Assignment 3

Tianle Yuan

04/20/2023

03 Object-Oriented Programming

Test your knowledge

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

A: They are “public”, “protected internal”, “protected”, “internal”, “private protected”, “private”. We can use the picture shown by Kim:

Summary table

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

“public”: can be accessed by all the classes;
“protected internal”: can’t be accessed by non-derived class from other assemblies;
“protected”: can’t be accessed by all the non-derived classes;
“internal”: can’t be accessed by different assembly;
“private protected”: can only be accessed by inside of the class or derived class in the same assembly;
“private”: can only be accessed at the inside of the class.

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

A: “static”: the keywords used to define a member that belongs to the type itself rather than an instance of the type. We can access static member without creating an instance.
“const”: the keywords used to define a compile-time constant (its value must be assigned when it is

declared). “Implicitly static” -- we can access it without creating an instance.
“randomly”: the keywords used to define a “read-only” field -- the value can only be assigned during declaration or inside the constructor of the class. The value can be assigned at runtime. The field can be either an instance or a static members.

3. What does a constructor do?

A: A constructor is used to create instance of the class.
The constructor can be overloaded with multiple parameters.
If there is no constructor, the compiler provides default parameterless constructor.
Constructors do not have return type.
The derived class constructor will automatically make a call to the base class parameterless constructor.

4. Why is the partial keyword useful?

A: It split the definition of class, struct, or interface. The partial keyword helps to organize the code for maintenance; also is helpful for teamwork; save time for code regeneration.

5. What is a tuple?

A: The data structure that store a fixed number of elements which can be different type.

6. What does the C# record keyword do?

A: The keyword used to define immutable reference types with value semantics. It is helpful to create simple data object without complex behave.

7. What does overloading and overriding mean?

A: Overloading means the methods in both base class and subclass share same function name and parameters. Keywords: abstract, virtual in base class method. Overloading in derived class. Run-time polymorphism.
Overriding means the methods in base class and subclass have different function signatures. Compile-time polymorphism.

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

A: Field is the variable declared in class/ structure; usually private for encapsulation; it is used to store internal status of object.
Property is the member that provides access to the private field; usually public for encapsulation; we can use get/set combination to achieve different read/write authority.

9. How do you make a method parameter optional?

A: Use overloading. So that for different case different body of same function can be achieved.
Also you can provide default value for optional parameter to avoid input corresponding parameter.

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

A: Interface is a special class that only have public abstract methods; no fields. It do not have constructor and support multiple inheritance.

Abstract is a special class that at least have one public abstract methods; can have fields. It has constructor and do not support multiple inheritance.

11. What accessibility level are members of an interface?

A: Public and abstract.

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

A: 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.

A: 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.

A: False. It is method hiding. The method is totally different with base class.

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

A: False

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

A: True

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

A: True

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

A: True

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

A: 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.

A: False

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

A: True

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

A: True

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

A: 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.)

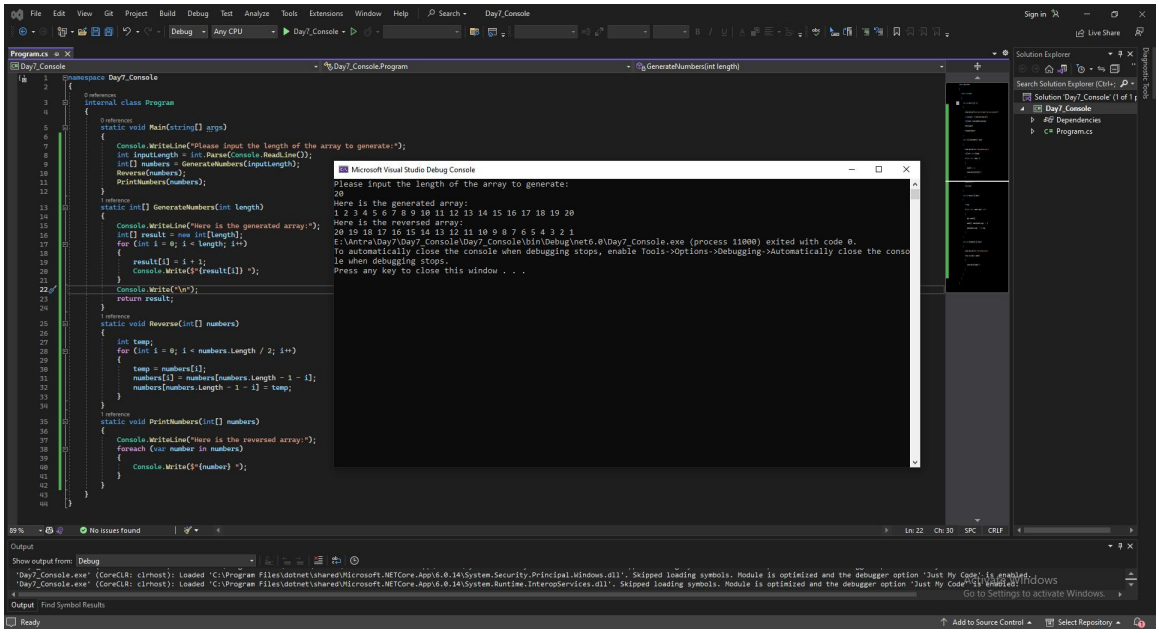
The PrintNumbersmethod should use a foror foreachloop to print out each item in the array. The Reversemethod will be the hardest. Give it a try and see what you can make happen. If you get stuck, here’s a couple of hints:

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].

A: Here below is the code I wrote:



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.

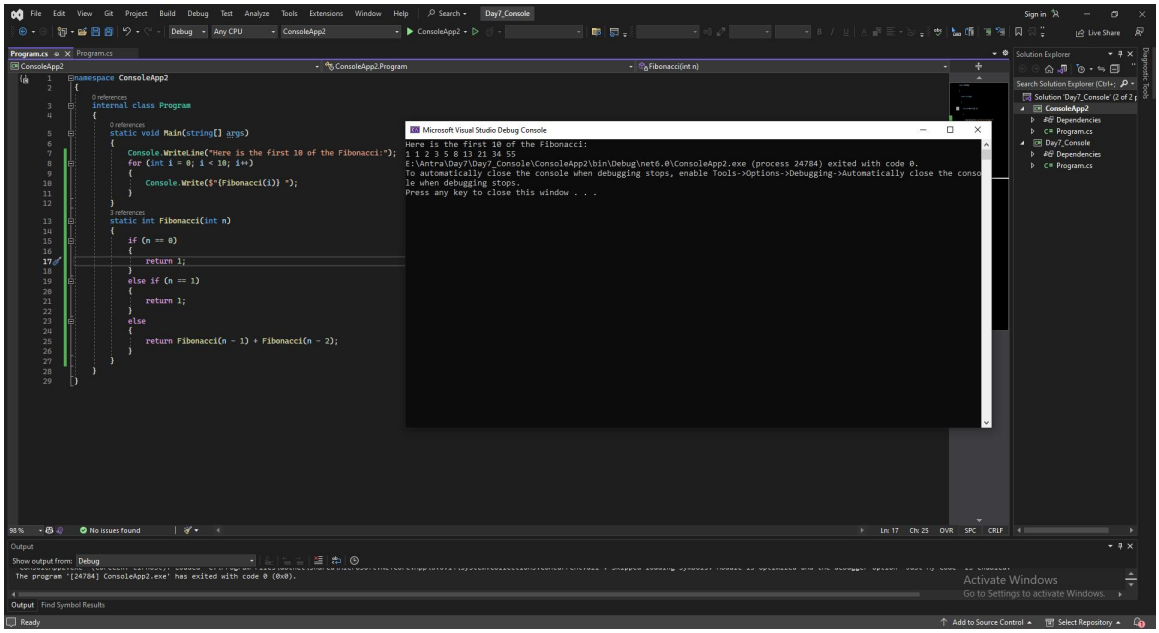
Your mission, should you choose to accept it, is to create a method called Fibonacci, which takes in a number and returns that number of the Fibonacci sequence. So if someone calls Fibonacci(3), it would return the 3rd number in the Fibonacci sequence, which is 2. If someone calls Fibonacci(8), it would 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?

A: Here below is the code I wrote:



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/.

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.

```
// Abstraction
3 references
abstract class Person
{
    // Encapsulation
    7 references
    public string Name { get; set; }
    5 references
    public DateTime Birthdate { get; set; }
    5 references
    public decimal Salary { get; set; }
    2 references
    public List<string> Addresses { get; private set; }

    0 references
    public Person()
    {
        Addresses = new List<string>();
    }

    0 references
    public void AddAddress(string address)
    {
        Addresses.Add(address);
    }
}

7 references
class Student : Person, IStudentService
{
    3 references
    public List<Course> Courses { get; private set; }

    2 references
    public Student()
    {
        Courses = new List<Course>();
    }

    3 references
    public void AddCourse(Course course)
    {
        Courses.Add(course);
    }
```

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

```
abstract class Person
{
    // Encapsulation
    7 references
    public string Name { get; set; }
    5 references
    public DateTime Birthdate { get; set; }
    5 references
    public decimal Salary { get; set; }
    2 references
    public List<string> Addresses { get; private set; }
```

```
7 references
class Student : Person, IStudentService
{
    3 references
    public List<Course> Courses { get; private set; }
```

```
3 references
class Instructor : Person, IInstructorService
{
    2 references
    public DateTime JoinDate { get; set; }
    1 reference
    public Department Department { get; set; }
    1 reference
    public bool IsDepartmentHead { get; set; }
```

```
11 references
class Course
{
    2 references
    public string Name { get; set; }
    4 references
    public int Credits { get; set; }
    3 references
    public float GradePoints { get; set; }
    1 reference
    public List<Student> EnrolledStudents { get; private set; }

    2 references
    public Course()
    {
        EnrolledStudents = new List<Student>();
    }
}

4 references
class Department
{
    1 reference
    public string Name { get; set; }
    1 reference
    public Instructor Head { get; set; }
    1 reference
    public decimal Budget { get; set; }
    1 reference
    public DateTime SchoolYearStart { get; set; }
    1 reference
    public DateTime SchoolYearEnd { get; set; }
    3 references
    public List<Course> OfferedCourses { get; private set; }
```

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

```
// Interfaces
2 references
public interface IPersonService
{
    2 references
    int CalculateAge();
    3 references
    decimal CalculateSalary();
}

1 reference
public interface IStudentService : IPersonService
{
    3 references
    float CalculateGPA();
}

1 reference
public interface IInstructorService : IPersonService
{
    3 references
    int CalculateExperience();
}
```

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

```
3 references
class Instructor : Person, IInstructorService
{
    2 references
    public DateTime JoinDate { get; set; }
    1 reference
    public Department Department { get; set; }
    1 reference
    public bool IsDepartmentHead { get; set; }

    1 reference
    public int CalculateAge()
    {
        int age = DateTime.Now.Year - Birthdate.Year;
        return age;
    }

    2 references
    public decimal CalculateSalary()
    {
        return Salary + CalculateBonus();
    }

    // Polymorphism
    3 references
    public int CalculateExperience()
    {
        int experience = DateTime.Now.Year - JoinDate.Year;
        return experience;
    }

    1 reference
```

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.

```
// Interfaces
2 references
public interface IPersonService
{
    2 references
    int CalculateAge();
    3 references
    decimal CalculateSalary();
}

1 reference
public interface IStudentService : IPersonService
{
    3 references
    float CalculateGPA();
}

1 reference
public interface IInstructorService : IPersonService
{
    3 references
    int CalculateExperience();
}
```

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

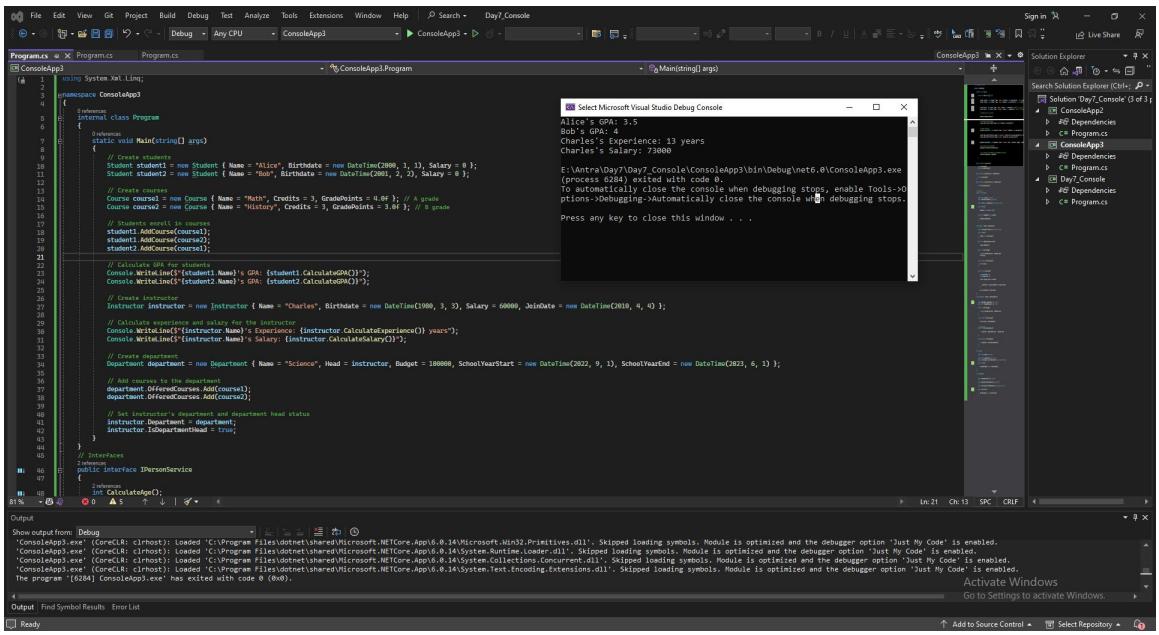
Department

Will have one Instructor as head

Will have Budget for school year (start and end Date Time)

Will offer list of courses

A: Here below is the code I wrote:



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.

```
9 references
public class Color
{
    2 references
    public int Red { get; set; }
    2 references
    public int Green { get; set; }
    2 references
    public int Blue { get; set; }
    1 reference
    public int Alpha { get; set; }

    2 references
    public Color(int red, int green, int blue, int alpha)
    {
        Red = red;
        Green = green;
        Blue = blue;
        Alpha = alpha;
    }

    1 reference
    public Color(int red, int green, int blue) : this(red, green, blue, 255) { }

    0 references
    public int GetGrayscale()
    {
        return (Red + Green + Blue) / 3;
    }
}
```

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.

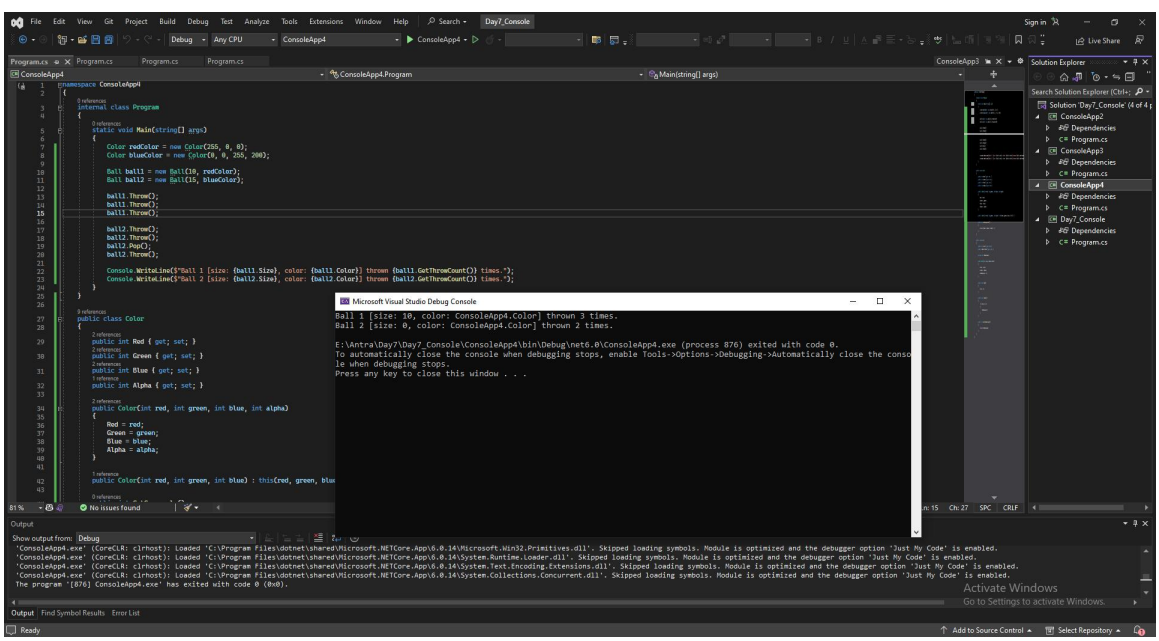
Create a Throw method that adds 1 to the throw count, but only if the ball hasn’t been popped (has a size of 0).

A method that returns the number of times the ball has been thrown.

```
5 references
public class Ball
{
    5 references
    public int Size { get; set; }
    3 references
    public Color Color { get; set; }

    private int throwCount;

    2 references
    public Ball(int size, Color color)
    {
        Size = size;
        Color = color;
        throwCount = 0;
    }

    1 reference
    public void Pop()
    {
        Size = 0;
    }

    6 references
    public void Throw()
    {
        if (Size != 0)
        {
            throwCount++;
        }
    }

    2 references
    public int GetThrowCount()
    {
        return throwCount;
    }
}
}
```

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.)

A: Here below is the code I wrote:



Explore following topics

- **Fields**
- **Access modifiers**
- **Enumeration types**
- **Constructors**
- **Methods**
- **Properties**
- **Inheritance**
- **Interfaces**