So if someone calls Fibonacci(3), it would return the 3rd number in the Fibonacci sequence, which is 2. If someone calls Fibonacci(8), it would return 21.
In your Mainmethod, write code to loop through the first 10 numbers of the Fibonacci sequence and print them out.
Hint #1:Start with your base case. We know that if it is the 1st or 2nd number, the value will be 1.
Hint #2:For every other item, how is it defined in terms of the numbers before it? Can you come up with an equation or formula that calls the Fibonaccimethod again?



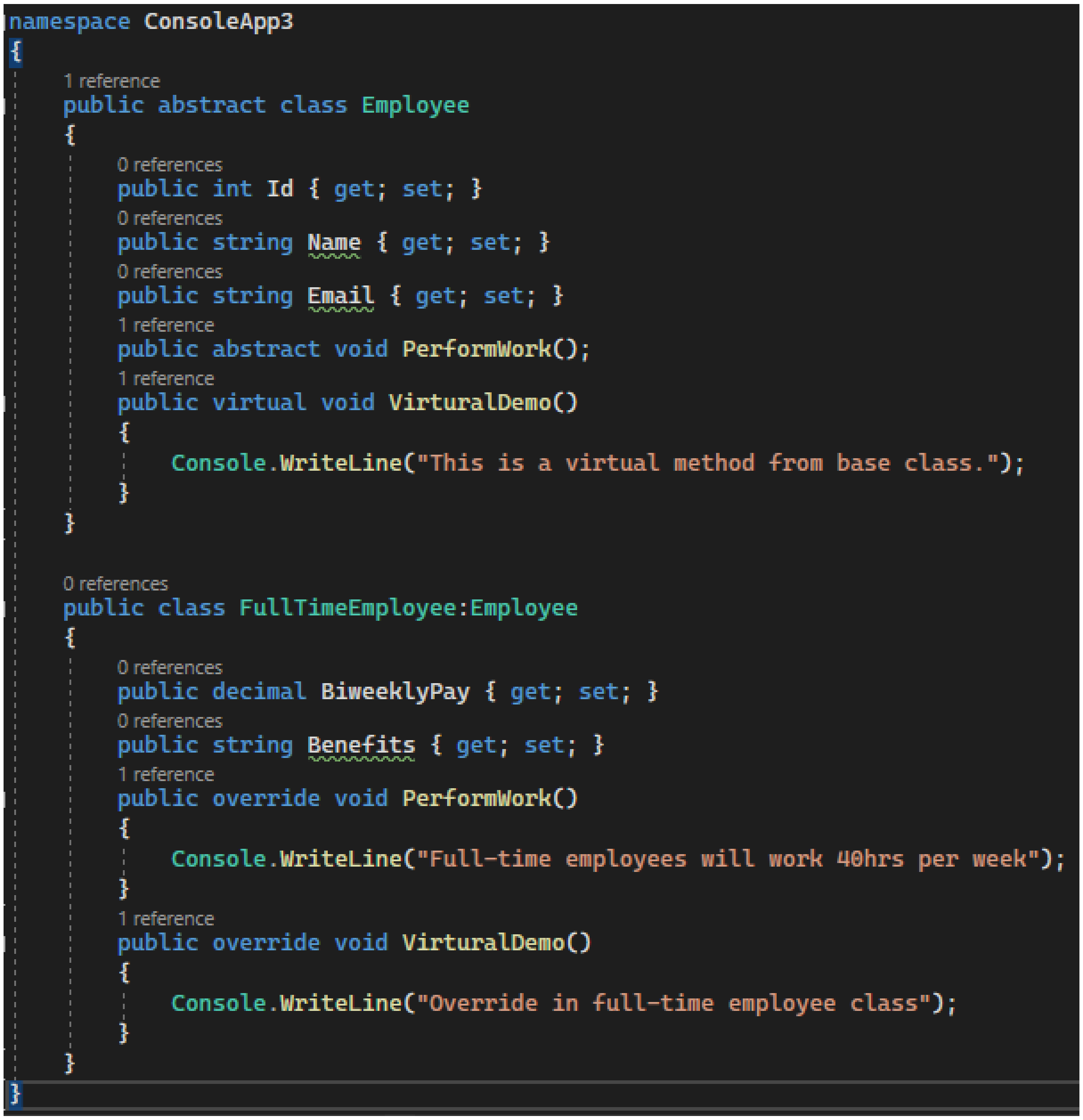
```
using System;

namespace ConsoleApp
{
    0 references
    public class Class1
    {
        4 references
        public static int Fibonacci(int k)
        {
            if (k == 1 || k == 2)
                return 1;
            else
                return Fibonacci(k - 1) + Fibonacci(k - 2);
        }
        0 references
        public static void Main()
        {
            int n = 0;
            Console.WriteLine("The first 10 numbers of the Fibonacci sequence are:");
            for (int i = 0; i < 10; i++)
                Console.Write("{0} ", Fibonacci(i+1));
            Console.WriteLine();
            Console.WriteLine("Enter a number:");
            n = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine($"The {n}th fibonacci number is {Fibonacci(n)}");
        }
    }
}
```



```
The first 10 numbers of the Fibonacci sequence are:
1 1 2 3 5 8 13 21 34 55
Enter a number:
5
The 5th fibonacci number is 5
```

Designing and Building Classes using object-oriented principles

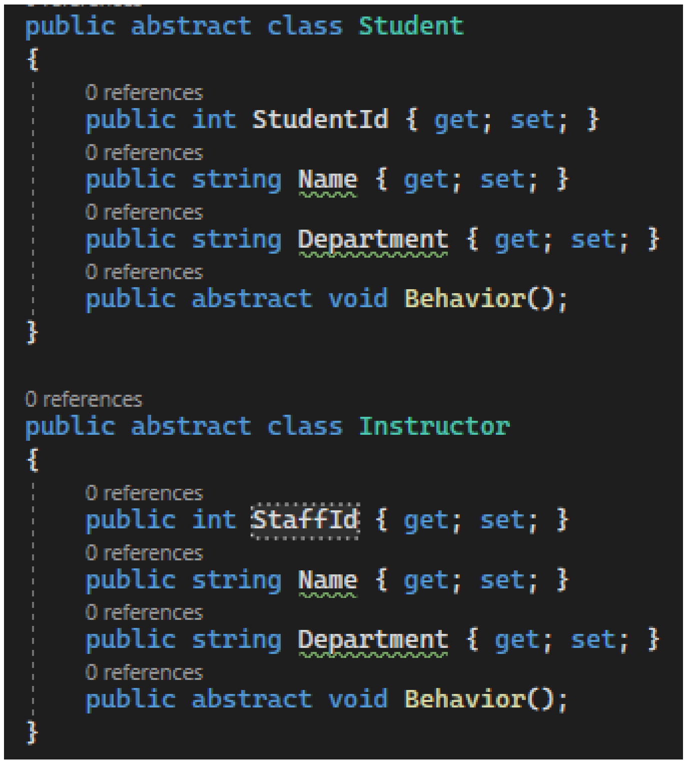
1. Write a program that that demonstrates use of four basic principles of object-oriented programming /Abstraction/, /Encapsulation/, /Inheritance/ and /Polymorphism/.



```
namespace ConsoleApp3
{
    1 reference
    public abstract class Employee
    {
        0 references
        public int Id { get; set; }
        0 references
        public string Name { get; set; }
        0 references
        public string Email { get; set; }
        1 reference
        public abstract void PerformWork();
        1 reference
        public virtual void VirturalDemo()
        {
            Console.WriteLine("This is a virtual method from base class.");
        }
    }

    0 references
    public class FullTimeEmployee:Employee
    {
        0 references
        public decimal BiweeklyPay { get; set; }
        0 references
        public string Benefits { get; set; }
        1 reference
        public override void PerformWork()
        {
            Console.WriteLine("Full-time employees will work 40hrs per week");
        }
        1 reference
        public override void VirturalDemo()
        {
            Console.WriteLine("Override in full-time employee class");
        }
    }
}
```

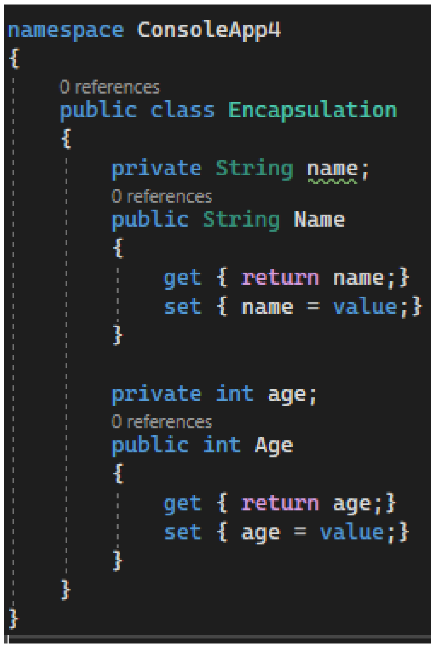
Employee class is an encapsulation that enables the employee properties and methods hiding from the user. The method PerformWork() is an abstract method which is in the abstract class. It doesn't have any body. FullTimeEmployee class is inherited from Employee class. It has all the members from the base class.
In FullTimeEmployee class, it has the method to overriding the method PerformWork() and VirturalDemo().

2. Use /Abstraction/ to define different classes for each person type such as Student and Instructor. These classes should have behavior for that type of person.



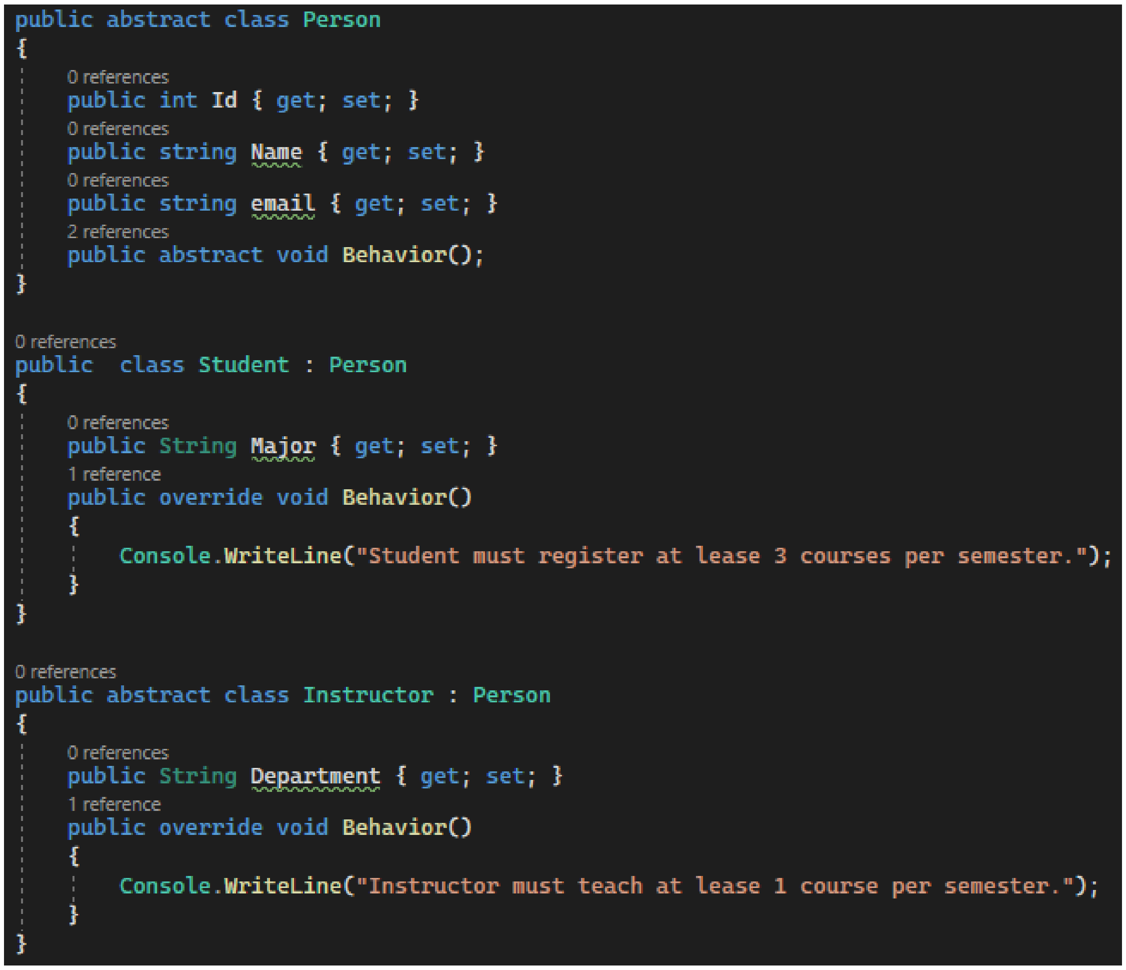
```
public abstract class Student
{
    0 references
    public int StudentId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Department { get; set; }
    0 references
    public abstract void Behavior();
}

0 references
public abstract class Instructor
{
    0 references
    public int StaffId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Department { get; set; }
    0 references
    public abstract void Behavior();
}
```

3. Use /Encapsulation/ to keep many details private in each class.



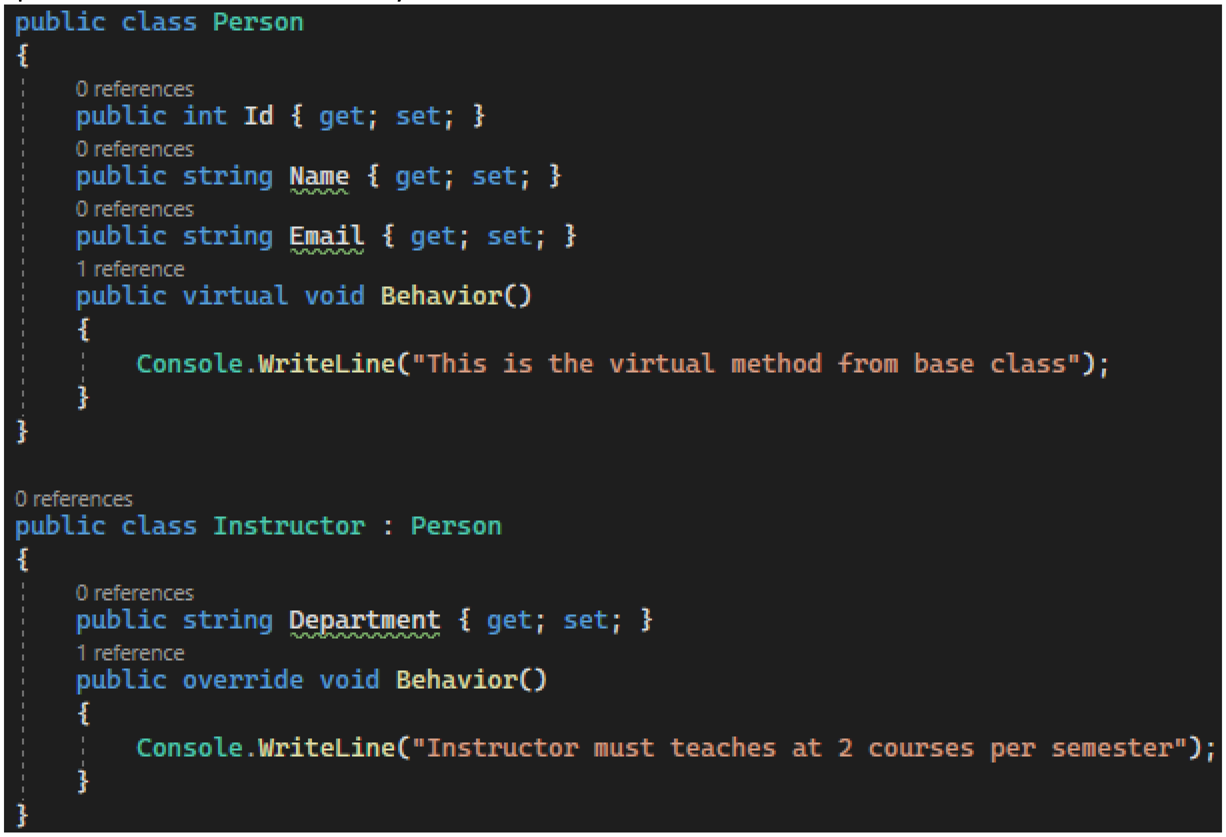
```
namespace ConsoleApp4
{
    0 references
    public class Encapsulation
    {
        private String name;
        0 references
        public String Name
        {
            get { return name;}
            set { name = value;}
        }

        private int age;
        0 references
        public int Age
        {
            get { return age;}
            set { age = value;}
        }
    }
}
```

4. Use /Inheritance/ by leveraging the implementation already created in the Person class to save code in Student and Instructor classes.



```
public abstract class Person
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string email { get; set; }
    2 references
    public abstract void Behavior();
}

0 references
public  class Student : Person
{
    0 references
    public String Major { get; set; }
    1 reference
    public override void Behavior()
    {
        Console.WriteLine("Student must register at lease 3 courses per semester.");
    }
}

0 references
public abstract class Instructor : Person
{
    0 references
    public String Department { get; set; }
    1 reference
    public override void Behavior()
    {
        Console.WriteLine("Instructor must teach at lease 1 course per semester.");
    }
}
```

5. Use /Polymorphism/ to create virtual methods that derived classes could override to create specific behavior such as salary calculations.



```
public class Person
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Email { get; set; }
    1 reference
    public virtual void Behavior()
    {
        Console.WriteLine("This is the virtual method from base class");
    }
}

0 references
public class Instructor : Person
{
    0 references
    public string Department { get; set; }
    1 reference
    public override void Behavior()
    {
        Console.WriteLine("Instructor must teaches at 2 courses per semester");
    }
}
```

6. Make sure to create appropriate /interfaces/ such as ICourseService, IStudentService, IInstructorService, IDepartmentService, IPersonService, IPersonService (should have person specific methods). IStudentService, IInstructorService should inherit from IPersonService.

Person

Calculate Age of the Person

Calculate the Salary of the person, Use decimal for salary

Salary cannot be negative number

Can have multiple Addresses, should have method to get addresses

Instructor

Belongs to one Department and he can be Head of the Department

Instructor will have added bonus salary based on his experience, calculate his years of experience based on Join Date

Student

Can take multiple courses

Calculate student GPA based on grades for courses

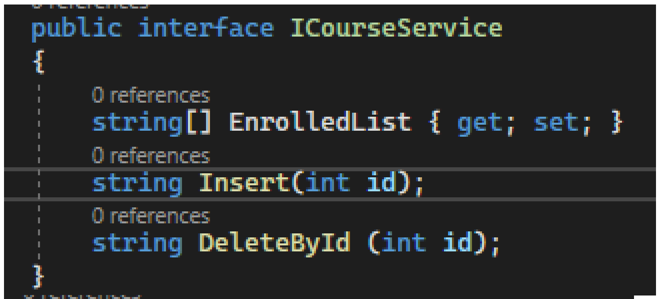
Each course will have grade from A to F

Course

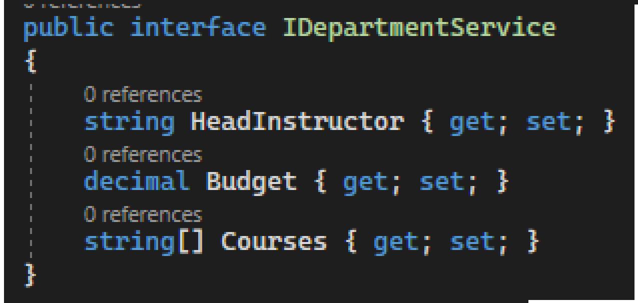
Will have list of enrolled students

Department

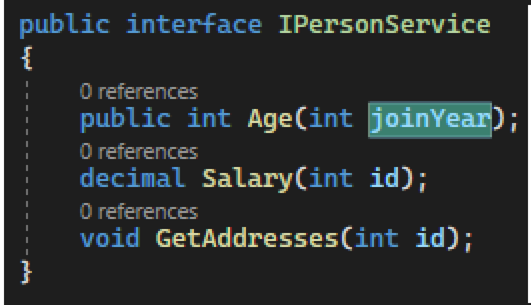
Will have one Instructor as head
Will have Budget for school year (start and end Date Time)
Will offer list of courses



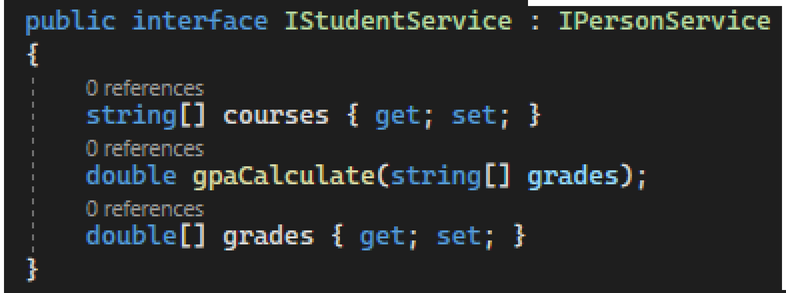
```
public interface ICourseService
{
    0 references
    string[] EnrolledList { get; set; }
    0 references
    string Insert(int id);
    0 references
    string DeleteById (int id);
}
```



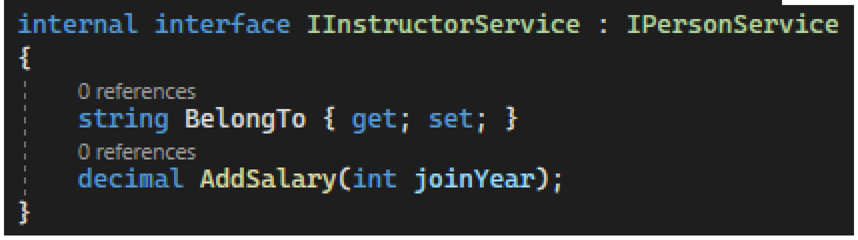
```
public interface IDepartmentService
{
    0 references
    string HeadInstructor { get; set; }
    0 references
    decimal Budget { get; set; }
    0 references
    string[] Courses { get; set; }
}
```



```
public interface IPersonService
{
    0 references
    public int Age(int joinYear);
    0 references
    decimal Salary(int id);
    0 references
    void GetAddresses(int id);
}
```

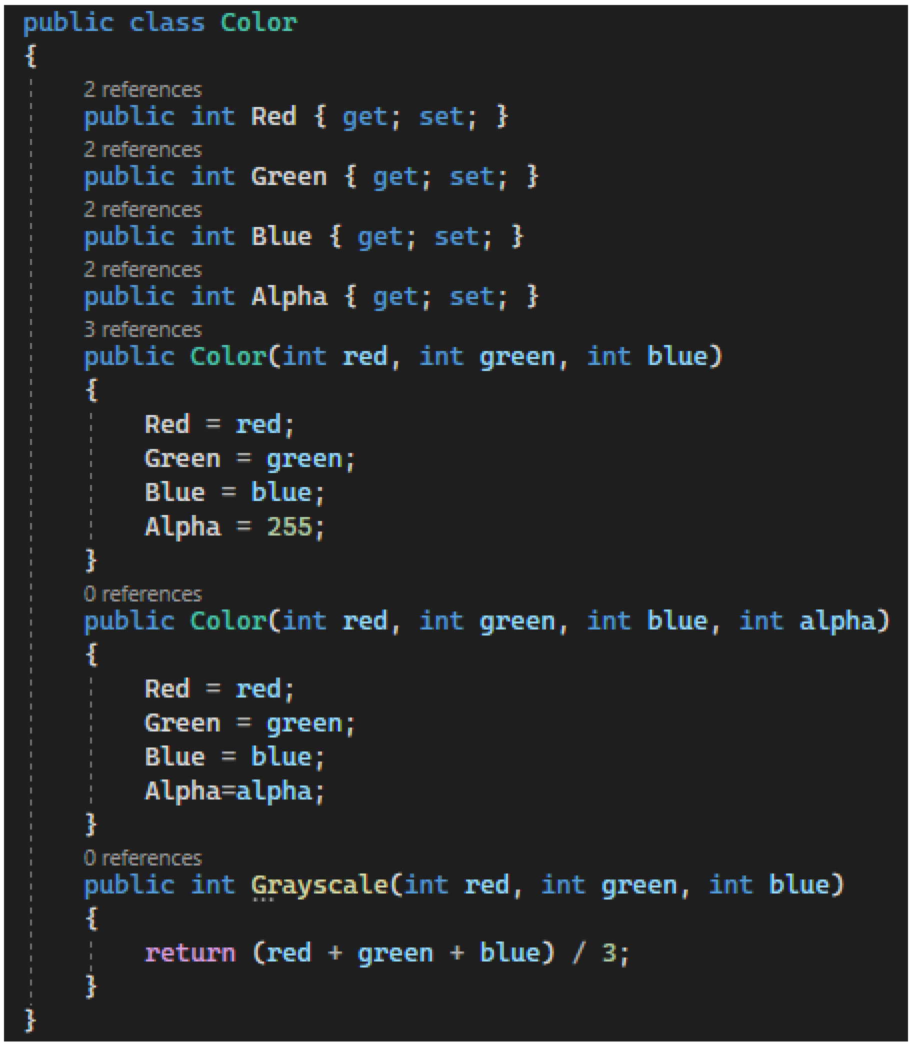


```
public interface IStudentService : IPersonService
{
    0 references
    string[] courses { get; set; }
    0 references
    double gpaCalculate(string[] grades);
    0 references
    double[] grades { get; set; }
}
```

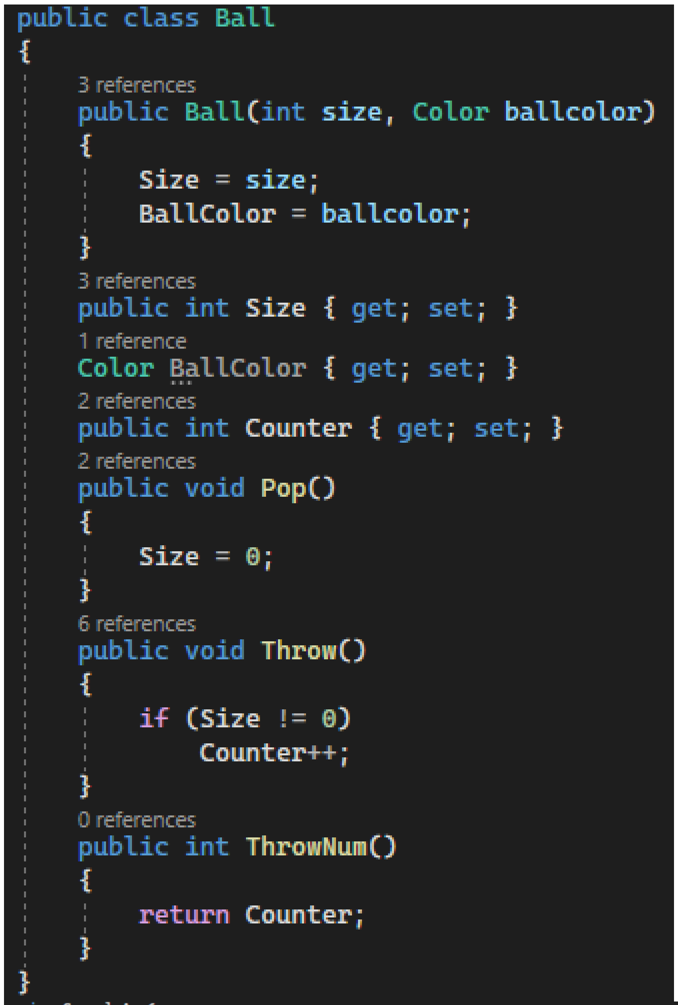


```
internal interface IInstructorService : IPersonService
{
    0 references
    string BelongTo { get; set; }
    0 references
    decimal AddSalary(int joinYear);
}
```

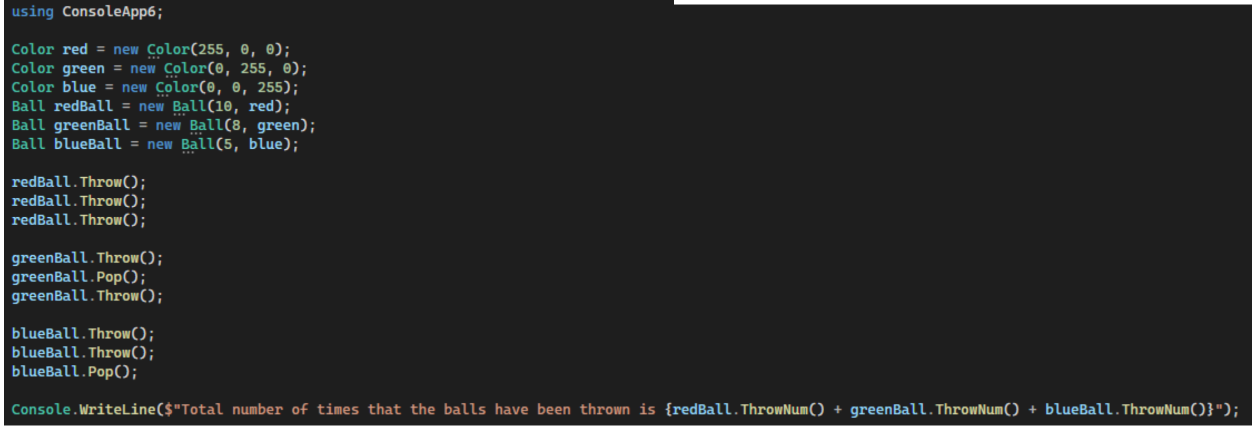
7. Try creating the two classes below, and make a simple program to work with them, as described below
Create a Color class:
On a computer, colors are typically represented with a red, green, blue, and alpha (transparency) value, usually in the range of 0 to 255. Add these as instance variables.
A constructor that takes a red, green, blue, and alpha value.
A constructor that takes just red, green, and blue, while alpha defaults to 255 (opaque).
Methods to get and set the red, green, blue, and alpha values from a Colorinstance.
A method to get the grayscale value for the color, which is the average of the red, green and blue values.
Create a Ball class:
The Ball class should have instance variables for size and color (the Color class you just created).
Let's also add an instance variable that keeps track of the number of times it has been thrown.
Create any constructors you feel would be useful.
Create a Pop method, which changes the ball's size to 0.
Create a Throw method that adds 1 to the throw count, but only if the ball hasn't been popped (has a size of 0).
A method that returns the number of times the ball has been thrown.
Write some code in your Main method to create a few balls, throw them around a few times, pop a few, and try to throw them again, and print out the number of times that the balls have been thrown. (Popped balls shouldn't have changed.)



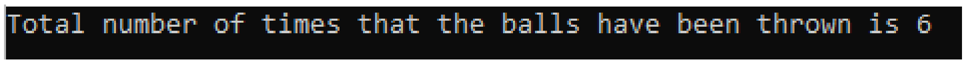
```
public class Color
{
    2 references
    public int Red { get; set; }
    2 references
    public int Green { get; set; }
    2 references
    public int Blue { get; set; }
    2 references
    public int Alpha { get; set; }
    3 references
    public Color(int red, int green, int blue)
    {
        Red = red;
        Green = green;
        Blue = blue;
        Alpha = 255;
    }
    0 references
    public Color(int red, int green, int blue, int alpha)
    {
        Red = red;
        Green = green;
        Blue = blue;
        Alpha=alpha;
    }
    0 references
    public int Grayscale(int red, int green, int blue)
    {
        return (red + green + blue) / 3;
    }
}
```



```
public class Ball
{
    3 references
    public Ball(int size, Color ballcolor)
    {
        Size = size;
        BallColor = ballcolor;
    }
    3 references
    public int Size { get; set; }
    1 reference
    Color BallColor { get; set; }
    2 references
    public int Counter { get; set; }
    2 references
    public void Pop()
    {
        Size = 0;
    }
    6 references
    public void Throw()
    {
        if (Size != 0)
            Counter++;
    }
    0 references
    public int ThrowNum()
    {
        return Counter;
    }
}
```



```
using ConsoleApp6;

Color red = new Color(255, 0, 0);
Color green = new Color(0, 255, 0);
Color blue = new Color(0, 0, 255);
Ball redBall = new Ball(10, red);
Ball greenBall = new Ball(8, green);
Ball blueBall = new Ball(5, blue);

redBall.Throw();
redBall.Throw();
redBall.Throw();

greenBall.Throw();
greenBall.Pop();
greenBall.Throw();

blueBall.Throw();
blueBall.Throw();
blueBall.Pop();

Console.WriteLine($"Total number of times that the balls have been thrown is {redBall.ThrowNum() + greenBall.ThrowNum() + blueBall.ThrowNum()}");
```



```
Total number of times that the balls have been thrown is 6
```