1. 1) public: no restrictions on who can access it;
   2) private: can only be accessed within the class or struct where it is declared;
   3) protected: limited to the current class and its derived class;
   4) internal: can only accessed within the current assembly (e.g., .exe file);
   5) protected internal: can only be accessed within the current assembly or from the derived class;
   6) private protected: can only be accessed by the current class and its derived class within the same assembly.
2. 1) static: belongs to the type, not instances, and exits for the lifetime of the application domain; does not by itself limit if the value is mutable;
   2) const: value is set at compile time and cannot be changed; cannot be applied to complex data types;
   3) readonly: value is set at runtime either by declaration or in the constructor; ca e applied to complex data types.
3. The constructor initializes objects and initializes type members.
4. The partial keyword allows a single class, struct, interface, or method to be split across multiple files, which is useful for managing large codebases and flexibly organizing or integrating codes from different sources.
5. Tuple is a data structure to group multiple values, of the same type or not, together without having to create a class or struct. It holds a fixed number of items that can be named and modified. It is useful as the return value from functions.
6. The record keyword defines a reference type that provides built-in functionality for encapsulating data.
7. Overloading allows multiple methods in the same class to have the same name but different sets of parameters, which is a form of compile-time (static) polymorhism; overriding allows a subclass to provide a specific (override) implementation of a (virtual) method defined in its superclass, which enables runtime (dynamic) polymorphism.
8. A field allows direct access to data and cannot include additional logic for getting or setting values; a property encapsulates the data and have controlled read and write access via accessors that may include additional logic.
9. A method parameter becomes optional when a default value for it is provided in the method signature.
10. An interface defines a set of members (methods, properties, events, indexers) that only include their signatures and a class or struct must implement. Different from interfaces, abstract classes can include concrete methods and can be instantiated when all abstract members are implemented, despite that a class cannot inherit multiple classes as it can implement multiple interfaces.
11. Members of an interface are always implicitly public.
12. True. This is runtime polymorphism.
13. True.
14. False. The “new” keyword hides the derived class’s implementation from the base class, so that the base class’s version of the method is invoked in the scenario of (type) upcasting.
15. False. Abstract methods can only exist in abstract classes or interfaces, while non-abstract classes must provide concrete implementations for abstract methods in their base abstract

classes.

16. True.
17. True.
18. True.
19. False.
20. False.
21. True.
22. False.
23. True.

1.

```
using System;

class Program
{
	static void Main()
	{
		int[] numbers = GenerateNumbers();
		Reverse(numbers);
		PrintNumbers(numbers);
	}

	// Method to create an array of integers from 1 to 10
	static int[] GenerateNumbers()
	{
		int[] numbers = new int[10];
		for (int i = 0; i < 10; i++)
		{
			numbers[i] = i + 1;
		}
		return numbers;
	}

	// Method to reverse the elements of an array
	static void Reverse(int[] arr)
	{
		Array.Reverse(arr);
	}

	// Method to print the elements of an array
	static void PrintNumbers(int[] arr)
	{
		Console.WriteLine("Array elements:");
		foreach (int num in arr)
```

```
            {
                Console.Write(num + " ");
            }
            Console.WriteLine();
        }
}
```

```
Array elements:
10 9 8 7 6 5 4 3 2 1

=== Code Execution Successful ===
```

Bonus:

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = GenerateNumbers();
        Reverse(numbers);
        PrintNumbers(numbers);
    }

    // Method to create an array of integers from 1 to n
    static int[] GenerateNumbers(int n = 5)
    {
        int[] numbers = new int[n];
        for (int i = 0; i < n; i++)
        {
            numbers[i] = i + 1;
        }
        return numbers;
    }

    // Method to reverse the elements of an array using a for loop
    static void Reverse(int[] arr)
    {
        int length = arr.Length;
        for (int i = 0; i < length / 2; i++)
        {
            int temp = arr[i];
            arr[i] = arr[length - i - 1];
            arr[length - i - 1] = temp;
        }
    }
```

```
    // Method to print the elements of an array
    static void PrintNumbers(int[] arr)
    {
        Console.WriteLine("Array elements:");
        foreach (int num in arr)
        {
            Console.Write(num + " ");
        }
        Console.WriteLine();
    }
}
```

```
Array elements:
5 4 3 2 1

=== Code Execution Successful ===
```

2.

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        // Example usage
        foreach(int n in Enumerable.Range(1,8)){
            int result = Fibonacci(n);
            Console.WriteLine($"The {n}-th number in the Fibonacci sequence is: {result}");
        }
    }

    static int Fibonacci(int n)
    {
        // Base case: n equals 1 or 2
        if (n == 1 || n == 2)
        {
            return 1;
        }

        int prev = 1;
        int current = 1;
```

```
            for (int i = 3; i <= n; i++)
            {
                int next = prev + current;
                prev = current;
                current = next;
            }

            return current;
        }
}
```

```
The 1-th number in the Fibonacci sequence is: 1
The 2-th number in the Fibonacci sequence is: 1
The 3-th number in the Fibonacci sequence is: 2
The 4-th number in the Fibonacci sequence is: 3
The 5-th number in the Fibonacci sequence is: 5
The 6-th number in the Fibonacci sequence is: 8
The 7-th number in the Fibonacci sequence is: 13
The 8-th number in the Fibonacci sequence is: 21

=== Code Execution Successful ===
```

1.

```
using System;

class Mother
{
    // Abstraction
    public virtual void Say()
    {
        Console.WriteLine("Mother says yes.");
    }

    // Abstraction
    public virtual void Say(string word = "no")
    {
        Console.WriteLine($"Mother says {word}.");
    }
}

class Daughter : Mother
{
    // Inheritance, Polymorphism
    public override void Say()
    {
        Console.WriteLine("Daughter says yes.");
    }
```

```
}

class Program
{
	static void Main()
	{
		// Encapsulation
		Mother mom = new Mother();
		Daughter daughter = new Daughter();

		// Polymorphism
		mom.Say(); // Output: Mother says yes.
		mom.Say("maybe"); // Output: Mother says maybe.
		daughter.Say(); // Output: Daughter says yes.
		daughter.Say("perhaps"); // Output: Mother says perhaps.
	}
}
```

2 ~ 6.

```
using System;
using System.Collections.Generic;

// Interfaces
interface IPersonService
{
	void Behave();
}

interface IStudentService
{
	void Run();
}

interface IInstructorService : IPersonService
{
}

interface IDepartmentService
{
	void Run();
}

interface ICourseService
{
```

```
}

// Person class
class Person : IPersonService
{
	private string _name;
	private DateTime _birthDate;

	public string Name
	{
		get { return _name; }
		set { _name = value; }
	}

	public int Age
	{
		get { return CalculateAge(); }
	}

	public virtual decimal Salary { get; set; }

	private List<string> _addresses = new List<string>();

	public void Behave()
	{
		Console.WriteLine($"The {_name} acts as a Person.");
	}

	public void SetBirthDate(DateTime birthDate)
	{
		_birthDate = birthDate;
	}

	public void AddAddress(string address)
	{
		_addresses.Add(address);
	}

	public List<string> GetAllAddresses()
	{
		return _addresses;
	}

	private int CalculateAge()
```

```
        {
            DateTime now = DateTime.Now;
            int age = now.Year - _birthDate.Year;
            if (now < _birthDate.AddYears(age))
            {
                age--;
            }
            return age;
        }
}

// Instructor class
class Instructor : Person
{
        private Department _department;
        private DateTime _joinDate;

        public Department Department
        {
            get { return _department; }
            set { _department = value; }
        }

        public DateTime JoinDate
        {
            get { return _joinDate; }
            set { _joinDate = value; }
        }

        public override decimal Salary
        {
            get { return base.Salary; }
            set
            {
                int yearsOfExperience = CalculateYearsOfExperience();
                base.Salary = value + 100 * yearsOfExperience;
            }
        }

        public Instructor(DateTime joinDate)
        {
            _joinDate = joinDate;
        }
```

```
    private int CalculateYearsOfExperience()
    {
        DateTime now = DateTime.Now;
        int years = now.Year - _joinDate.Year;
        if (now < _joinDate.AddYears(years))
        {
            years--;
        }
        return years;
    }
}

// Student class
class Student : Person
{
    private List<Course> _courses = new List<Course>();

    public decimal GPA
    {
        get { return CalculateGPA(); }
    }

    public void AddCourseGrade(Course course, int credits, char grade)
    {
        course.AddStudentGrade(this, credits, grade);
        _courses.Add(course);
    }

    private decimal CalculateGPA()
    {
        if (_courses.Count == 0)
            return 0;

        decimal totalGradePoints = 0;
        decimal totalCredits = 0;

        foreach (var course in _courses)
        {
            var gradeInfo = course.StudentGrades[this];
            int credits = gradeInfo.credits;
            char grade = gradeInfo.grade;

            decimal gradePoints;
            switch (grade)
```

```
                {
                    case 'A':
                        gradePoints = 4;
                        break;
                    case 'B':
                        gradePoints = 3;
                        break;
                    case 'C':
                        gradePoints = 2;
                        break;
                    case 'D':
                        gradePoints = 1;
                        break;
                    default:
                        gradePoints = 0;
                        break;
                }

                totalGradePoints += gradePoints * credits;
                totalCredits += credits;
            }

            return totalGradePoints / totalCredits;
        }
}

// Course class
class Course : ICourseService
{
    private Dictionary<Student, (int credits, char grade)> _studentGrades = new
Dictionary<Student, (int, char)>();

    public Dictionary<Student, (int credits, char grade)> StudentGrades
    {
        get { return _studentGrades; }
    }

    public void AddStudentGrade(Student student, int credits, char grade)
    {
        _studentGrades.Add(student, (credits, grade));
    }

    // Method to calculate the total grade points earned by all students in the course
    public decimal CalculateTotalGradePoints()
```

```
{
    decimal totalGradePoints = 0;

    foreach (var gradeInfo in _studentGrades.Values)
    {
        int credits = gradeInfo.credits;
        char grade = gradeInfo.grade;

        decimal gradePoints;
        switch (grade)
        {
            case 'A':
                gradePoints = 4;
                break;
            case 'B':
                gradePoints = 3;
                break;
            case 'C':
                gradePoints = 2;
                break;
            case 'D':
                gradePoints = 1;
                break;
            default:
                gradePoints = 0;
                break;
        }

        totalGradePoints += gradePoints * credits;
    }

    return totalGradePoints;
}

// Method to calculate the total credits earned by all students in the course
public int CalculateTotalCredits()
{
    int totalCredits = 0;

    foreach (var gradeInfo in _studentGrades.Values)
    {
        totalCredits += gradeInfo.credits;
    }
```

```
            return totalCredits;
        }
}

// Department class
class Department : IDepartmentService
{
        private Instructor _departmentHead;
        private decimal _budget;
        private Tuple<DateTime, DateTime> _schoolYear;
        private List<Course> _courses = new List<Course>();

        public Instructor DepartmentHead
        {
            get { return _departmentHead; }
            set { _departmentHead = value; }
        }

        public decimal Budget
        {
            get { return _budget; }
            set { _budget = value; }
        }

        public Tuple<DateTime, DateTime> SchoolYear
        {
            get { return _schoolYear; }
            set { _schoolYear = value; }
        }

        public List<Course> Courses
        {
            get { return _courses; }
        }

        public void Run()
        {
            Console.WriteLine($"The department is directed by
{DepartmentHead.Name}.");
        }
}

class Program
{
```

```
    static void Main(string[] args)
    {
        // Example usage
        Student student = new Student();
        student.Name = “Alice”;
        student.SetBirthDate(new DateTime(2000, 1, 1));
        student.AddAddress(“123 Main St”);
        student.Behave();
        Console.WriteLine($”Age: {student.Age}”);
        Console.WriteLine($”Address: {string.Join(“, “,
student.GetAllAddresses())}”);

        Instructor instructor = new Instructor(new DateTime(2010, 1, 1));
        instructor.Name = “John Walker”;
        instructor.SetBirthDate(new DateTime(1975, 1, 1));
        instructor.Salary = 50000;
        instructor.AddAddress(“456 Elm St”);
        instructor.Behave();
        Console.WriteLine($”Age: {instructor.Age}”);
        Console.WriteLine($”Address: {string.Join(“, “,
instructor.GetAllAddresses())}”);
        Console.WriteLine($”Salary: {instructor.Salary}”);

        Course course = new Course();
        // course.AddStudentGrade(student, 3, ‘A’);
        student.AddCourseGrade(course, 3, ‘A’);
        Console.WriteLine($”Student GPA: {student.GPA}”);

        Department department = new Department();
        department.DepartmentHead = instructor;
        department.Run();
    }
}
```

```
The Alice acts as a Person.
Age: 24
Address: 123 Main St
The John Walker acts as a Person.
Age: 49
Address: 456 Elm St
Salary: 51400
Student GPA: 4
The department is directed by John Walker.

=== Code Execution Successful ===
```

7.

```
using System;

class Color
{
	private int _red;
	private int _green;
	private int _blue;
	private int _alpha;

	// Constructor taking red, green, blue values
	public Color(int red, int green, int blue)
	{
		_red = Clamp(red, 0, 255);
		_green = Clamp(green, 0, 255);
		_blue = Clamp(blue, 0, 255);
		_alpha = 255; // Default alpha value
	}

	// Constructor taking red, green, blue, alpha values
	public Color(int red, int green, int blue, int alpha)
	{
		_red = Clamp(red, 0, 255);
		_green = Clamp(green, 0, 255);
		_blue = Clamp(blue, 0, 255);
		_alpha = Clamp(alpha, 0, 255);
	}

	// Properties
	public int Red
	{
		get { return _red; }
		set { _red = Clamp(value, 0, 255); }
	}

	public int Green
	{
		get { return _green; }
		set { _green = Clamp(value, 0, 255); }
	}

	public int Blue
	{
		get { return _blue; }
		set { _blue = Clamp(value, 0, 255); }
```

```
    }

    public int Alpha
    {
        get { return _alpha; }
        set { _alpha = Clamp(value, 0, 255); }
    }

    // Method to calculate Grayscale
    public double Grayscale()
    {
        return (_red + _green + _blue) / 3.0;
    }

    // Helper method to clamp value between min and max
    private int Clamp(int value, int min, int max)
    {
        return Math.Min(Math.Max(value, min), max);
    }

    // Main function
    static void Main(string[] args)
    {
        Color color1 = new Color(210, 120, 12, 125);
        Console.WriteLine($"Grayscale: {color1.Grayscale()}");
        Console.WriteLine($"Alpha: {color1.Alpha}");

        Color color2 = new Color(210, 120, 12);
        Console.WriteLine($"Grayscale: {color2.Grayscale()}");
        Console.WriteLine($"Alpha: {color2.Alpha}");
    }
}
```

```
Grayscale: 114
Alpha: 125
Grayscale: 114
Alpha: 255

=== Code Execution Successful ===
```

```
using System;

class Ball
{
    private decimal _size;
    private Color _color;
```

```
private int _throwCount;

// Constructor with size only (default color to white)
public Ball(decimal size)
{
	_size = size;
	_color = new Color(255, 255, 255, 255); // Default color to white
	_throwCount = 0;
}

// Constructor with size and color (including alpha)
public Ball(decimal size, int red, int green, int blue, int alpha)
{
	_size = size;
	_color = new Color(red, green, blue, alpha);
	_throwCount = 0;
}

// Constructor with size and color (without alpha)
public Ball(decimal size, int red, int green, int blue)
{
	_size = size;
	_color = new Color(red, green, blue);
	_throwCount = 0;
}

// Pop method
public void Pop()
{
	_size = 0;
}

// Throw method
public void Throw()
{
	if (_size != 0)
	{
		_throwCount++;
	}
}

// Method to get throw count
public int GetThrowCount()
{
```

```
            return _throwCount;
        }
}

class Program
{
        static void Main(string[] args)
        {
            Ball ball1 = new Ball(10); // Create a ball with size only
            Ball ball2 = new Ball(15, 255, 0, 0, 128); // Create a ball with size and color (with alpha)
            Ball ball3 = new Ball(20, 0, 255, 0); // Create a ball with size and color (without alpha)

            // Throw all balls once
            ball1.Throw();
            ball2.Throw();
            ball3.Throw();

            // Pop the first ball
            ball1.Pop();

            // Throw all balls twice
            ball1.Throw();
            ball2.Throw();
            ball3.Throw();
            ball1.Throw();
            ball2.Throw();
            ball3.Throw();

            // Output throw counts
            Console.WriteLine($"Throw count for ball 1: {ball1.GetThrowCount()}");
            Console.WriteLine($"Throw count for ball 2: {ball2.GetThrowCount()}");
            Console.WriteLine($"Throw count for ball 3: {ball3.GetThrowCount()}");
        }
}
```

```
Throw count for ball 1: 1
Throw count for ball 2: 3
Throw count for ball 3: 3

=== Code Execution Successful ===
```