Sealed Class

`private`	Member is accessible inside the type only. This is the default.
`internal`	Member is accessible inside the type and any type in the same assembly.
`protected`	Member is accessible inside the type and any type that inherits from the type.
`public`	Member is accessible everywhere.
`internal` `protected`	Member is accessible inside the type, any type in the same assembly, and any type that inherits from the type. Equivalent to a fictional access modifier named `internal_or_protected`.
`private` `protected`	Member is accessible inside the type and any type that inherits from the type and is in the same assembly. Equivalent to a fictional access modifier named `internal_and_protected`. This combination is only available with C# 7.2 or later.

Sealed Class

Useful when you don't want a class to inherit from another.
It is also aplicable at a method level

```
using System;
namespace SealedClass {
  sealed class Animal {

  }

  // trying to inherit sealed class
  // Error Code
  class Dog : Animal {

  }

  class Program  {
    static void Main (string [] args) {

      // create an object of Dog class
      Dog d1 = new Dog();
      Console.ReadLine();
    }
  }
}
```

Declaring sealed class

Compilation error when you try to inherit from a sealed class

error CS0509: 'Dog': cannot derive from sealed type 'Animal'

Inheritance in C#

Interfaces

Interfaces and multiple inheritance

```
interface IPolygon {

  // method without body
  void calculateArea();
}
```

Interface declaration

Interface firm, does not allow Access modifiers

- Declaration needs the use of the keyword Interface.
- Commonly the name of the Interface starts with the letter I.
- Members of an interface don't have implementation. In the latest versions of C# you can have default implementation.
- Interface can contain the following members: properties, methods, events, indexers.
- Every member is public by default

```
interface IPolygon {
  // method without body
  void calculateArea(int a, int b);

}

interface IColor {

  void getColor();
}
```

Interfaces declaration

```
// implements two interface
class Rectangle : IPolygon, IColor {

  // implementation of IPolygon interface
  public void calculateArea(int a, int b) {

    int area = a * b;
    Console.WriteLine("Area of Rectangle: " + area);
  }

  // implementation of IColor interface
  public void getColor() {

    Console.WriteLine("Red Rectangle");

  }
}

class Program {
  static void Main (string [] args) {

    Rectangle r1 = new Rectangle();

    r1.calculateArea(100, 200);
    r1.getColor();
  }
}
```

Interfaces implementation

Implementation of the interface IPolygon

Implementation of the interface IColor

Result

```
Area of Rectangle: 20000
Red Rectangle
```

Interfaces in C#

Interfaces

Describe a group of related functions that can belong to any class or struct

```
public interface IPersonRepository
{
    void SavePerson(Person person);
    Person GetPerson(int Id);
}

public class ServiceRepository : IPersonRepository
{
    public void SavePerson(Person person)
    {
        // Logic to save record to Memory
    }

    public Person GetPerson(int Id)
    {
        // Logic to retrieve the Person according the id
        return new Person("Aaron B", 22, 1);
    }
}
```

Creating an interface and its implementation

```
IPersonRepository GetRepository(string repositoryType)
{
    IPersonRepository repository = null;

    switch(repositoryType)
    {
        case "Service": repository = new
ServiceRepository();
        break;
        case "CSV": repository = new CSVRepository();
        break;
        case "SQL": repository = new SQLRepository();
        break;
    }

    return repository;
}
```

Using interface as a return type

Interfaces pitfalls

Interfaces can be easily misused. Some of the common interface pitfalls are the following.

```
public interface ISaveable {
  void Save();
  void Save(string message); // Added Member
}
```

```
public class Catalog : ISaveable
{
  public void Save()
  {
    Console.Write("Saved (catalog)");
  }
}
*** ERROR Save(string) is missing ***
```

Adding members breaks implementers

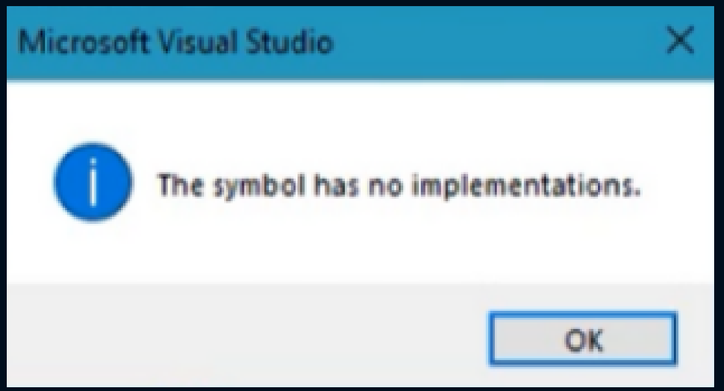
```
public interface ISaveable {
  void Save();
  // void Save(string message) REMOVED
}
```

```
public class InventoryItem
{

  ISaveable saver = new SQLSaver();
  saver.Save("Added inventory"); *** ERROR ***

}
```

Removing members breaks implementers



Hard to debug (go to implementation)

Default and explicit implementation

```
public interface ILogger
{
    void Log(LogLevel level, string message);
    void Log(Exception ex) => Log(LogLevel.Error, ex.ToString()); //*
}

class ConsoleLogger : ILogger
{
    public void Log(LogLevel level, string message) { ... }
    //Log(Exception ex) uses the default implementation
}
```

Default implementation

Useful in the following scenarios:
Backward compatibility
Optional functionality
Code Reuse
Progressive enrichment

```
public interface ISaveable {
  void Save();
}

public class Catalog : ISaveable
{
  void ISaveable.Save()
  {
    Console.Write("Saved");
  }
}
```

```
Catalog catalog = new Catalog();

catalog.Save();
*** COMPILER ERROR ***


ISaveable saveable = catalog;

saveable.Save();
// "Saved"

(ISaveable(catalog)).Save();
// "Saved"
```

Explicit implementation

Useful in the following scenarios:
Interface name conflicts
Hiding members from its public interface

Abstract class

Class that cannot be instantiated directly but serves as a blueprint. They can contain a mixture of abstract and non-abstract (concrete) members.

```
public abstract class Animal
{
    public string Name { get; set; }

    public abstract void MakeSound();

    public abstract string Species { get; }
}
```

Declaring an abstract class

It is intended for use as a base class

```
public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }

    public override string Species => "Canine";
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow!");
    }

    public override string Species => "Feline";
}
```

Derived classes

abstract vs virtual

Abstract	Virtual
Member signature as placeholder with no implementation	Is a method with a default implementation
Only use in abstract classes	Use in abstract or concrete classes
Must be overridden by derived class	Optionally overridden by derived class

```
public abstract bool Validate();
```

```
public virtual bool Validate()
{
    ...
}
```

Differences between interfaces and abstract

Abstract classes	Interfaces
May have implementation code Single inheritance Access modifiers on members Can have fields with state and maintain internal state It can use the following members; • Properties • Methods • Events • Indexers • Fields • Constructors • Finalizers (destructors)	No implementation code *(on latest versions you can provide a default implementation) Multiple inheritance (implementation) Members are automatically public Cannot maintain internal state It can use the following members: • Properties • Methods • Events • Indexers

Interface polymorphism

Allows objects of different classes to be treated as instances of a common interface.
The interface defines a contract that specifies the methods that implementing classes must provide.
Through interface polymorphism, you can write code that operates on objects without needing to know their specific types, promoting flexibility and extensibility.

The appropriate implementation of the interface's methods is determined at runtime based on the actual type of the object.
Allow us to work with otherwise unrelated classes in a generalized and reusable way.

```
public interface ILogger
{
    void Log(string message);
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        // Log the message to a file
        Console.WriteLine($"Logging to file: {message}");
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        // Log the message to the console
        Console.WriteLine($"Logging to console: {message}");
    }
}

public class DataProcessor
{
    private readonly ILogger _logger;

    public DataProcessor(ILogger logger)
    {
        _logger = logger;
    }

    public void ProcessData(string data)
    {
        // Process the data

        // Log a message using the injected logger
        _logger.Log("Data processing complete");
    }
}
```

Example

Task

Exercise

Vehicle Management System:

Create a console application that manages different types of vehicles.

1. Create an abstraction that helps the user to handle the functionality of a vehicle
    a. Some common properties a vehicle can have is Make, Model, etc
2. Some examples of a vehicle is a car, a motorcycle, a helicopter, etc.
3. The Console application allows the user to perform actions with these vehicles like Starting or Stopping.