# Object-oriented programming (OOP) Lecture 5:Interface & Abstract, Partial, and Sealed classes

## Interfaces

* A **class** is a blueprint, which means it contains the members and methods that the instantiated objects will have.
* An **interface** can also be categorized as a blueprint, but unlike a class, an interface **doesn't have any method implementation**.
* Interfaces are more like a guideline for classes that implement the interface.
* The main features of interfaces in C# are as follows:

• **Interfaces can't have a method body**; they can only have the method signature.

• **Interfaces can have methods**, **properties**, **events**, and **indexes**.

• **An interface can't be instantiated**, so **no object of an interface can be created**.

• **One class can extend multiple interfaces**.(Multiple inheritance concept)

## When to use an interface?

* This is a very important question for the new developer. For example we have a core banking system. We all know that bank data is very sensitive. A little negligence is risky. So if the bank decided to develop a mobile banking web application from a third party developer then we can't provide full access to our main core banking application. So we must design the interface in our core banking application. The third-party developers only access the core banking application with the limited rights that we have given them in the interface. So the interface is very useful in such conditions.

## Example of an interface:

using System;

namespace BankAccont

{

interface IBankAccount

{

void Debit(double amount);

void Credit(double amount);

}

class BankAccount : IBankAccount

{

public void Debit(double amount)

{

Console.WriteLine($"${amount} has been debited from your account!");

}

public void Credit(double amount)

{

Console.WriteLine($"${amount} has been credited to your account!");

}

static void Main(string[] args)

{

BankAccount account = new BankAccount();

account.Credit(1000);

account.Debit(100);

}

}

}

In the above example code:

* In the preceding example, we can see that we have one interface, called IBankAccount, that has two members: Debit and Credit. Both of these methods have no implementations in the interface. In the interface, the method signatures are more like guidelines or requirements for the classes that will implement this interface. If any class implements this interface, then the class has to implement the method body.
* If the class doesn't implement any of the methods of the interface, the compiler will throw an error that the class has not implemented all the methods of the interface. By language design, if an interface is implemented by a class, all the members of the interface must be taken care of in the class.

## The abstract class

* An **abstract class** is a special kind of class that comes with the C# programming language. This class has **similar functionalities to an interface**. For example, an abstract class can have methods without implementation and with implementation. Consequently, when a class implements an abstract class, the class has to **override** the abstract methods of the abstract class.
* One of the main characteristics of an abstract class is that **it can't be instantiated**. An abstract class can only be used for inheritance.
* It might or might not have abstract methods and accessor.
* **Sealed and abstract modifiers** **can't** be placed in the same class, as they have completely separate meanings.

Let's take a look at an example of an abstract class:

using System;

namespace AbstractClass

{

abstract class Animal

{

public string name;

public int ageInMonths;

public abstract void Move();

public void Eat()

{

Console.WriteLine("Eating");

}

}

class Dog : Animal

{

public override void Move()

{

Console.WriteLine("Moving");

}

}

public class Program

{

static void Main(string[] args)

{

Dog dog = new Dog();

dog.Eat();

}

}

}

In the above example code

* In the preceding example, we saw that the Dog class is implementing the Animal class, and as the Animal class has an **abstract method** called **Move()**, the Dog class must override it.
* If we try to instantiate the abstract class, the compiler will throw an error.

## Difference between Abstract Class and Interface in C#

|  |  |  |
| --- | --- | --- |
| **Category** | **Abstract Class** | **Interface** |
| What is it? | Abstract doesn’t provide full abstraction. It contains non-abstract methods also | Interface provides full abstraction. It contains only method signature without body. |
| How to declare/create? | Using abstract modifier. | Using interfaces modifier. |
| Can it have fields? | Abstract class can have fields. | Interfaces can’t have fields. |
| Implementations of its members/methods? | Abstract classes can have implementations for some of their members (methods). | Interface can't have implementation for any of its members. |
| Access modifiers? | Abstract class members can have access modifiers. | Interface members can’t have access modifiers. |
| Default access modifiers? | Abstract class members can be private by default which can be changed. | Interface members can be public by default which can not be changed. |
| Inherit from another Abstract Class or Interface? | Abstract class can inherit from another abstract class or another interface. | Interface can inherit from another interface only and cannot inherit from an abstract class. |
| Multiple inheritance | Multiple inheritance is not achieved by abstract class. | Multiple inheritance can be achieved by interfaces. |

## The partial class

* You can **split a class**, a struct, or an interface into smaller portions that can be placed in different code files. If you want to do this, you have to use the keyword **partial**.
* Even though the code is in separate code files, when complied, they will be treated as one class altogether.
* **There are many benefits of partial classes**. One benefit is that different developers can work on different code files at a time. Another benefit is that if you are using autogenerated code and you want to extend some functionality of that autogenerated code, you can use a partial class in a separate file. Consequently, you are not directly touching the autogenerated code, but adding new functionality in the class.
* **The partial class has a few requirements**:

1. One of which is that all classes must have the keyword partial in their signatures.
2. **All the partial classes also have to have the same name**, but the file names can be different.
3. The partial classes also **have to have the same accessibility**, such as public, internal, and so on.

* The following is an example of a partial class:

// File name: Animal.cs

using System;

namespace AnimalProject

{

public partial class Animal

{

public string name;

public int ageInMonths;

public void Eat()

{

Console.WriteLine("Eating");

}

}

}

// File name: AnimalMoving.cs

using System;

namespace AnimalProject

{

public partial class Animal

{

public void Move()

{

Console.WriteLine("Moving");

}

}

}

As shown in the preceding code, you can create many partial classes of a class. This will increase the readability of your code, and your code organization will be more structured.

## The sealed class

* One of the principles of OOP is inheritance, but sometimes you may need to **restrict inheritance** in your code for the sake of your application's architecture.
* C# provides a keyword called **sealed**. If this keyword is placed before a class's signature, the class is considered a sealed class.
* If a class is sealed, that particular class **can't be inherited by other classes**. If any class tries to inherit a sealed class, the compiler will throw an error.
* Structs can also be sealed, and in that case, no class can inherit that struct.

Let's look at an example of a sealed class:

sealed class Animal

{

public string name;

public int ageInMonths;

public void Move()

{

Console.WriteLine("Moving");

}

public void Eat()

{

Console.WriteLine("Eating");

}

}

public class Program

{

public static void Main()

{

Animal dog = new Animal();

dog.name = "HO";

dog.ageInMonths = 1;

dog.Move();

dog.Eat();

}

}

In the above example code:

* In the preceding example, we can see how we can create a sealed class. Just using the sealed keyword before the class keyword makes the class a sealed class.
* In the preceding example, we created an Animal sealed class, and in the main method, we instantiated the class and used it. This is now working fine. However, if we try to create a Dog class that will inherit the Animal class, as in the following code, then the compiler will throw an error, saying that the sealed Animal class can't be inherited:

class Dog : Animal

{

public char gender;

}

**Note**

* Even if a sealed class can’t be inherited we can still consume a sealed class and its members by any other class by creating its instance.