

## 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 Main method will look something like this:

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

The GenerateNumbers method 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 PrintNumbers method should use a for or foreach loop to print out each item in the array. The Reverse method 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;
```

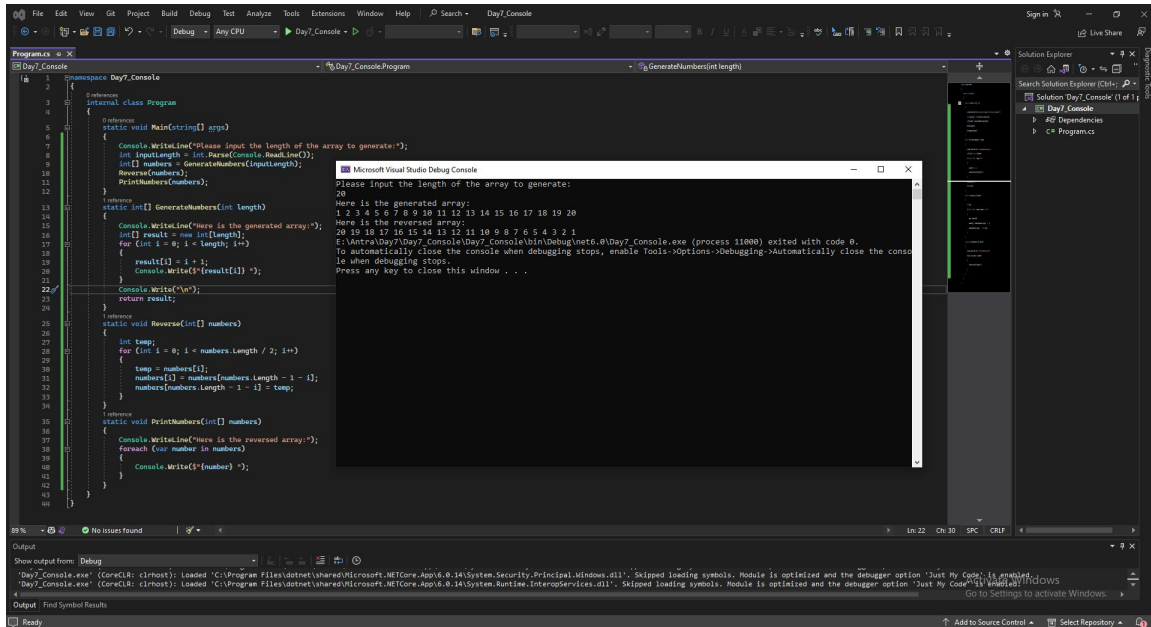
```
int temp = a;
```

```
a = b;
```

```
b = temp;
```

Hint #2: Getting the right indices to swap can be a challenge. Use a for loop, starting at 0 and going up to the length of the array / 2. The number you use in the for loop 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.

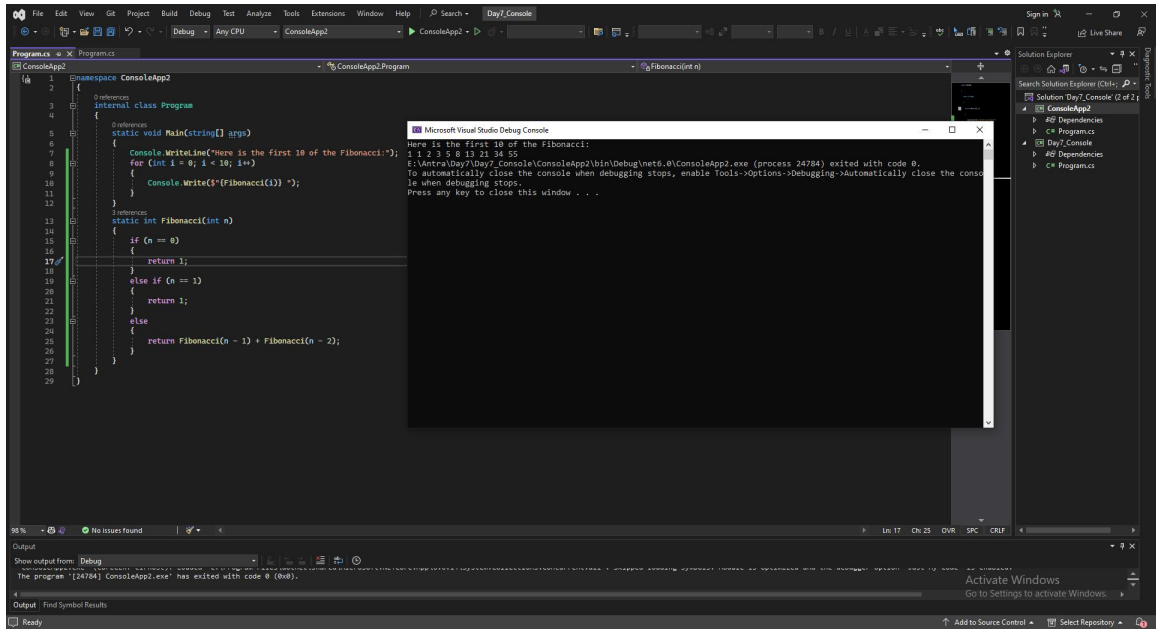
Your mission, should you choose to accept it, is to create a method called `Fibonacci`, which takes in a number and returns that number of the Fibonacci sequence. So if someone calls `Fibonacci(3)`, it would return the 3rd number in the Fibonacci sequence, which is 2. If someone calls `Fibonacci(8)`, it would return 21.

In your `Main` method, 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 `Fibonacci` method again?

A: Here below is the code I wrote:



## Designing and Building Classes using object-oriented principles

1. Write a program 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.

```

62 // Abstraction
63 3 references
64 abstract class Person
65 {
66     // Encapsulation
67     7 references
68     public string Name { get; set; }
69     5 references
70     public DateTime Birthdate { get; set; }
71     5 references
72     public decimal Salary { get; set; }
73     2 references
74     public List<string> Addresses { get; private set; }
75
76     0 references
77     public Person()
78     {
79         Addresses = new List<string>();
80     }
81
82     0 references
83     public void AddAddress(string address)
84     {
85         Addresses.Add(address);
86     }
87 }
88
89 7 references
90 class Student : Person, IStudentService
91 {
92     3 references
93     public List<Course> Courses { get; private set; }
94
95     2 references
96     public Student()
97     {
98         Courses = new List<Course>();
99     }
100
101     3 references
102     public void AddCourse(Course course)
103     {
104         Courses.Add(course);
105     }
106 }

```

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

```

63 abstract class Person
64 {
65     // Encapsulation
66     7 references
67     public string Name { get; set; }
68     5 references
69     public DateTime Birthdate { get; set; }
70     5 references
71     public decimal Salary { get; set; }
72     2 references
73     public List<string> Addresses { get; private set; }
74 }
75
76 7 references
77 class Student : Person, IStudentService
78 {
79     3 references
80     public List<Course> Courses { get; private set; }
81
82     2 references
83     public Student()
84     {
85         Courses = new List<Course>();
86     }
87
88     3 references
89     public void AddCourse(Course course)
90     {
91         Courses.Add(course);
92     }
93 }
94
95 3 references
96 class Instructor : Person, IInstructorService
97 {
98     2 references
99     public DateTime JoinDate { get; set; }
100     1 reference
101     public Department Department { get; set; }
102     1 reference
103     public bool IsDepartmentHead { get; set; }
104 }

```

```

154 11 references
155 class Course
156 {
157     2 references
158     public string Name { get; set; }
159     4 references
160     public int Credits { get; set; }
161     3 references
162     public float GradePoints { get; set; }
163     1 reference
164     public List<Student> EnrolledStudents { get; private set; }
165
166     2 references
167     public Course()
168     {
169         EnrolledStudents = new List<Student>();
170     }
171
172     4 references
173     class Department
174     {
175         1 reference
176         public string Name { get; set; }
177         1 reference
178         public Instructor Head { get; set; }
179         1 reference
180         public decimal Budget { get; set; }
181         1 reference
182         public DateTime SchoolYearStart { get; set; }
183         1 reference
184         public DateTime SchoolYearEnd { get; set; }
185         3 references
186         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.

```

45 // Interfaces
46 2 references
47 public interface IPersonService
48 {
49     2 references
50     int CalculateAge();
51     3 references
52     decimal CalculateSalary();
53 }
54
55 1 reference
56 public interface IStudentService : IPersonService
57 {
58     3 references
59     float CalculateGPA();
60 }
61
62 1 reference
63 public interface IInstructorService : IPersonService
64 {
65     3 references
66     int CalculateExperience();
67 }

```

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



```

122
123 3 references
124 class Instructor : Person, IInstructorService
125 {
126     2 references
127     public DateTime JoinDate { get; set; }
128     1 reference
129     public Department Department { get; set; }
130     1 reference
131     public bool IsDepartmentHead { get; set; }
132
133     1 reference
134     public int CalculateAge()
135     {
136         int age = DateTime.Now.Year - Birthdate.Year;
137         return age;
138     }
139
140     3 references
141     public decimal CalculateSalary()
142     {
143         return Salary + CalculateBonus();
144     }
145
146     // Polymorphism
147     3 references
148     public int CalculateExperience()
149     {
150         int experience = DateTime.Now.Year - JoinDate.Year;
151         return experience;
152     }
153
154     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.

```

45 // Interfaces
46 2 references
47 public interface IPersonService
48 {
49     2 references
50     int CalculateAge();
51     3 references
52     decimal CalculateSalary();
53 }
54
55 1 reference
56 public interface IStudentService : IPersonService
57 {
58     3 references
59     float CalculateGPA();
60 }
61
62 1 reference
63 public interface IInstructorService : IPersonService
64 {
65     3 references
66     int CalculateExperience();
67 }

```

**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

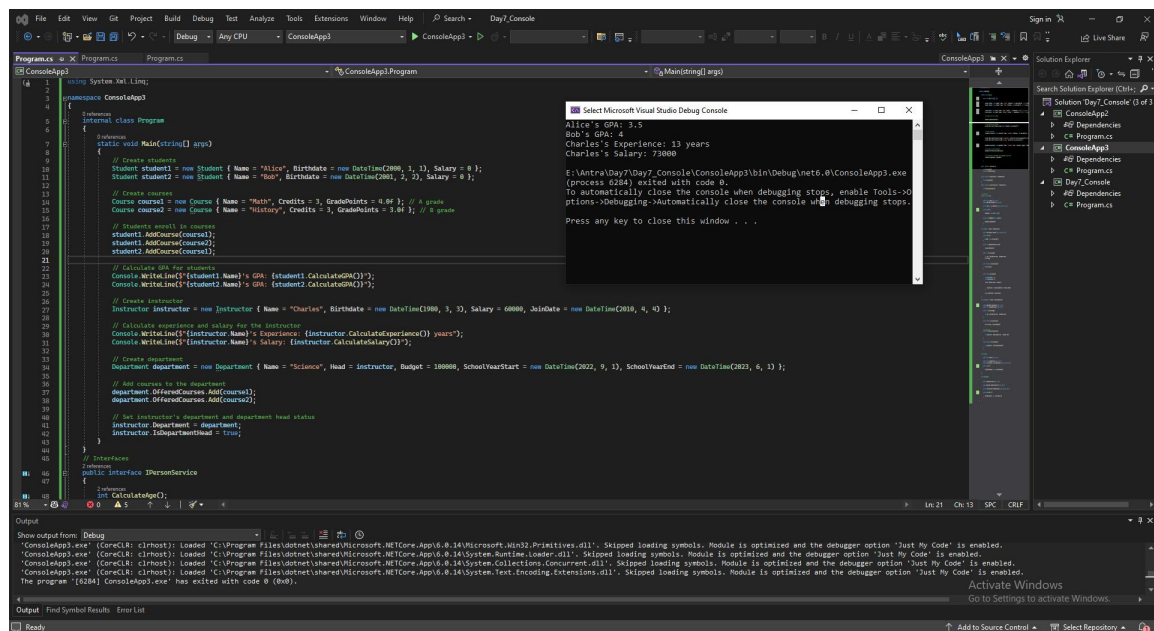
Department

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:



```
using System;
using System.Linq;

namespace ConsoleApp3
{
    internal class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            // Create student
            Student student1 = new Student { Name = "Alice", BirthDate = new DateTime(2000, 1, 1), Salary = 0 };
            Student student2 = new Student { Name = "Bob", BirthDate = new DateTime(2001, 2, 2), Salary = 0 };

            // Create course
            Course course1 = new Course { Name = "Math", Credits = 3, GradePoints = 4.0f }; // A grade
            Course course2 = new Course { Name = "History", Credits = 3, GradePoints = 3.0f }; // B grade

            student1.AddCourse(course1);
            student1.AddCourse(course2);
            student2.AddCourse(course1);

            // Calculate GPA for students
            Console.WriteLine($"Student1 Name: {student1.Name} GPA: {student1.CalculateGPA()}");
            Console.WriteLine($"Student2 Name: {student2.Name} GPA: {student2.CalculateGPA()}");

            // Create instructor
            Instructor instructor = new Instructor { Name = "Charles", BirthDate = new DateTime(1980, 3, 15), Salary = 60000, JoinDate = new DateTime(2010, 4, 10) };

            // Calculate experience and salary due the instructor
            Console.WriteLine($"Instructor Name: {instructor.Name} Experience: {instructor.CalculateExperience()} years");
            Console.WriteLine($"Instructor Name: {instructor.Name} Salary: {instructor.CalculateSalary()}");

            // Create department
            Department department = new Department { Name = "Science", Head = instructor, Budget = 300000, SchoolYearStart = new DateTime(2022, 9, 1), SchoolYearEnd = new DateTime(2023, 6, 1) };

            // Add courses to the department
            department.OfferedCourses.Add(course1);
            department.OfferedCourses.Add(course2);

            // Get instructor's department and department head status
            instructor.Department = department;
            instructor.IsDepartmentHead = true;

            // Interfaces
            public interface IPersonService
            {
                // ...
            }

            // ...
        }
    }
}
```

Output:

```
Show output from: Debug
ConsoleApp3.exe (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\Microsoft.Win32.Primitives.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
ConsoleApp3.exe (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\System.Runtime.Loader.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
ConsoleApp3.exe (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\System.Collections.Concurrent.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
ConsoleApp3.exe (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
The program '[6284] ConsoleApp3.exe' has exited with code 0 (0x0).
```

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.

```

27 9 references
28 public class Color
29 {
30     2 references
31     public int Red { get; set; }
32     2 references
33     public int Green { get; set; }
34     2 references
35     public int Blue { get; set; }
36     1 reference
37     public int Alpha { get; set; }
38
39     2 references
40     public Color(int red, int green, int blue, int alpha)
41     {
42         Red = red;
43         Green = green;
44         Blue = blue;
45         Alpha = alpha;
46     }
47
48     1 reference
49     public Color(int red, int green, int blue) : this(red, green, blue, 255) { }
50
51     0 references
52     public int GetGrayscale()
53     {
54         return (Red + Green + Blue) / 3;
55     }
56 }

```

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.

```

50 5 references
51 public class Ball
52 {
53     5 references
54     public int Size { get; set; }
55     3 references
56     public Color Color { get; set; }
57
58     private int throwCount;
59
60     2 references
61     public Ball(int size, Color color)
62     {
63         Size = size;
64         Color = color;
65         throwCount = 0;
66     }
67
68     1 reference
69     public void Pop()
70     {
71         Size = 0;
72     }
73
74     6 references
75     public void Throw()
76     {
77         if (Size != 0)
78         {
79             throwCount++;
80         }
81     }
82
83     2 references
84     public int GetThrowCount()
85     {
86         return throwCount;
87     }
88 }

```

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:

```

1  namespace ConsoleApp1
2  {
3      internal class Program
4      {
5          static void Main(string[] args)
6          {
7              Color redColor = new Color(255, 0, 0);
8              Color blueColor = new Color(0, 255, 255);
9
10             Ball ball1 = new Ball(10, redColor);
11             Ball ball2 = new Ball(0, blueColor);
12
13             ball1.Throw();
14             ball1.Throw();
15             ball1.Throw();
16
17             ball2.Throw();
18             ball2.Throw();
19             ball2.Throw();
20
21             Console.WriteLine("Ball 1 [size: {ball1.Size}, color: {ball1.Color}] thrown {ball1.GetThrowCount()} times.");
22             Console.WriteLine("Ball 2 [size: {ball2.Size}, color: {ball2.Color}] thrown {ball2.GetThrowCount()} times.");
23         }
24     }
25 }
26
27 namespace ConsoleApp1
28 {
29     public class Color
30     {
31         2 references
32         public int Red { get; set; }
33         2 references
34         public int Green { get; set; }
35         2 references
36         public int Blue { get; set; }
37         2 references
38         public int Alpha { get; set; }
39
40         2 references
41         public Color(int red, int green, int blue, int alpha)
42         {
43             Red = red;
44             Green = green;
45             Blue = blue;
46             Alpha = alpha;
47         }
48
49         1 reference
50         public Color(int red, int green, int blue) : this(red, green, blue, 255)
51         {
52         }
53     }
54 }

```

Microsoft Visual Studio Debug Console

```

Ball 1 [size: 10, color: ConsoleApp1.Color] thrown 3 times.
Ball 2 [size: 0, color: ConsoleApp1.Color] thrown 2 times.
E:\Antra\Day7\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 876) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Output

```

Show output from: Debug
'ConsoleApp1.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\Microsoft.Win32.Primitives.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'ConsoleApp1.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\System.Runtime.Loader.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'ConsoleApp1.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'ConsoleApp1.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.14\System.Collections.Concurrent.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
The program '[876] ConsoleApp1.exe' has exited with code 0 (0x0).

```

## **Explore following topics**

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