# Object and Class in OOP

**Class**:

- A **class** is a blueprint or template for creating objects.
- It defines the properties (attributes) and behaviors (methods) that the objects created from the class will have.
- For example, a `Car` class might have properties like `color`, `model`, and `speed`, and methods like `accelerate()` and `brake()`.

**Object**:

- An **object** is an instance of a class.
- It is a concrete entity created from the class blueprint, with its own unique state (values for properties).
- For example, if `Car` is a class, then `myCar` (a specific car with `color = "Red"` and `model = "Toyota"`) is an object

**Key Differences**:

| Class | Object |
|---|---|
| A blueprint or template. | An instance of a class. |
| Defines properties and behaviors. | Has actual values for properties. |
| Exists in code (logical entity). | Exists in memory (physical entity). |

-

**Example in C#:**

```csharp
// Class definition
class Car
{
    public string Color;  // Property
    public string Model;  // Property

    public void Accelerate()  // Method
    {
        Console.WriteLine("Car is accelerating!");
    }
}

// Creating an object
Car myCar = new Car();
myCar.Color = "Red";
myCar.Model = "Toyota";
myCar.Accelerate();  // Output: Car is accelerating!
```

## Constructor in OOP

**What is a Constructor?**

- A **constructor** is a special method in a class that is automatically called when an object of that class is created.
- It is used to initialize the object's properties or perform any setup required for the object.

**Key Features of Constructors**:

1. **Name**: The constructor has the same name as the class.
2. **No Return Type**: Unlike regular methods, constructors do not have a return type (not even `void`).
3. **Automatic Execution**: A constructor is automatically called when an object is created using the `new` keyword.
4. **Overloading**: You can have multiple constructors in a class with different parameters (this is called constructor overloading).

**Types of Constructors**:

1. **Default Constructor**:
    a. A constructor with no parameters.
    b. If you don't define any constructor, the compiler automatically provides a default constructor.
2. **Parameterized Constructor**:
    a. A constructor that takes parameters to initialize the object with specific values.
3. **Copy Constructor**:
    a. A constructor that creates an object by copying values from another object of the same class.

**Example in C#:**

```
class Car
{
    public string Color;  // Property
    public string Model;  // Property

    // Default Constructor
    public Car()
    {
        Color = "Unknown";
        Model = "Unknown";
    }

    // Parameterized Constructor
    public Car(string color, string model)
    {
        Color = color;
        Model = model;
    }

    // Method
    public void DisplayInfo()
    {
        Console.WriteLine($"Color: {Color}, Model: {Model}");
    }
}

class Program
{
    static void Main()
    {
        // Using the default constructor
        Car car1 = new Car();
        car1.DisplayInfo();  // Output: Color: Unknown, Model: Unknown

        // Using the parameterized constructor
        Car car2 = new Car("Red", "Toyota");
        car2.DisplayInfo();  // Output: Color: Red, Model: Toyota
    }
}
```

# Access Modifiers in OOP

### *What are Access Modifiers?*

- **Access modifiers** are keywords used to control the visibility and accessibility of classes, methods, properties, and other members in object-oriented programming.
- They define the scope of where a member (class, method, property, etc.) can be accessed.

### *Common Access Modifiers:*

1. **`public`**:
    a. The member is accessible from **anywhere**.
    b. Example: A public method can be called from any other class.
2. **`private`**:
    a. The member is accessible **only within the same class**.
    b. Example: A private field can only be accessed or modified by methods within the same class.
3. **`protected`**:
    a. The member is accessible **within the same class and derived (child) classes**.
    b. Example: A protected method can be accessed by the class itself and any subclass that inherits from it.
4. **`internal`**:
    a. The member is accessible **only within the same assembly (project)**.
    b. Example: An internal class can be accessed by any code in the same project but not from another project.
5. **`protected internal`**:
    a. The member is accessible **within the same assembly and also by derived classes in other assemblies**.
    b. Combines the behavior of `protected` and `internal`.
6. **`private protected`** (C# 7.2+):
    a. The member is accessible **only within the same class or derived classes in the same assembly**.
    b. Combines the behavior of `private` and `protected`.

c.

```
class Example
{
    public int PublicField = 10;         // Accessible anywhere
    private int PrivateField = 20;       // Accessible only within this class
    protected int ProtectedField = 30;   // Accessible within this class and derived class
    internal int InternalField = 40;     // Accessible within the same assembly
    protected internal int ProtIntField = 50; // Accessible within the same assembly and d
classes
    private protected int PrivProtField = 60; // Accessible within this class and derived
s in the same assembly
}

class DerivedClass : Example
{
    public void Display()
    {
        Console.WriteLine(PublicField);       // Allowed (public)
        // Console.WriteLine(PrivateField);   // Not allowed (private)
        Console.WriteLine(ProtectedField);    // Allowed (protected)
        Console.WriteLine(InternalField);     // Allowed (internal)
        Console.WriteLine(ProtIntField);      // Allowed (protected internal)
        Console.WriteLine(PrivProtField);     // Allowed (private protected)
    }
}

class Program
{
    static void Main()
    {
        Example obj = new Example();
        Console.WriteLine(obj.PublicField);   // Allowed (public)
        // Console.WriteLine(obj.PrivateField); // Not allowed (private)
        // Console.WriteLine(obj.ProtectedField); // Not allowed (protected)
        Console.WriteLine(obj.InternalField); // Allowed (internal)
        Console.WriteLine(obj.ProtIntField);  // Allowed (protected internal)
        // Console.WriteLine(obj.PrivProtField); // Not allowed (private protected)
    }
}
```

# Properties and Encapsulation in OOP

**What are Properties?**

- **Properties** are members of a class that provide a flexible way to read, write, or compute the values of private fields.
- They act as a bridge between the outside world and the internal state of an object.
- Properties typically have a **getter** (to read the value) and a **setter** (to write the value).

- **What is Encapsulation?**
- **Encapsulation** is one of the four fundamental principles of OOP (along with inheritance, polymorphism, and abstraction).
- It refers to the bundling of data (fields) and methods (functions) that operate on the data into a single unit (a class).
- Encapsulation also involves restricting direct access to some of an object's components, which is achieved using **access modifiers** (like `private`) and **properties**.

- **How Properties and Encapsulation Work Together**
- **Private Fields**: Data is stored in private fields to prevent direct access from outside the class.
- **Public Properties**: Properties provide controlled access to these private fields using `get` and `set` accessors.
- **Validation**: Properties allow you to add logic (e.g., validation) when getting or setting values

```
class Person
{
    // Private field (encapsulated)
    private string _name;
    private int _age;

    // Public property for _name
    public string Name
    {
        get { return _name; }  // Getter
        set { _name = value; } // Setter
    }

    // Public property for _age with validation
    public int Age
    {
        get { return _age; }
        set
        {
            if (value > 0)  // Validation
                _age = value;
            else
                Console.WriteLine("Age must be greater than 0.");
        }
    }
}

class Program
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "John";  // Set the name using the property
        person.Age = 25;       // Set the age using the property

        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");  // Get the values using t
he properties

        person.Age = -5;  // Invalid age (validation in the setter will prevent this)
    }
```

# Inheritance in OOP

**What is Inheritance?**

- **Inheritance** is one of the four fundamental principles of Object-Oriented Programming (OOP).
- It allows a class (called a **child class** or **subclass**) to inherit properties and methods from another class (called a **parent class** or **superclass**).
- Inheritance promotes **code reusability** and establishes a **hierarchical relationship** between classes.

- **Key Concepts of Inheritance**:

- **Parent Class (Base Class)**:
  - The class whose properties and methods are inherited.
  - Example: A `Vehicle` class with properties like `Speed` and methods like `Move()`.
- **Child Class (Derived Class)**:
  - The class that inherits from the parent class.
  - Example: A `Car` class that inherits from `Vehicle` and adds its own properties like `NumberOfDoors`.
- **Reusability**:
  - The child class can reuse the code from the parent class, reducing duplication.
- **Extensibility**:
  - The child class can extend the functionality of the parent class by adding new properties or methods.
- **Method Overriding**:
  - The child class can override methods from the parent class to provide its own implementation

csharp Copy

```csharp
// Parent class (Base class)
class Vehicle
{
    public int Speed { get; set; }

    public void Move()
    {
        Console.WriteLine("The vehicle is moving.");
    }
}

// Child class (Derived class)
class Car : Vehicle  // Inherits from Vehicle
{
    public int NumberOfDoors { get; set; }

    public void DisplayInfo()
    {
        Console.WriteLine($"Speed: {Speed}, Doors: {NumberOfDoors}");
    }
}

class Program
{
    static void Main()
    {
        Car myCar = new Car();
        myCar.Speed = 60;           // Inherited from Vehicle
        myCar.NumberOfDoors = 4;    // Specific to Car
        myCar.Move();               // Inherited from Vehicle
        myCar.DisplayInfo();        // Specific to Car
    }
}
```

**Types of Inheritance**:

1. **Single Inheritance**:
    a. A class inherits from only one parent class.
    b. Example: `Car` inherits from `Vehicle`.
2. **Multilevel Inheritance**:
    a. A class inherits from a parent class, which in turn inherits from another class.
    b. Example: `Car` inherits from `Vehicle`, and `Vehicle` inherits from `Machine`.
3. **Hierarchical Inheritance**:
    a. Multiple classes inherit from a single parent class.
    b. Example: `Car` and `Bike` both inherit from `Vehicle`.
4. **Multiple Inheritance** (Not supported in C#):

a. A class inherits from more than one parent class.
b. Example: `Car` inherits from `Vehicle` and `Machine` (not allowed in C#).

# Polymorphism in OOP

**What is Polymorphism?**

- **Polymorphism** is one of the four fundamental principles of Object-Oriented Programming (OOP).
- The word "polymorphism" comes from Greek, meaning "many forms."
- It allows objects of different classes to be treated as objects of a common parent class, enabling a single interface to represent different underlying forms (data types).

## Key Concepts of Polymorphism:

- **One Interface, Multiple Implementations**:
  - Polymorphism allows methods to behave differently based on the object that invokes them.
  - For example, a `Draw()` method can behave differently for a `Circle` object and a `Square` object.
- **Types of Polymorphism**:
  - **Compile-Time Polymorphism (Static Polymorphism)**:
    - Achieved through **method overloading** (multiple methods with the same name but different parameters).
  - **Run-Time Polymorphism (Dynamic Polymorphism):**

**Method Overriding**:

- The child class can override a method from the parent class to provide its own implementation.
- In C#, this is done using the `virtual` keyword in the parent class and the `override` keyword in the child class.

```
class Vehicle
{
    public virtual void Move()  // Virtual method
    {
        Console.WriteLine("The vehicle is moving.");
    }
}

class Car : Vehicle
{
    public override void Move()  // Override method
    {
        Console.WriteLine("The car is driving.");
    }
}

class Program
{
    static void Main()
    {
        Vehicle myVehicle = new Vehicle();
        myVehicle.Move();  // Output: The vehicle is moving.

        Car myCar = new Car();
        myCar.Move();      // Output: The car is driving.
    }
}
```

**2. Compile-Time Polymorphism (Method Overloading):**

csharp Copy

```csharp
class MathOperations
{
    // Method with two integers
    public int Add(int a, int b)
    {
        return a + b;
    }

    // Method with three integers
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }

    // Method with two doubles
    public double Add(double a, double b)
    {
        return a + b;
    }
}

class Program
{
    static void Main()
    {
        MathOperations math = new MathOperations();

        Console.WriteLine(math.Add(5, 10));          // Output: 15
        Console.WriteLine(math.Add(5, 10, 15));      // Output: 30
        Console.WriteLine(math.Add(5.5, 10.5));      // Output: 16.0
    }
}
```

**Summary**:

- **Polymorphism** allows objects of different classes to be treated as objects of a common parent class.
- It is achieved through **method overriding** (run-time polymorphism) and **method overloading** (compile-time polymorphism).
- Polymorphism promotes **flexibility**, **extensibility**, and **code reusability** in OOP.

Polymorphism is a powerful feature that enables you to write clean, modular, and maintainable code.

# Abstraction in OOP

**What is Abstraction?**

- **Abstraction** is one of the four fundamental principles of Object-Oriented Programming (OOP).
- It focuses on **hiding the internal implementation details** of an object and exposing only the necessary features or functionalities to the outside world.
- Abstraction allows you to work with objects at a higher level, without worrying about how they work internally.

- **Key Concepts of Abstraction**:
- **Hide Complexity**:
    - Abstraction simplifies complex systems by breaking them into smaller, manageable parts.
    - For example, when you drive a car, you don't need to know how the engine works; you only need to know how to use the steering wheel, pedals, and gears.
- **Focus on What, Not How**:
    - Abstraction focuses on **what an object does** rather than **how it does it**.
- **Achieved Through**:
    - **Abstract Classes**: Classes that cannot be instantiated and may contain abstract methods (methods without implementation).
    - **Interfaces**: Contracts that define a set of methods a class must implement.
-

```
// Abstract class
abstract class Animal
{
    // Abstract method (no implementation)
    public abstract void MakeSound();

    // Regular method with implementation
    public void Sleep()
    {
        Console.WriteLine("The animal is sleeping.");
    }
}

// Derived class
class Dog : Animal
{
    // Implementation of the abstract method
    public override void MakeSound()
    {
        Console.WriteLine("The dog barks.");
    }
}

class Program
{
    static void Main()
    {
        // Animal myAnimal = new Animal();  // Error: Cannot instantiate an abstract class
        Animal myDog = new Dog();  // Polymorphism: Using base class reference
        myDog.MakeSound();  // Output: The dog barks.
        myDog.Sleep();      // Output: The animal is sleeping.
    }
}
```

# Struct in OOP

**What is a Struct?**

- A **struct** (short for structure) is a value type in C# that encapsulates a group of related variables.
- It is similar to a **class** but is typically used for lightweight objects that do not require inheritance or reference-type behavior.
  Structs are stored on the **stack** (unless they are part of a class), making them more efficient for small, simple data structure

**Key Features of Structs**:

1. **Value Type**:
   a. Structs are value types, meaning they hold their data directly rather than referencing it (like classes do).
   b. When a struct is assigned to a new variable, a **copy** of the struct is created.
2. **No Inheritance**:
   a. Structs cannot inherit from other structs or classes, and they cannot be used as a base for other structs or classes.
   b. However, they can implement **interfaces**.
3. **Default Constructor**:
   a. Structs automatically have a default constructor that initializes all fields to their default values (e.g., `0` for integers, `false` for booleans).
4. **Immutability**:
   a. Structs are often used for immutable data types, where the values do not change after creation.
5. **Performance**:
   a. Structs are more efficient than classes for small, frequently used data structures because they are allocated on the stack.

```
// Define a struct
struct Point
{
    public int X;  // Field
    public int Y;  // Field

    // Constructor (optional)
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Method
    public void Display()
    {
        Console.WriteLine($"X: {X}, Y: {Y}");
    }
}

class Program
{
    static void Main()
    {
        // Create a struct instance
        Point p1 = new Point(10, 20);
        p1.Display();  // Output: X: 10, Y: 20

        // Create another struct instance (copy of p1)
        Point p2 = p1;
        p2.X = 30;  // Modify the copy
        p2.Display();  // Output: X: 30, Y: 20

        // Original struct remains unchanged
        p1.Display();  // Output: X: 10, Y: 20
    }
}
```

## When to Use a Struct:

1. **Small Data Structures**:
    a. Use structs for small, simple data structures that do not require inheritance or polymorphism.
    b. Example: A `Point` or `Rectangle` in a graphics application.
2. **Immutable Data**:
    a. Use structs for immutable data types where the values do not change after creation.
3. **Performance-Critical Scenarios**:
    a. Use structs when performance is critical, as they are allocated on the stack and avoid the overhead of garbage collection

**Key Differences Between Structs and Classes:**

| Feature | Struct | Class |
|---|---|---|
| **Type** | Value type (stored on the stack). | Reference type (stored on the heap). |
| **Inheritance** | Cannot inherit from other structs/classes. | Supports inheritance. |
| **Default Constructor** | Always has a default constructor. | Can define a custom default constructor. |
| **Performance** | Faster for small, simple data structures. | Slightly slower due to heap allocation. |
| **Copy Behavior** | Copying creates a new instance. | Copying creates a reference to the same object. |
| **Nullability** | Cannot be `null` (unless nullable). | Can be `null`. |

**Limitations of Structs**:

1. **No Inheritance**:
    a. Structs cannot inherit from other structs or classes.
2. **No Custom Default Constructor**:
    a. You cannot define a custom default constructor for a struct.
3. **Size Limitations**:
    a. Structs should be small (typically less than 16 bytes) to avoid performance issues.

# What is an Interface?

An **interface** in C# is a **contract** that defines a set of methods, properties, events, or indexers that a class or struct must implement. It specifies **what** a class should do, but not **how** it should do it. Interfaces are used to achieve **abstraction** and **multiple inheritance** in C#.

**Key Points:**

1. **Abstraction**: Interfaces provide a way to define a blueprint for classes without implementing the actual functionality.
2. **Multiple Inheritance**: A class can implement multiple interfaces, allowing it to inherit behavior from multiple sources.

```
// Define an interface
public interface IAnimal
{
    void MakeSound(); // Method signature
    string Name { get; set; } // Property signature
}

// Implement the interface in a class
public class Dog : IAnimal
{
    public string Name { get; set; }

    public void MakeSound()
    {
        Console.WriteLine($"{Name} says: Woof!");
    }
}

// Usage
IAnimal myDog = new Dog { Name = "Buddy" };
myDog.MakeSound(); // Output: Buddy says: Woof!
```

## Interface vs Abstract Class

| Feature | Interface | Abstract Class |
|---|---|---|
| **Implementation** | No implementation (only signatures). | Can have partial implementation. |
| **Multiple Inheritance** | A class can implement multiple interfaces. | A class can inherit only one abstract class. |
| **Fields** | Cannot contain fields. | Can contain fields. |
| **Constructors** | Cannot have constructors. | Can have constructors. |
| **Access Modifiers** | Members are public by default. | Can have private/protected members. |

### Multiple Inheritance with Interfaces:

In C#, a class can implement **multiple interfaces**, allowing it to inherit behavior from multiple sources. Here's an example:

```
// Define multiple interfaces
public interface IFlyable
{
    void Fly();
}

public interface ISwimmable
{
    void Swim();
}

// Implement multiple interfaces in a class
public class Duck : IFlyable, ISwimmable
{
    public void Fly()
    {
        Console.WriteLine("Duck is flying.");
    }

    public void Swim()
    {
        Console.WriteLine("Duck is swimming.");
    }
}

// Usage
Duck myDuck = new Duck();
myDuck.Fly();  // Output: Duck is flying.
myDuck.Swim(); // Output: Duck is swimming.
```

**Explicit Interface Implementation:**
When a class implements multiple interfaces with the same method signature, you can use **explicit interface implementation** to avoid conflicts. This allows you to specify which interface's method is being implemented.

```
// Define two interfaces with the same method signature
public interface IWriter
{
    void Write();
}

public interface IPrinter
{
    void Write();
}

// Implement both interfaces explicitly
public class Device : IWriter, IPrinter
{
    void IWriter.Write()
    {
        Console.WriteLine("Writing to a file.");
    }

    void IPrinter.Write()
    {
        Console.WriteLine("Printing to paper.");
    }
}

// Usage
Device myDevice = new Device();

((IWriter)myDevice).Write(); // Output: Writing to a file.
((IPrinter)myDevice).Write(); // Output: Printing to paper.
```

# Association, Aggregation, and Composition

### *Association*

- **Definition**: A relationship between two or more classes where objects of one class are connected to objects of another class. It can be a one-to-one, one-to-many, or many-to-many relationship.
- **Characteristics**:
  - Objects can exist independently.
  - No ownership is implied.
  -
- **Example**: A `Teacher` and a `Student` are associated because a teacher teaches students, but both can exist independently.

```
public class Teacher
{
    public string Name { get; set; }
}

public class Student
{
    public string Name { get; set; }
    public Teacher Teacher { get; set; } // Association
}
```

### *Aggregation*

- **Definition**: A specialized form of association where one class represents a "whole" and the other represents a "part." The "part" can exist independently of the "whole."
- **Characteristics**:
  - Represents a "has-a" relationship.
  - The "part" can belong to multiple "wholes."
  - No strong lifecycle dependency.
- **Example**: A `Department` has `Professors`, but professors can exist even if the department is deleted.

```
public class Professor
{
    public string Name { get; set; }
}

public class Department
{
    public string Name { get; set; }
    public List<Professor> Professors { get; set; } // Aggregation

    public Department()
    {
        Professors = new List<Professor>();
    }
}
```

### *Composition*

- **Definition**: A stronger form of aggregation where the "part" cannot exist without the "whole." The lifecycle of the "part" is tightly coupled with the "whole."
- **Characteristics**:
    - Represents a "contains-a" relationship.
    - The "part" cannot exist independently.
    - If the "whole" is destroyed, the "part" is also destroyed.
- **Example**: A `Car` has an `Engine`. The engine cannot exist without the car.

```
public class Engine
{
    public string Type { get; set; }
}

public class Car
{
    public string Model { get; set; }
    public Engine CarEngine { get; set; } // Composition

    public Car()
    {
        CarEngine = new Engine(); // Engine is created with the car
    }
}
```

## Enum in OOP

**What is an Enum?**

- An **enum** (short for enumeration) is a special data type in C# that allows you to define a set of named constants.
- It is used to represent a fixed set of values, making the code more readable and maintainable.
- Enums are value types and are stored on the stack
-
- **Key Features of Enums**:
- **Named Constants**:
    - Enums provide a way to assign meaningful names to integral values.
    - Example: Instead of using `0`, `1`, and `2` to represent days of the week, you can use `Day.Monday`, `Day.Tuesday`, etc.
- **Type Safety**:
    - Enums ensure type safety by restricting the values that can be assigned to a variable.
- **Underlying Type**:
    - By default, the underlying type of an enum is `int`, but you can specify other integral types (e.g., `byte`, `short`, `long`).
- **Immutability**:
    - Enum values are constants and cannot be changed at runtime.
- **Useful for Switch Statements**:
    -
-

### Example of an Enum in C#:

csharp Copy

```csharp
// Define an enum
enum Day
{
    Monday,    // 0 (default)
    Tuesday,   // 1
    Wednesday, // 2
    Thursday,  // 3
    Friday,    // 4
    Saturday,  // 5
    Sunday     // 6
}

class Program
{
    static void Main()
    {
        // Assign an enum value
        Day today = Day.Wednesday;

        // Use the enum in a switch statement
        switch (today)
        {
            case Day.Monday:
                Console.WriteLine("It's Monday!");
                break;
            case Day.Tuesday:
                Console.WriteLine("It's Tuesday!");
                break;
            case Day.Wednesday:
                Console.WriteLine("It's Wednesday!");
                break;
            default:
                Console.WriteLine("It's some other day.");
                break;
        }

        // Get the underlying integer value
        int dayValue = (int)today;
        Console.WriteLine($"The value of {today} is {dayValue}.");  // Output: The value of Wedn
esday is 2.
    }
}
```

# Static Types in OOP

**What are Static Types?**

- In Object-Oriented Programming (OOP), **static types** refer to members (fields, properties, methods, or classes) that belong to the type itself rather than to a specific instance of the type.
- Static members are shared across all instances of a class and can be accessed directly using the class name, without creating an object.

**Key Features of Static Types:**

1. **Shared Across Instances**:
    a. Static members are shared by all instances of a class. Changing a static member affects all instances.
2. **No Instance Required**:
    a. Static members can be accessed directly using the class name, without creating an instance of the class.
3. **Lifetime**:
    a. Static members are created when the program starts and remain in memory until the program ends.
4. **Common Use Cases**:
    a. Utility methods (e.g., `Math.Sqrt`).
    b. Constants (e.g., `Math.PI`).
    c. Singleton design pattern.
    d. Counters or shared state.

**Static Fields and Methods:**

csharp Copy

```csharp
class Counter
{
    // Static field (shared across all instances)
    public static int Count = 0;

    // Static method
    public static void Increment()
    {
        Count++;
    }

    // Instance method
    public void DisplayCount()
    {
        Console.WriteLine($"Count: {Count}");
    }
}

class Program
{
    static void Main()
    {
        // Access static field and method directly using the class name
        Counter.Increment();
        Counter.Increment();

        Console.WriteLine($"Static Count: {Counter.Count}");  // Output: Static Count: 2

        // Create instances of Counter
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        c1.DisplayCount();  // Output: Count: 2
        c2.DisplayCount();  // Output: Count: 2

        // Increment using static method
        Counter.Increment();
        c1.DisplayCount();  // Output: Count: 3
    }
}
```

**Key Benefits of Static Types**:

1. **Memory Efficiency**:
    a. Static members are allocated once and shared across all instances, reducing memory usage.
2. **Global Access**:
    a. Static members can be accessed globally without creating an instance of the class.
3. **Utility Functions**:
    a. Static classes and methods are ideal for utility functions that do not depend on instance-specific data.
4. **Singleton Pattern**:
    a. Static members are often used to implement the Singleton design pattern, ensuring only one instance of a class exists

. **Limitations of Static Types**:

5. **No Instance-Specific Data**:
    a. Static members cannot access instance-specific data or methods.
6. **Global State**:
    a. Overuse of static members can lead to global state, making the code harder to test and maintain.
7. **Thread Safety**:
    a. Static members can cause thread-safety issues in multi-threaded applications if not properly synchronized.