Abstract class, Interface

Lesson Objectives

- Define and describe abstract classes
- Explain interfaces
- Compare abstract classes and interfaces
- C# common interfaces

Section 1

ABSTRACT CLASSES

Abstract Classes

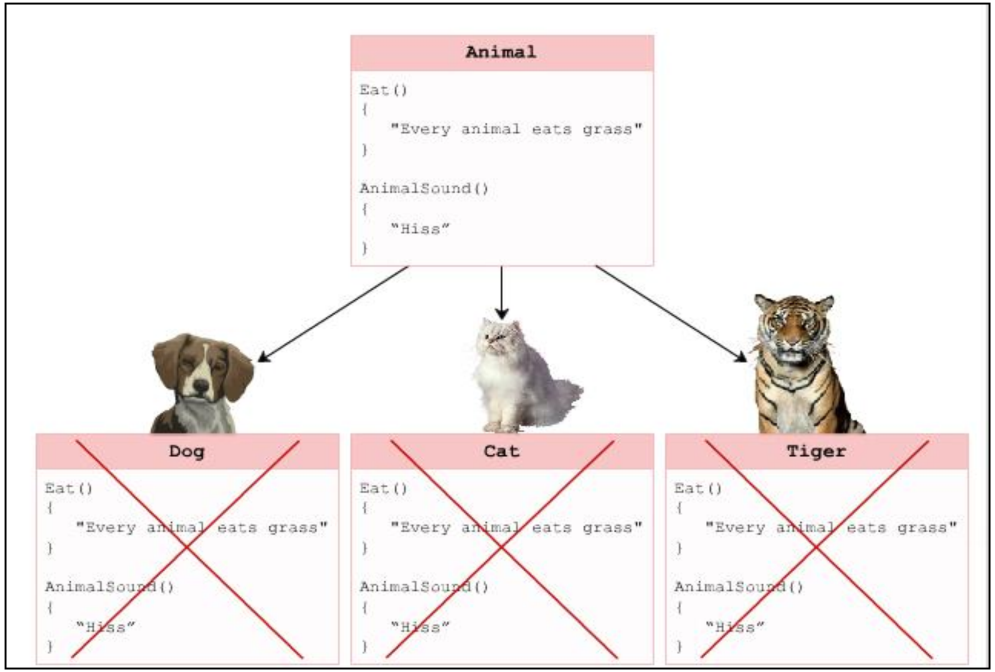
- C# allows designing a class specifically to be used as a base class by declaring it an abstract class.
- Such class can be referred to as an incomplete base class, as it cannot be instantiated, but it can only be implemented or derived.
- An abstract class is declared using the abstract keyword which may or may not contain one or more of the following:
  - ✓ normal data member(s)
  - ✓ normal method(s)
  - ✓ abstract method(s)

Purpose 1-2

- C# allows designing a class specifically to be used as a base class by declaring it an abstract class.
- Such class can be referred to as an incomplete base class, as it cannot be instantiated, but it can only be implemented or derived.
- An abstract class is declared using the abstract keyword which may or may not contain one or more of the following:
  - ✓ normal data member(s)
  - ✓ normal method(s)
  - ✓ abstract method(s)

Purpose 2-2

- The following figure displays an example of abstract class and subclasses:



Definition 1-4

- An abstract class can implement methods that are similar for all the subclasses.

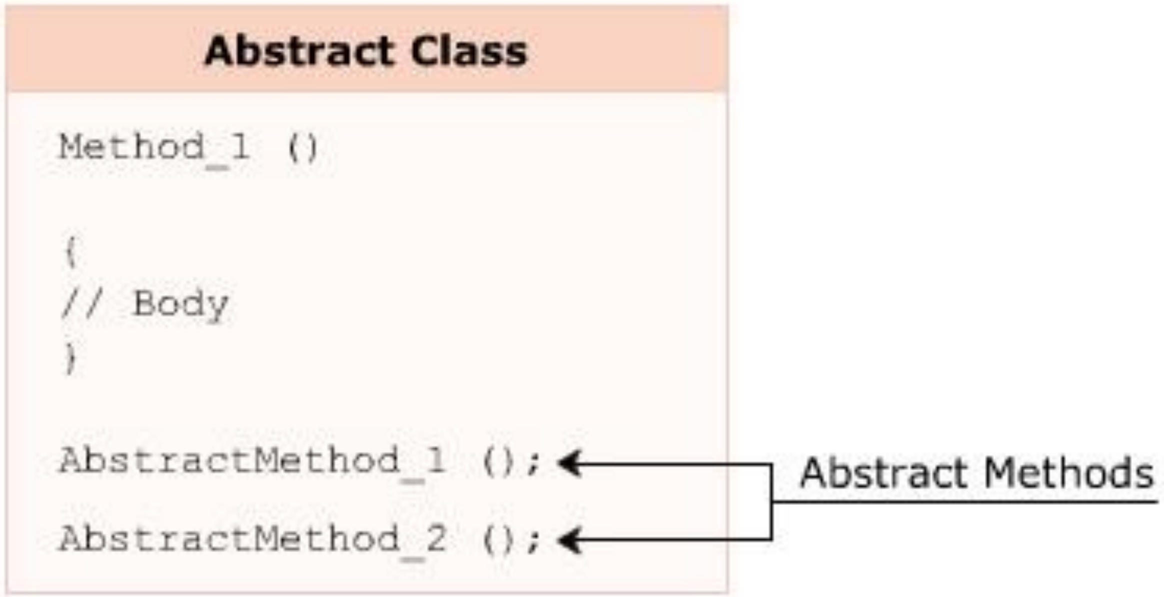
A class that is defined using the abstract keyword and that contains at least one method which is not implemented in the class itself is referred to as an abstract class.

In the absence of the abstract keyword, the class will not be compiled.

Since the abstract class contains at least one method without a body, the class cannot be instantiated using the `new` keyword.

Definition 2-4

- The following figure displays the contents of an abstract class:



```
Method_1 ()

{
// Body
}

AbstractMethod_1 ();
AbstractMethod_2 ();
```

Definition 3-4

- The following syntax is used for declaring an abstract class:

Syntax

```
public abstract class <ClassName>
{
    <access_modifier> abstract <return_type>
        <MethodName>(argument_list);
}
```

where,

- ✓ `abstract:` Specifies that the declared class is abstract.
- ✓ `ClassName:` Specifies the name of the class.

Definition 4-4

- The following code declares an abstract class **`Animal`**:

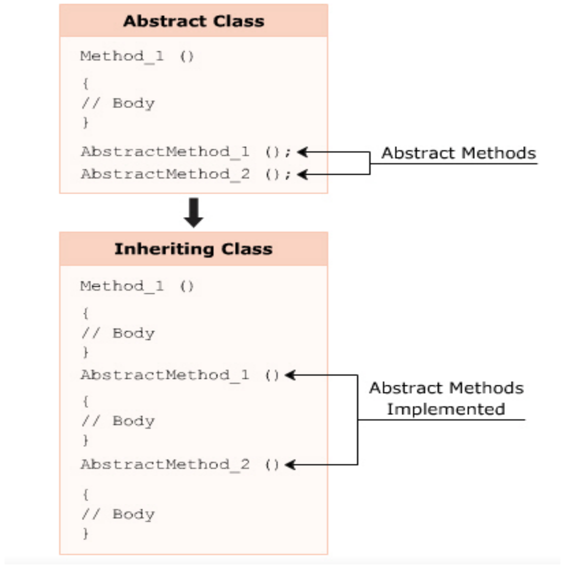
Snippet

```
public abstract class Animal
{
   //Non-abstract method implementation public void Eat()
   {
      Console.WriteLine("Every animal eats food in order to
                                       survive");
   }
   //Abstract method declaration
     public abstract void AnimalSound();
     public abstract void Habitat();
 }
```

- In the code:
  - ✓ The abstract class **`Animal`** is created using the `abstract` keyword.
  - ✓ The **`Animal`** class implements the non-abstract method, **`Eat()`**, as well as declares two abstract methods, **`AnimalSound()`** and **`Habitat()`**.

Implementation 1-4

- The subclass inheriting the abstract class has to override and implement the abstract methods with the same name and arguments.
- On failing to implement, the subclass cannot be instantiated as the C# compiler considers it as abstract.
- The following figure displays an example of inheriting an abstract class:



Implementation 2-4

- The following syntax is used to implement an abstract class:

Syntax

```
class <ClassName> : <AbstractClassName>
{
// class members;
}
```

where,

`AbstractClassName`: Specifies the name of the inherited abstract class.

Implementation 3-4

- The following code declares and implements an abstract class:

Snippet

```
abstract class Animal
{
            public void Eat()
         {
                Console.WriteLine(“Every animal eats food in order to survive”);
          }

          public abstract void AnimalSound();
}
class Lion : Animal
{
       public override void AnimalSound()
      {
                Console.WriteLine(“Lion roars”);
          }
   static void Main(string[] args)
 {
              Lion objLion = new Lion();
              objLion.AnimalSound();
              objLion.Eat();
     }
}
```

Implementation 4-4

Output

```
Lion roars
Every animal eats food in order to survive
```

- In the code:
  - ✓ The abstract class **`Animal`** is declared, and the class **`Lion`** inherits the abstract class **`Animal`**.
  - ✓ Since the **`Animal`** class declares an abstract method called **`AnimalSound()`**, the **`Lion`** class overrides the method **`AnimalSound()`** using the `override` keyword and implements it.
  - ✓ The `Main()` method of the **`Lion`** class then invokes the methods **`AnimalSound()`** and **`Eat()`** using the dot(**`.`**) operator.

Abstract Methods 1-2

- The methods in the abstract class that are declared without a body are termed as abstract methods.
- Following are the features of the abstract methods:

These methods are implemented in the inheriting class.

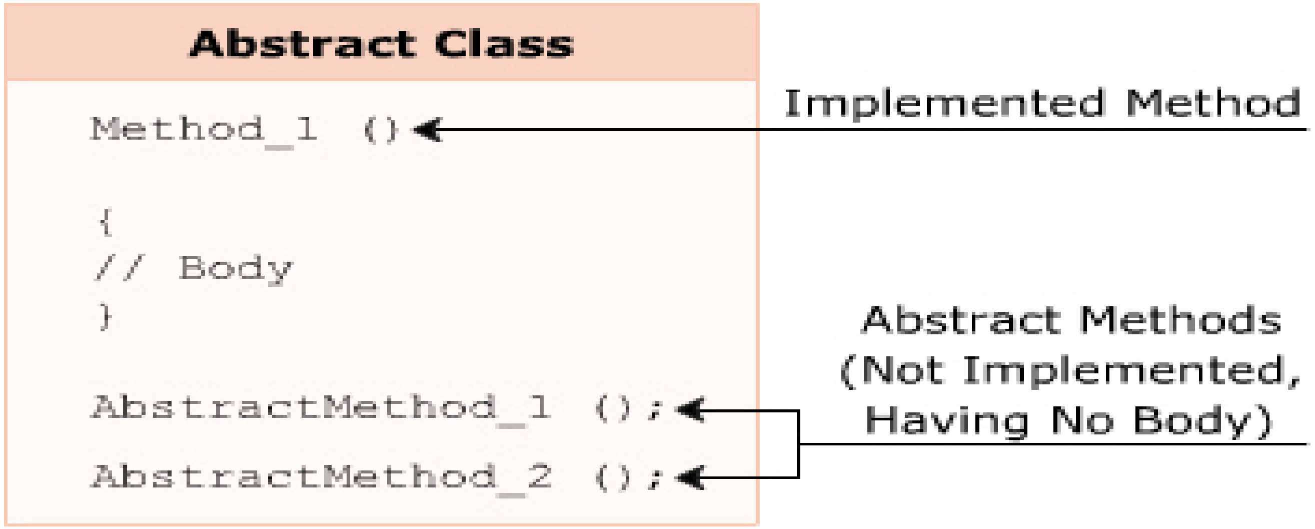
They are declared with an access modifier, a return type, and a signature.

They do not have a body and the method declaration ends with a semicolon.

They provide a common functionality for the classes inheriting the abstract class. The subclasses of the abstract class can override and implement the abstract methods.

Abstract Methods 2-2

- The following figure displays an example of abstract methods:



Abstract Classes - Summary

- What’s an abstract class?
- How to implement an abstract class?
- What’s an abstract method?

Section 2

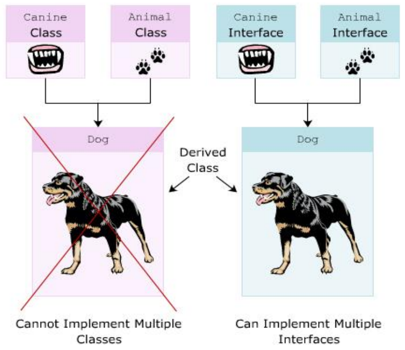
INTERFACES

Interfaces

- A subclass in C# cannot inherit two or more base classes because C# does not support multiple inheritance.
- To overcome this drawback, interfaces were introduced.
- A class in C# can implement multiple interfaces.

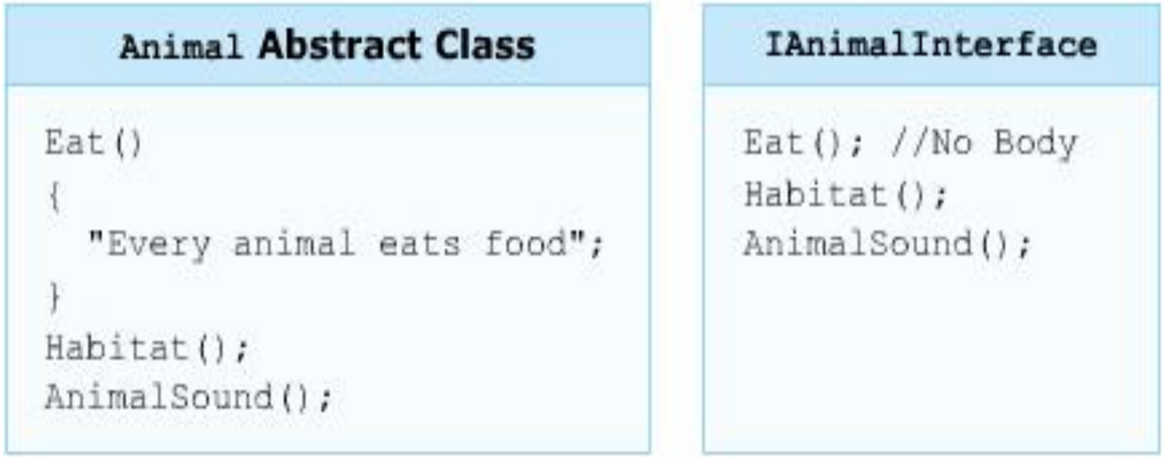
Purpose of Interfaces

- Consider a class **`Dog`** that needs to inherit features of **`Canine`** and **`Animal`** classes.
- The **`Dog`** class cannot inherit methods of both these classes as C# does not support multiple inheritance.
- However, if **`Canine`** and **`Animal`** are declared as interfaces, the class **`Dog`** can implement methods from both the interfaces.
- The following figure displays an example of subclasses with interfaces in C#:



Interfaces 1-2

- An interface contains only abstract members that cannot implement any method.
- An interface cannot be instantiated but can only be inherited by classes or other interfaces.
- An interface is declared using the keyword `interface`.
- In C#, by default, all members declared in an interface have `public` as the access modifier.
- The following figure displays an example of an interface:



```
Eat()
{
  "Every animal eats food";
}
Habitat();
AnimalSound();
```

```
Eat(); //No Body
Habitat();
AnimalSound();
```

Interfaces 2-2

- The following syntax is used to declare an interface:

```
interface <InterfaceName>
{
     //interface members
}
```

where,

- ✓ `interface`: Declares an interface.
- ✓ `InterfaceName`: Is the name of the class.

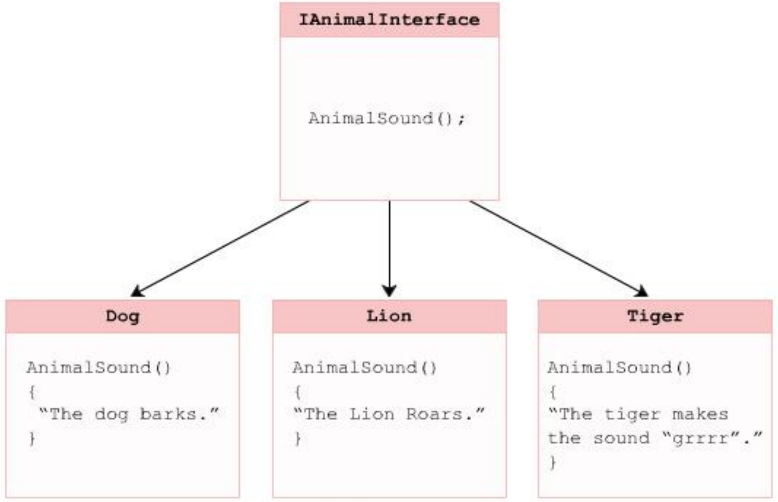
- The following code declares an interface **`IAnimal`**:

```
interface IAnimal
{
     void AnimalType();
}
```

- In the code:
  - ✓ The interface **`IAnimal`** is declared which contains an abstract method **`AnimalType()`**.

Implementing an Interface 1-4

- An interface is implemented by a class in a way similar to inheriting a class.
- When implementing an interface in a class, implement all the abstract methods declared in the interface. If all the methods are not implemented, the class cannot be compiled.
- The methods implemented in the class should be declared with the same name and signature as defined in the interface.
- The following figure displays the implementation of an interface:


IAnimalInterface
AnimalSound();
Dog
AnimalSound() { "The dog barks." }
Lion
AnimalSound() { "The Lion Roars." }
Tiger
AnimalSound() { "The tiger makes the sound "grrrr"." }

Implementing an Interface 2-4

- The following syntax is used to implement an interface:

Syntax

```
class <ClassName> : <InterfaceName>
{
     //Implement the interface methods.
     //Define class members.
}
```

where,

✓ `InterfaceName`: Specifies the name of the interface.

Implementing an Interface 3-4

- The following code declares an interface **`IAnimal`** and implements it in the class **`Dog`**:

Snippet

```
interface IAnimal
{
    void Habitat();
}

class Dog : IAnimal
{
     public void Habitat()
     {
         Console.WriteLine("Can be housed with human beings");
     }
     static void Main(string[] args)
     {

         Dog objDog = new Dog();
         Console.WriteLine(objDog.GetType().Name);
         objDog.Habitat();
     }

}
```

Output

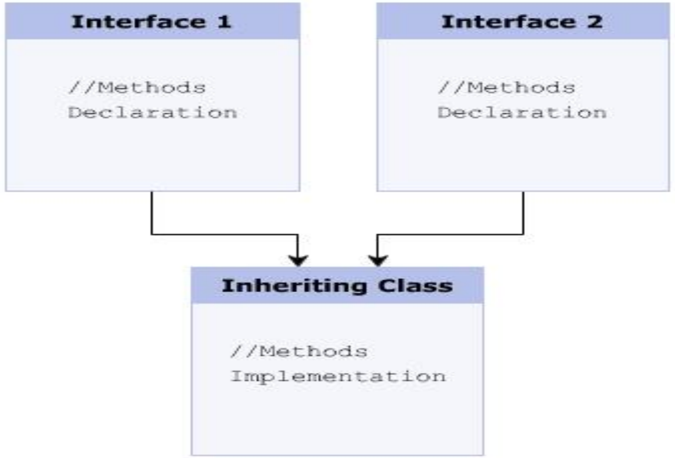
```
Dog
Can be housed with human beings
```

Implementing an Interface 4-4

- In the code:
  - ✓ The code creates an interface **`IAnimal`** that declares the method **`Habitat()`**.
  - ✓ The class **`Dog`** implements the interface **`IAnimal`** and its method **`Habitat()`**.
  - ✓ In the **`Main()`** method of the **`Dog`** class, the class name is displayed using the object and then, the method **`Habitat()`** is invoked using the instance of the **`Dog`** class.

Interfaces and Multiple Inheritance 1-3

- Multiple interfaces can be implemented in a single class which provides the functionality of multiple inheritance.
- You can implement multiple interfaces by placing commas between the interface names while implementing them in a class.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- The `override` keyword is not used while implementing abstract methods of an interface.
- The following figure displays the concept of multiple inheritance using interfaces:



Interfaces and Multiple Inheritance 2-3

- The following syntax is used to implement multiple interfaces:

Syntax

```
class <ClassName> : <Interface1>, <Interface2>
{
     //Implement the interface methods
}
```

- where,
  - ✓ `Interface1`: Specifies the name of the first interface.
  - ✓ `Interface2`: Specifies the name of the second interface.

Interfaces and Multiple Inheritance 3-3

- The following code declares and implements multiple interfaces:

Snippet

```
interface ITerrestrialAnimal
{
    void Eat();
}
interface IMarineAnimal
{
    void Swim();
}
class Crocodile : ITerrestrialAnimal, IMarineAnimal
{
   public void Eat()
   {
      Console.WriteLine("The Crocodile eats flesh");
   }
   public void Swim()
   {
      Console.WriteLine("The Crocodile can swim four times faster than an Olympic 
swimmer");
   }
   static void Main(string[] args)
   {
     Crocodile objCrocodile = new Crocodile();
     objCrocodile.Eat();
     objCrocodile.Swim();
   }

}
```

Output

```
The Crocodile eats flesh
The Crocodile can swim four times faster than an Olympic 
swimmer
```

Explicit Interface Implementation 1-5

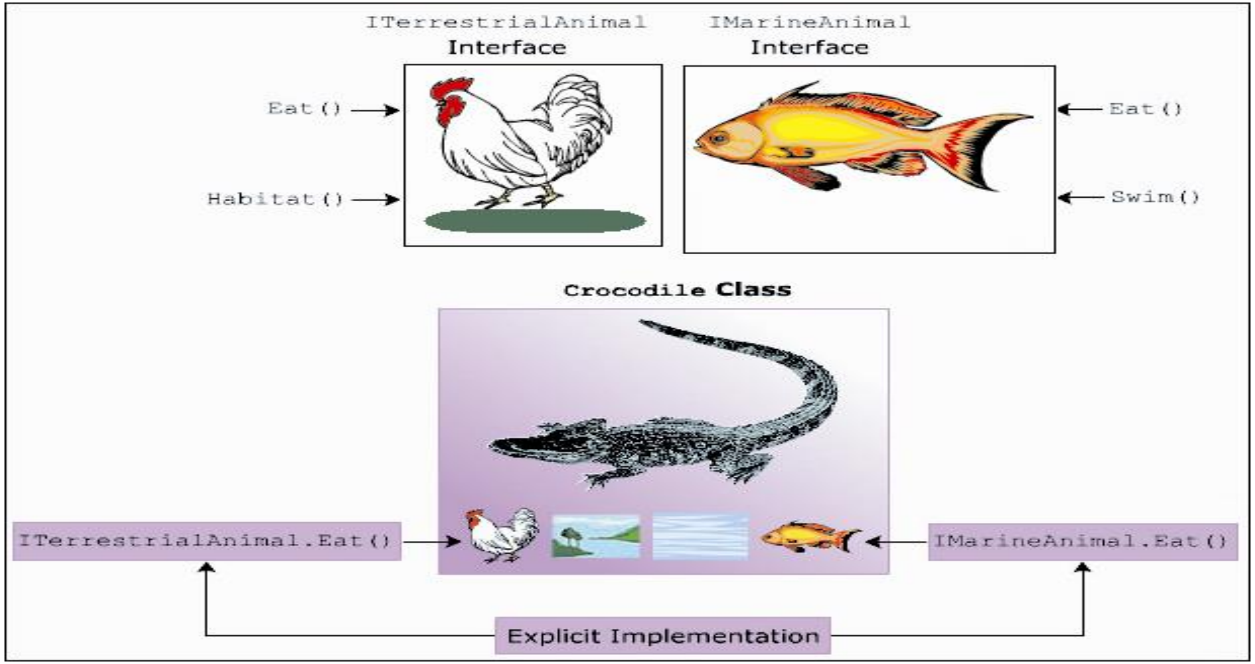
- A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- If an interface has a method name identical to the name of a method declared in the inheriting class, this interface has to be explicitly implemented.

Example

- Consider the interfaces **`ITerrestrialAnimal`** and **`IMarineAnimal`**. The interface **`ITerrestrialAnimal`** declares methods **`Eat()`** and **`Habitat()`**.
- The interface **`IMarineAnimal`** declares methods **`Eat()`** and **`Swim()`**.
- The class **`Crocodile`** implementing the two interfaces has to explicitly implement the method **`Eat()`** from both interfaces by specifying the interface name before the method name.
- While explicitly implementing an interface, you cannot mention modifiers such as `abstract`, `virtual`, `override`, or `new`.

Explicit Interface Implementation 2-5

- The following figure displays the explicit implementation of interfaces:



Explicit Interface Implementation 3-5

- The following syntax is used to explicitly implement interfaces:

Syntax

```
class <ClassName> : <Interface1>, <Interface2>
{
    <access modifier> Interface1.Method();
    {
    //statements;
    }
    <access modifier> Interface2.Method();
    {
    //statements;
    }

}
```

- where,
  - ✓ `Interface1`: Specifies the first interface implemented.
  - ✓ `Interface2`: Specifies the second interface implemented.
  - ✓ `Method()`: Specifies the same method name declared in the two interfaces.

Explicit Interface Implementation 4-5

- The following code demonstrates the use of implementing interfaces explicitly:

Snippet

```
interface ITerrestrialAnimal
{
      string Eat();
}
interface IMarineAnimal
{
      string Eat();
}
class Crocodile : ITerrestrialAnimal, IMarineAnimal
{
      string ITerrestrialAnimal.Eat()
      {
          string terCroc = "Crocodile eats other animals";
          return terCroc;
      }
      string IMarineAnimal.Eat()
      {
           string marCroc = "Crocodile eats fish and marine animals";
           return marCroc;
      }
      public string EatTerrestrial()
      {
             ITerrestrialAnimal objTerAnimal;
             objTerAnimal = this;
             return objTerAnimal.Eat();
       }
       public string EatMarine()
       {
            IMarineAnimal objMarAnimal;
            objMarAnimal = this;
            return objMarAnimal.Eat();
       }
       public static void Main(string[] args)
       {
            Crocodile objCrocodile = new Crocodile();
            string terCroc = objCrocodile.EatTerrestrial();
            Console.WriteLine(terCroc);
            string marCroc = objCrocodile.EatMarine();
            Console.WriteLine(marCroc);
       }
}
```

Explicit Interface Implementation 5-5

Output

```
Crocodile eats other animals
Crocodile eats fish and marine animals
```

- In the code:
  - ✓ The class **`Crocodile`** explicitly implements the method **`Eat()`** of the two interfaces, `ITerrestrialAnimal` and **`IMarineAnimal`**.
  - ✓ The method **`Eat()`** is called by creating a reference of the two interfaces and then calling the method.

Interface Inheritance 1-2

- An interface can inherit multiple interfaces but cannot implement them. The implementation has to be done by a class.
- The following syntax is used to inherit an interface:

Syntax

```
interface <InterfaceName> : <Inherited_InterfaceName>
{
// method declaration;
}
```

- where,
  - ✓ `InterfaceName`: Specifies the name of the interface that inherits another interface.
  - ✓ `Inherited_InterfaceName`: Specifies the name of the inherited interface.

Interface Inheritance 2-2

- The following code declares interfaces that are inherited by other interfaces:

Snippet

```
interface IAnimal
{
     void Drink();
}
interface ICarnivorous
{
    void Eat();
}
interface IReptile : IAnimal, ICarnivorous
{
    void Habitat();
}
class Crocodile : IReptile
{
      public void Drink()
      {
         Console.WriteLine(“Drinks fresh water”);
      }
      public void Habitat()
      {
          Console.WriteLine(“Can stay in Water and Land”);
      }
      public void Eat()
      {
           Console.WriteLine(“Eats Flesh”);
      }
      static void Main(string[] args)
      {
           Crocodile objCrocodile = new Crocodile();
           Console.WriteLine(objCrocodile.GetType().Name);
           objCrocodile.Habitat();
           objCrocodile.Eat();
           objCrocodile.Drink();
      }
}
```

Output

```
Crocodile
Can stay in Water and Land
Eats Flesh
Drinks fresh water
```

Interface- Summary

- Syntax to declare a interface
- Implement a interface
- Inheritance interface

Abstract Classes and Interfaces

- Abstract classes and interfaces both declare methods without implementing them.
- Although both abstract classes and interfaces share similar characteristics, they serve different purposes in a C# application.
- The similarities between abstract classes and interfaces are as follows:

Neither an abstract class nor an interface can be instantiated.

Both, abstract classes as well as interfaces, contain abstract methods.

Abstract methods of both, the abstract class as well as the interface, are implemented by the inheriting subclass.

Both, abstract classes as well as interfaces, can inherit multiple interfaces.

Differences Between an Abstract Class and an Interface

- Abstract classes and interfaces are similar because both contain abstract methods that are implemented by the inheriting class.
- However, there are certain differences between an abstract class and an interface as shown in the following table:

Abstract Classes	Interfaces
An abstract class can inherit a class and multiple interfaces.	An interface can inherit multiple interfaces but cannot inherit a class.
An abstract class can have methods with a body.	An interface cannot have methods with a body.
An abstract class method is implemented using the `override` keyword.	An interface method is implemented without using the `override` keyword.
An abstract class is a better option when you need to implement common methods and declare common abstract methods.	An interface is a better option when you need to declare only abstract methods.
An abstract class can declare constructors and destructors.	An interface cannot declare constructors or destructors.
An abstract class better performance than a interface	

Recommendations for Using Abstract Classes and Interfaces

- An abstract class can inherit another class whereas an interface cannot inherit a class.
- Therefore, abstract classes and interfaces have certain similarities as well as certain differences.
- Following are the guidelines to decide when to use an interface and when to use an abstract class:

Interface	•If a programmer wants to create reusable programs and maintain multiple versions of these programs, it is recommended to create an abstract class. •Abstract classes helps to maintain the version of the programs in a simple manner. •Unlike abstract classes, interfaces cannot be changed once they are created. •A new interface needs to be created to create a new version of the existing interface.
Abstract class	•If a programmer wants to create different methods that are useful for different types of objects, it is recommended to create an interface. •There must exist a relationship between the abstract class and the classes that inherit the abstract class. •On the other hand, interfaces are suitable for implementing similar functionalities in dissimilar classes.

C# Common Interfaces

- IComparable
- IComparer
- IComparable<T>

IComparable

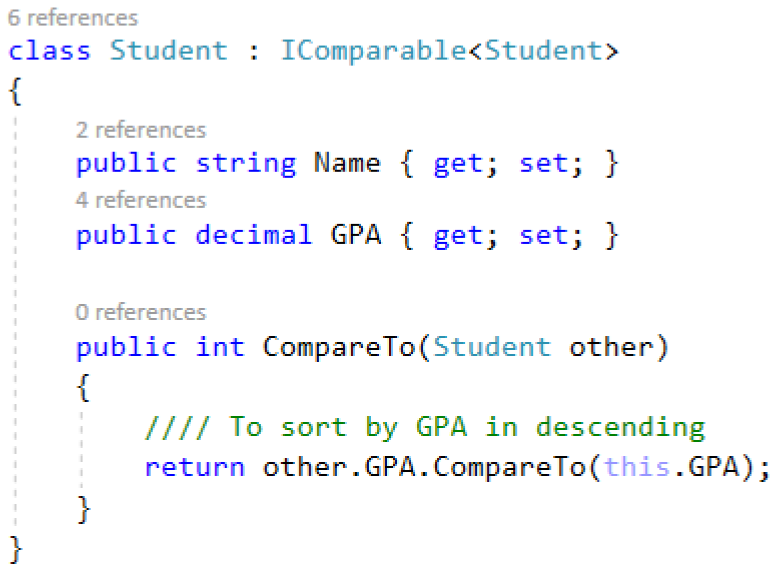
- Defines a generalized type-specific comparison method that a value type or class implements to **order** or **sort** its instances.
- Methods to implement:
  - ✓ public int CompareTo(object obj)

IComparable<T>

- Defines a generalized type-specific comparison method that a value type or class implements to **order** or **sort** its instances.
- Methods to implement:
  - ✓ public int CompareTo(T other)

IComparable

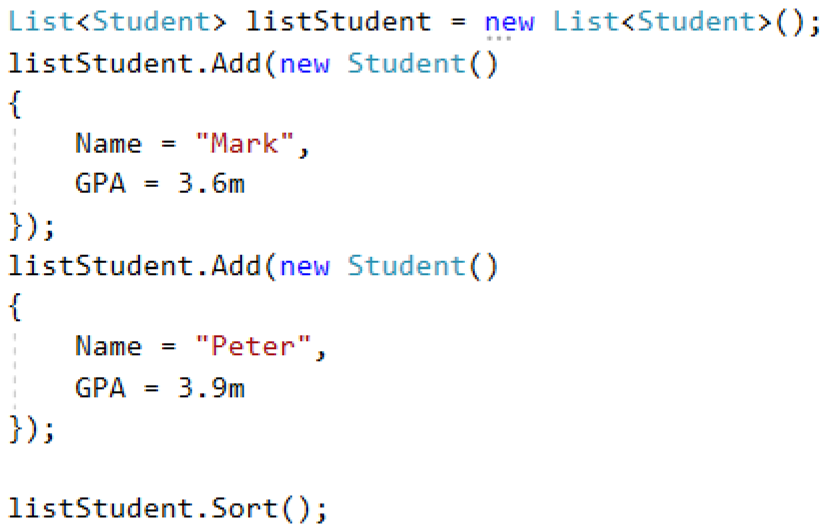
Implement interface



```
6 references
class Student : IComparable<Student>
{
    2 references
    public string Name { get; set; }
    4 references
    public decimal GPA { get; set; }

    0 references
    public int CompareTo(Student other)
    {
        //// To sort by GPA in descending
        return other.GPA.CompareTo(this.GPA);
    }
}
```

then use it



```
List<Student> listStudent = new List<Student>();
listStudent.Add(new Student()
{
    Name = "Mark",
    GPA = 3.6m
});
listStudent.Add(new Student()
{
    Name = "Peter",
    GPA = 3.9m
});

listStudent.Sort();
```

Summary

- An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- IntelliSense provides access to member variables, functions, and methods of an object or a class.
- When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- C# common interfaces ICompareable, IComparer, IEnummerable, IQueryable, IDisposable, IList, ICollection and IDictionary

Thank you