Session 7: Abstract Classes and Interfaces

Objectives

- **Define and describe abstract classes**
- **Explain interfaces**
- **Compare abstract classes and interfaces**

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

◆ Consider the base class, **Animal**, that defines methods such as **Eat()**, **Habitat()**, and **AnimalSound()**.

◆ The **Animal** class is inherited by different subclasses such as **Dog**, **Cat**, **Lion**, and **Tiger**.

◆ The dogs, cats, lions, and tigers neither share the same food, habitat nor do they make similar sounds.

◆ Hence, the **Eat()**, **Habitat()**, and **AnimalSound()** methods must be different for different animals even though they inherit the same base class.

◆ These differences can be incorporated using abstract classes.

Purpose 2-2

◆ Following figure displays an example of abstract class and subclasses:
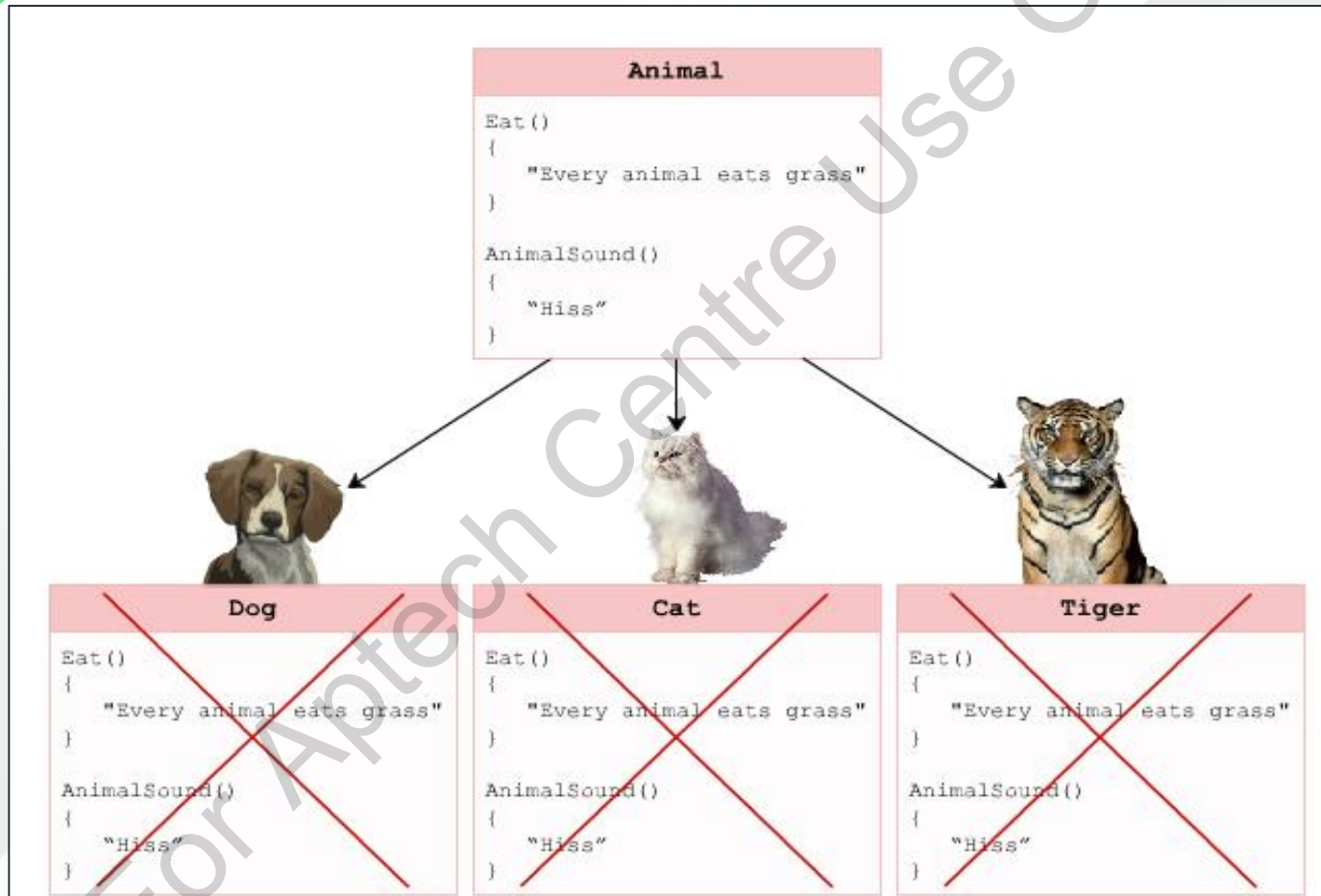
Definition 1-2

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.

◆ Following figure displays the contents of an abstract class:

```
Abstract Class

Method_1 ()

{
// Body
}

AbstractMethod_1 ();  ←
AbstractMethod_2 ();  ←
```
Abstract Methods
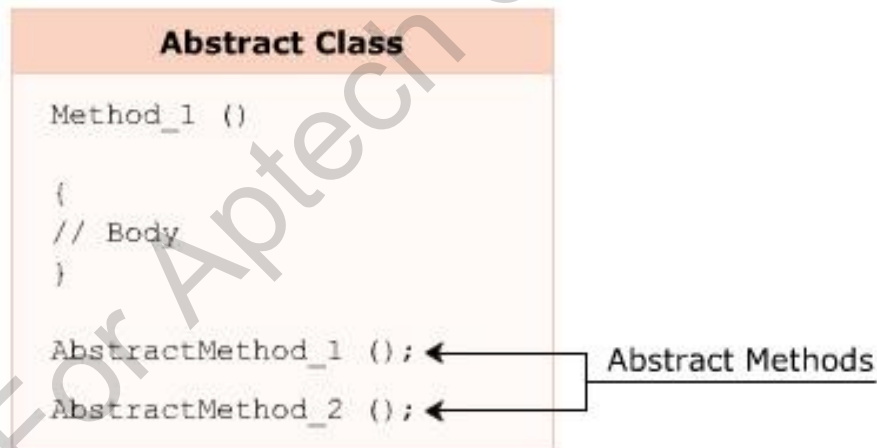
Definition 2-2

◆ 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.

Implementation 1-2

- The subclass inheriting the abstract class has to override and implement the abstract methods with the same name and arguments.

- On failing, the subclass cannot be instantiated as the C# compiler considers it as abstract.

- Following figure displays an example of inheriting an abstract class:
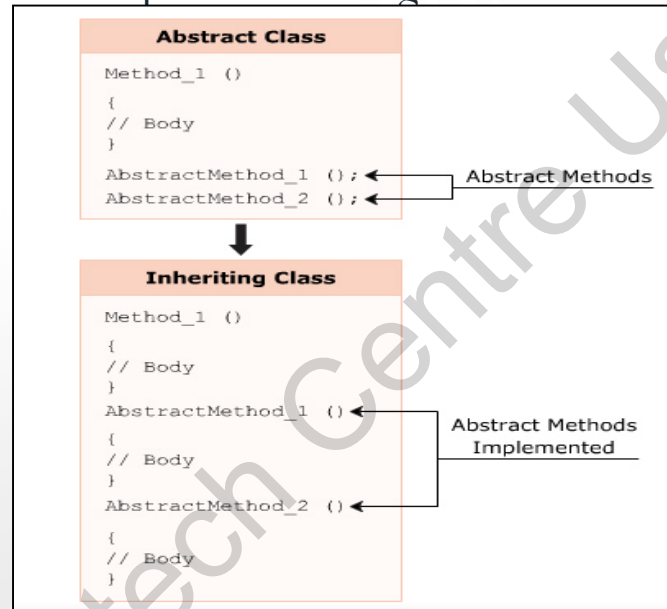
- Following syntax is used to implement an abstract class:

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

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

Implementation 2-2

◆ 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();
   }
}
```

Output

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

Implement Abstract Base Class Using IntelliSense

◆ IntelliSense provides access to member variables, functions, and methods of an object or a class.

◆ IntelliSense can be used to implement system-defined abstract classes.

◆ Steps performed to implement an abstract class using IntelliSense are as follows:

1. Place Cursor	• Place the cursor after the **class IntelliSenseDemo** statement.
2. Type the Following : TimeZone	• The class declaration becomes class IntelliSenseDemo: TimeZone.
3. Click Smart Tag	• Click the smart tag that appears below the **TimeZone** class.
	• IntelliSense provides four override methods from the system-defined TimeZone class to the user-defined **IntelliSenseDemo** class.

Abstract Methods

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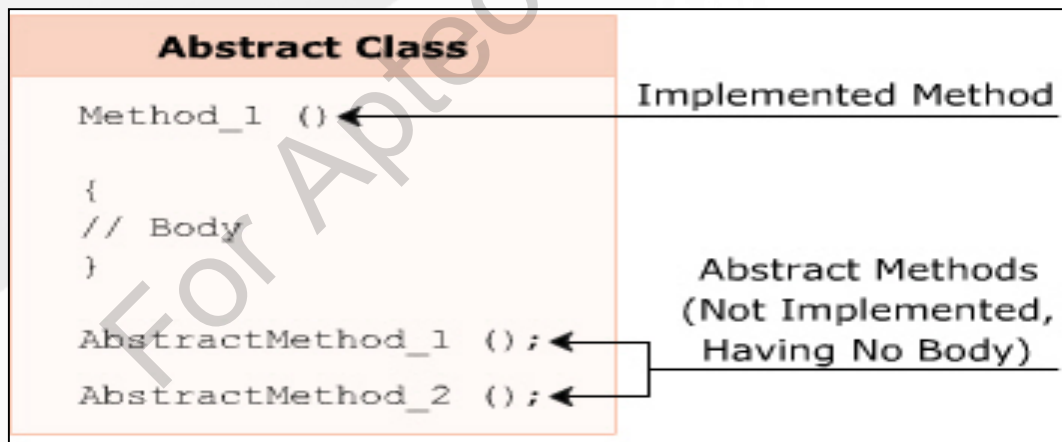
◆ Following are the features of the abstract methods:

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.

◆ Following figure displays an example of abstract methods:

```
Abstract Class

Method_1 ()          Implemented Method

{
// Body
}

AbstractMethod_1 ();    Abstract Methods
                        (Not Implemented,
AbstractMethod_2 ();    Having No Body)
```

Multiple Inheritance through 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.
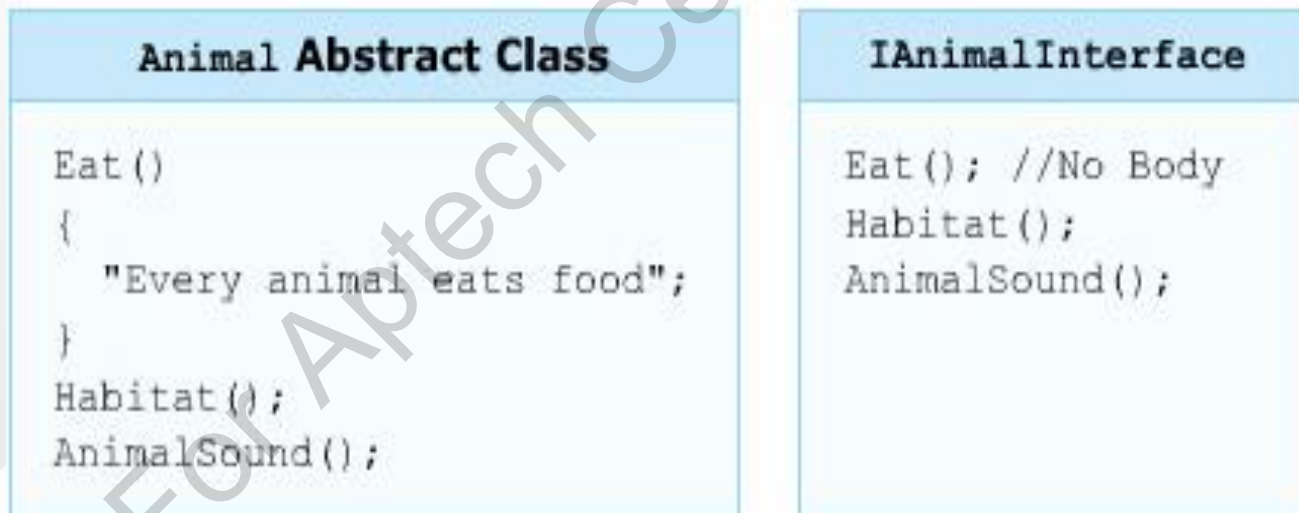
◆ A class in C# can implement multiple interfaces.

Purpose of Interfaces

- Consider a class **Dog** that must 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.

- Following figure displays an example of subclasses with interfaces in C#:
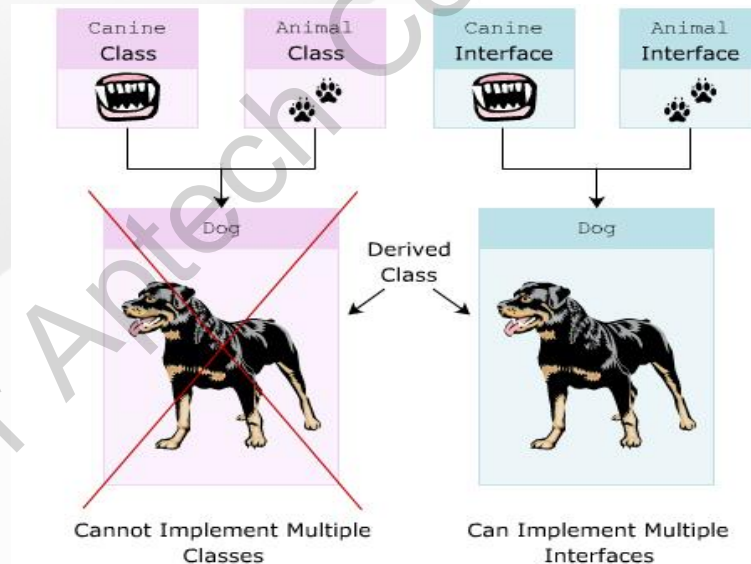
Interfaces 1-2

◆ An interface contains only abstract members that cannot implement any method.

◆ An interface is declared using the keyword interface.

◆ In C#, by default, all members declared in an interface have public as the access modifier.

◆ Following figure displays an example of an interface:

Animal **Abstract Class**
| ```
Eat()
{
   "Every animal eats food";
}
Habitat();
AnimalSound();
``` |

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

Interfaces 2-2

- Following syntax is used to declare an interface:

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

where,

- `interface`: Declares an interface.
- `InterfaceName`: Is the name of the interface.
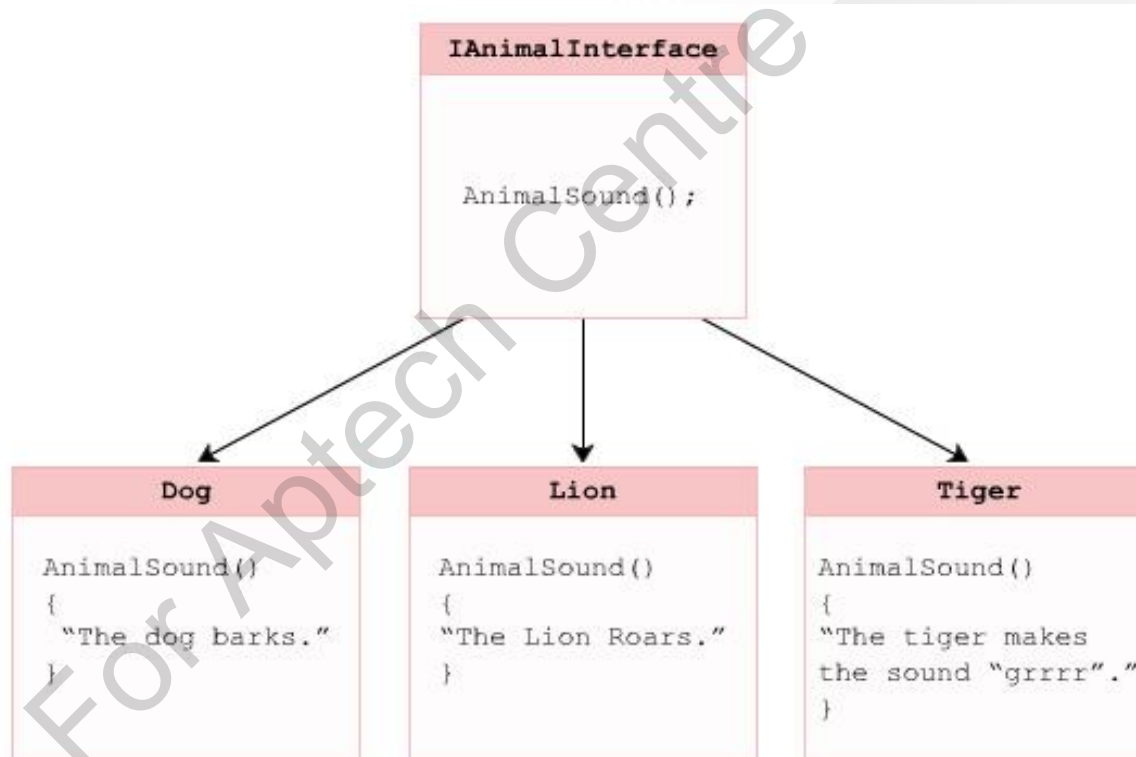
- Following code declares an interface **IAnimal**:

Snippet

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

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

Implementing an Interface 1-3

◆ 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.

◆ 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 2-3

◆ 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.

◆ 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);
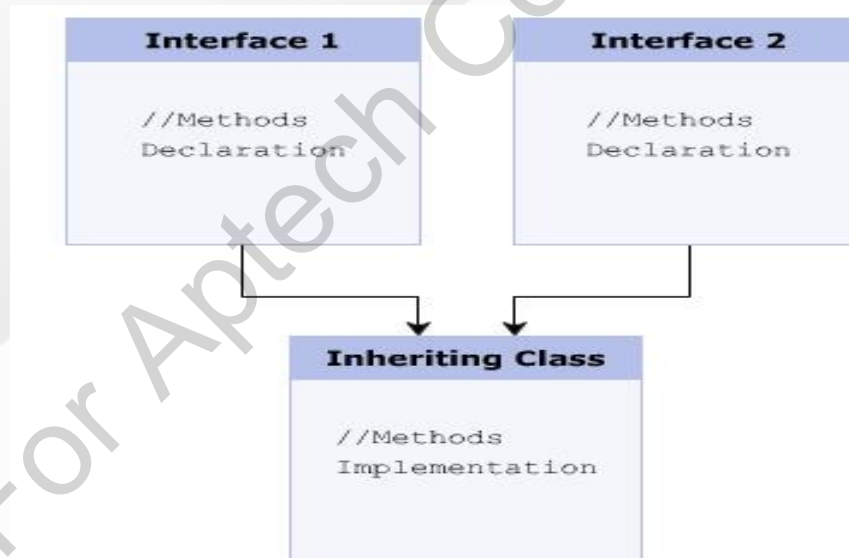      objDog.Habitat();
    }

}
```

Implementing an Interface 3-3

◆ 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 1-2

◆ Multiple interfaces can be implemented in a single class which provides the functionality of multiple inheritance.

◆ 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.

◆ Following figure displays the concept of multiple inheritance using interfaces:

Interfaces and Multiple Inheritance 2-2

◆ 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.

Explicit Interface Implementation 1-3

◆ 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 2-3

◆ Following figure displays the explicit implementation of interfaces:

Explicit Interface Implementation 3-3

◆ Following syntax is used to explicitly implement interfaces:

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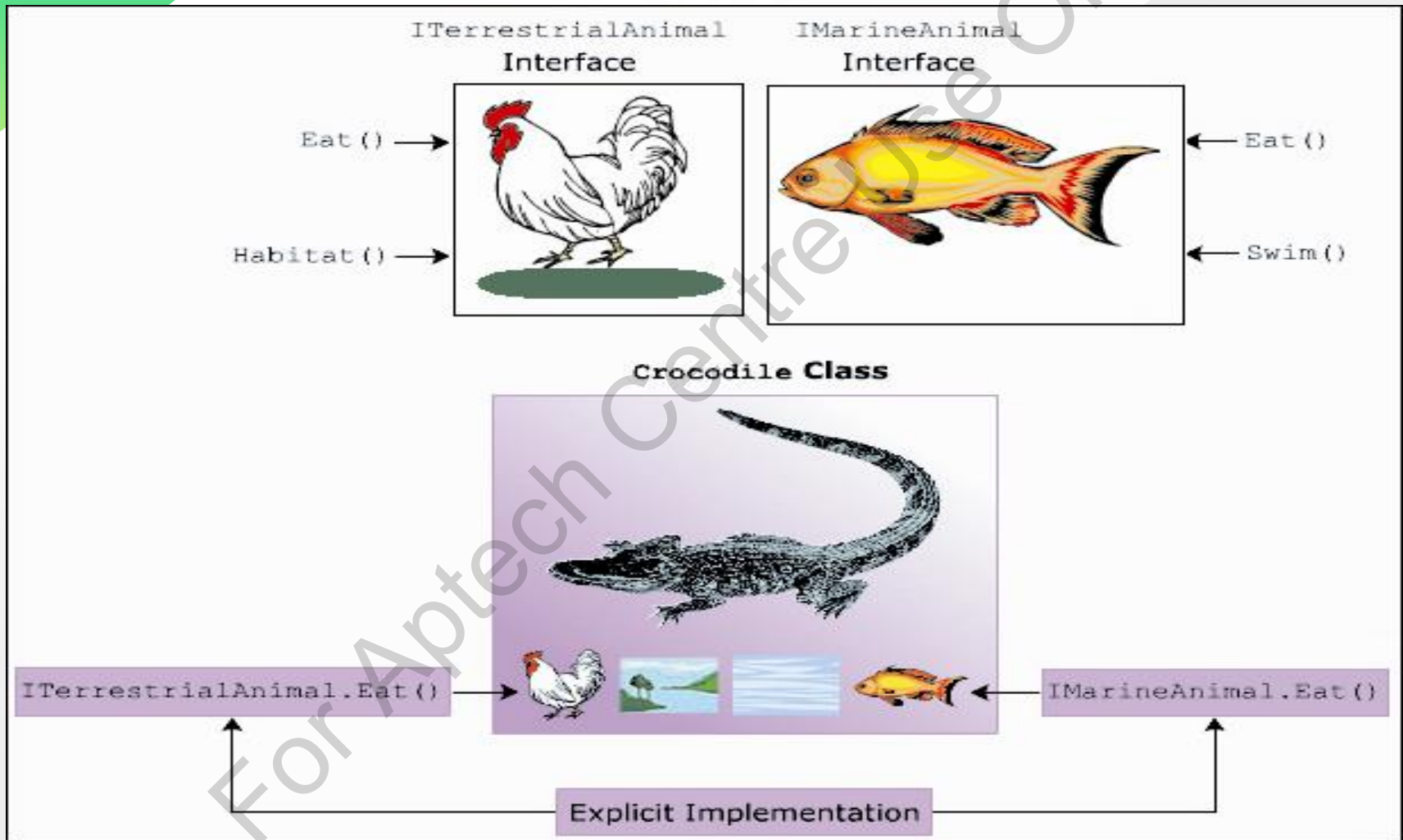class <ClassName> : <Interface1>, <Interface2>
{
    <access modifier> Interface1.Method();
    {
    //statements;
    }
    <access modifier> Interface2.Method();
    {
    //statements;
    }

}
```

◆ where,

◈ `Interface1:` Specifies the first interface implemented.

◈ `Interface2:` Specifies the second interface implemented.

◈ `Method():` Specifies the same method name declared in the two interfaces.

Interface Inheritance 1-5

◆ An interface can inherit multiple interfaces but cannot implement them.

◆ The implementation has to be done by a class.

Example

◈ Consider three interfaces, **IAnimal**, **ICarnivorous**, and **IReptile**.

◈ The interface **IAnimal** declares methods defining general behavior of all animals.

◈ The interface **ICarnivorous** declares methods defining the general eating habits of carnivorous animals.

◈ The interface **IReptile** inherits interfaces **IAnimal** and **ICarnivorous**.

◈ However, these interfaces cannot be implemented by the interface **IReptile** as interfaces cannot implement methods.

◈ The class implementing the **IReptile** interface must implement the methods declared in the **IReptile** interface as well as the methods declared in the **IAnimal** and **ICarnivorous** interfaces.

Interface Inheritance 2-5

◆ Following figure displays an example of interface inheritance:

Interface Inheritance 3-5

◆ Following syntax is used to inherit an interface:

◆ where,

- `InterfaceName`: Specifies the name of the interface that inherits another interface.

- `Inherited_InterfaceName`: Specifies the name of the inherited interface.

Interface Inheritance 4-5

◆ Following code declares interfaces that are inherited by other interfaces:

Snippet

```csharp
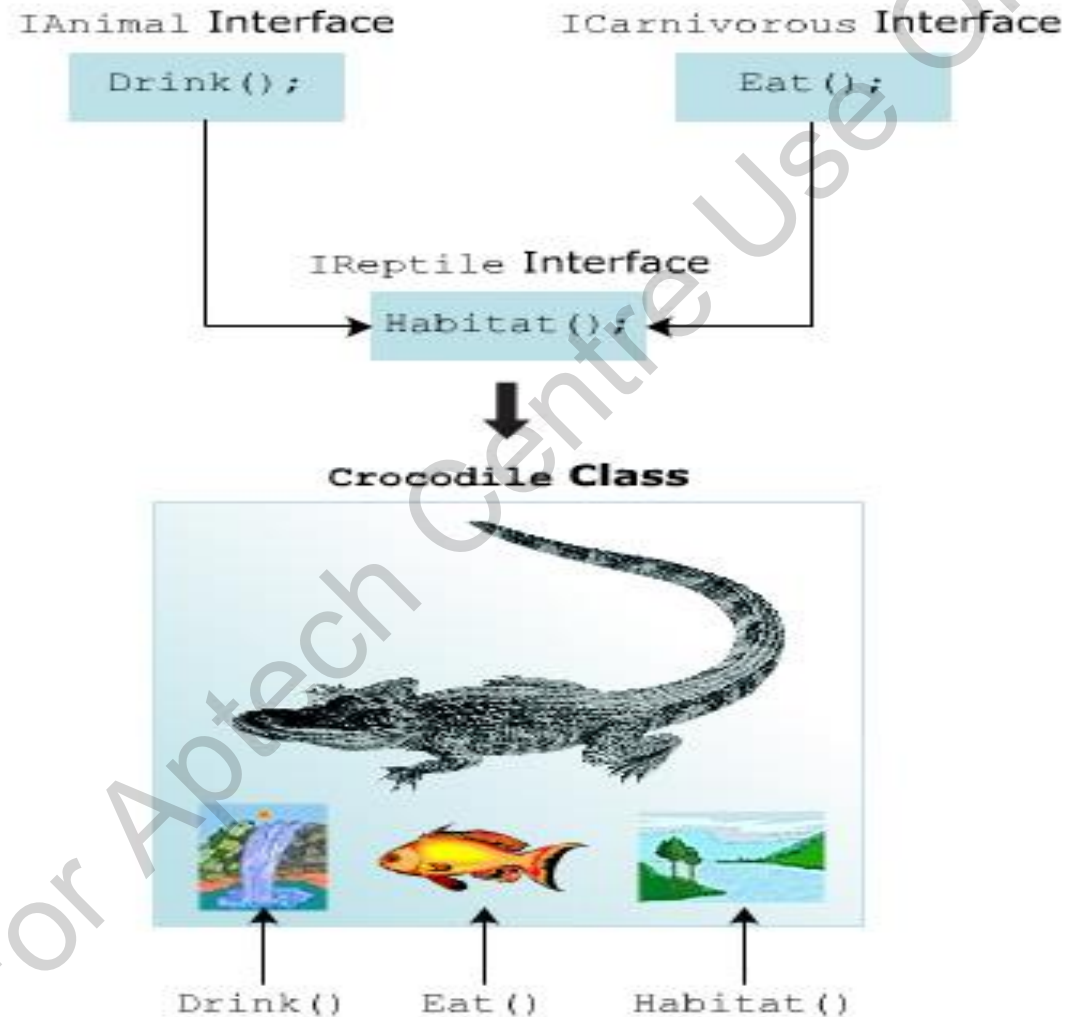interface IAnimal
{
    void Drink();
}
interface ICarnivorous
{
    void Eat();
}
interface IReptile : IAnimal, ICarnivorous
{
    void Habitat();
}
class Crocodile : IReptile
{
    public void Drink()
    {
        Console.WriteLine("Drinks fresh water");
    }
    public void Habitat()
    {
        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 Inheritance 5-5

- In the code:

  - Three interfaces, **`IAnimal`**, **`ICarnivorous`**, and **`IReptile`**, are declared.

  - The three interfaces declare methods **`Drink()`**, **`Eat()`**, and **`Habitat()`** respectively.

  - The **`IReptile`** interface inherits the **`IAnimal`** and **`ICarnivorous`** interfaces.

  - The class **`Crocodile`** implements the interface **`IReptile`**, its declared method **`Habitat()`** and the inherited methods **`Eat()`** and **`Drink()`** of the **`ICarnivorous`** and **`IAnimal`** interfaces.

Interface Re-implementation 1-2

- A class can re-implement an interface.

- Re-implementation occurs when the method declared in the interface is implemented in a class using the `virtual` keyword and this virtual method is then overridden in the derived class.

- Following code demonstrates the purpose of re-implementation of an interface:

Snippet

```
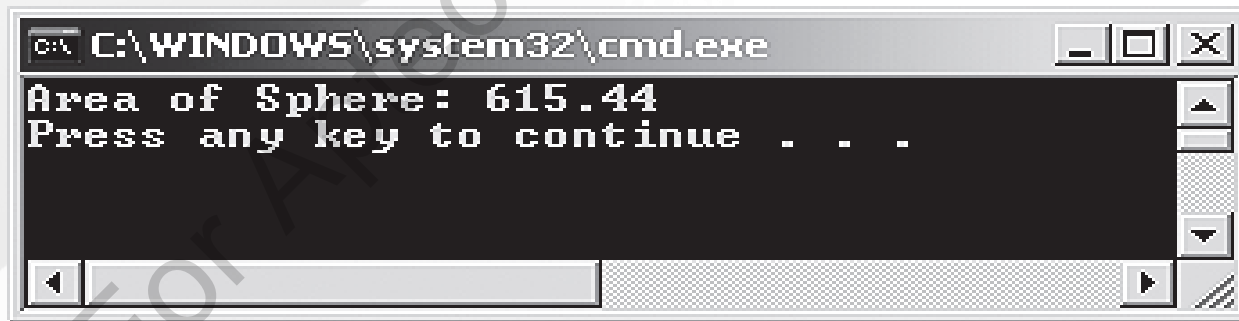using System;
interface IMath {
  void Area();
}
class Circle : IMath
{
    public const float PI = 3.14F;
    protected float Radius;
    protected double AreaOfCircle;
    public virtual void Area()
    {
    AreaOfCircle = (PI * Radius * Radius);
    }
    }
    class Sphere : Circle
    {
    double _areaOfSphere;
    public override void Area()
    {
     base.Area();
     _areaOfSphere = (AreaOfCircle * 4 );
    }
static void Main(string[] args)
{
    Sphere objSphere = new Sphere();
    objSphere.Radius = 7;
    objSphere.Area();
    Console.WriteLine("Area of Sphere: {0:F2}" ,
    objSphere._areaOfSphere);
}

}
```

Interface Re-implementation 2-2

- In the code:
  - The interface **IMath** declares the method **Area()**.
  - The class **Circle** implements the interface **IMath**.
  - The class **Circle** declares a virtual method **Area()** that calculates the area of a circle.
  - The class **Sphere** inherits the class **Circle** and overrides the base class method **Area()** to calculate the area of the sphere.
- Following figure displays the output of re-implementation of an interface:

```
C:\WINDOWS\system32\cmd.exe
Area of Sphere: 615.44
Press any key to continue . . .
```

The `is` and `as` Operators in Interfaces 1-3

◆ The `is` and `as` operators in C# when used with interfaces, verify whether the specified interface is implemented or not.

`is` Operator	`as` Operator
Checks the compatibility between two types or classes and returns a boolean value based on the check operation performed.	Returns null if the two types or classes are not compatible with each other.

The `is` and `as` Operators in Interfaces 2-3

◆ Following code demonstrates an interface with the `is` operator:

Snippet

```
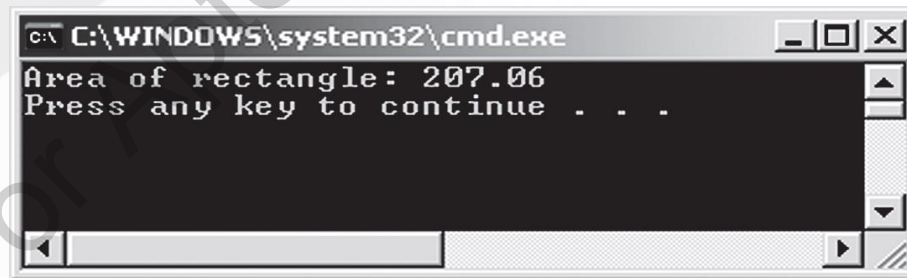using System;
interface ICalculate {
    double Area();
}
class Rectangle : ICalculate{
    float _length;
    float _breadth;
    public Rectangle(float valOne, float valTwo) {
    _length = valOne;
    _breadth = valTwo;
    }
    public double Area() {
    return _length * _breadth;
    }
    static void Main(string[] args) {
        Rectangle objRectangle = new Rectangle(10.2F, 20.3F);
        if (objRectangle is ICalculate) {
        Console.WriteLine("Area of rectangle: {0:F2}" , objRectangle.Area());
        }
        else {
        Console.WriteLine("Interface method not implemented");
        }
    }
}
```

◆ In the code:

- ◈ An interface **ICalculate** declares a method **Area()**.

- ◈ The class **Rectangle** implements the interface **ICalculate** and it consists of a parameterized constructor that assigns the dimension values of the rectangle.

- ◈ The **Area()** method calculates the area of the rectangle. The `Main()` method creates an instance of the class **Rectangle.**

- ◈ The `is` operator is used within the `if-else` construct to check whether the class **Rectangle** implements the methods declared in the interface **ICalculate**.

◆ Following figure displays the output of the example using `is` operator:

```
C:\WINDOWS\system32\cmd.exe
Area of rectangle: 207.06
Press any key to continue . . .
```

Abstract Classes and Interfaces 1-2

◆ Abstract classes and interfaces both declare methods without implementing them.

◆ 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.

Abstract Classes and Interfaces 2-2

- Following figure displays the similarities between abstract class and interface:
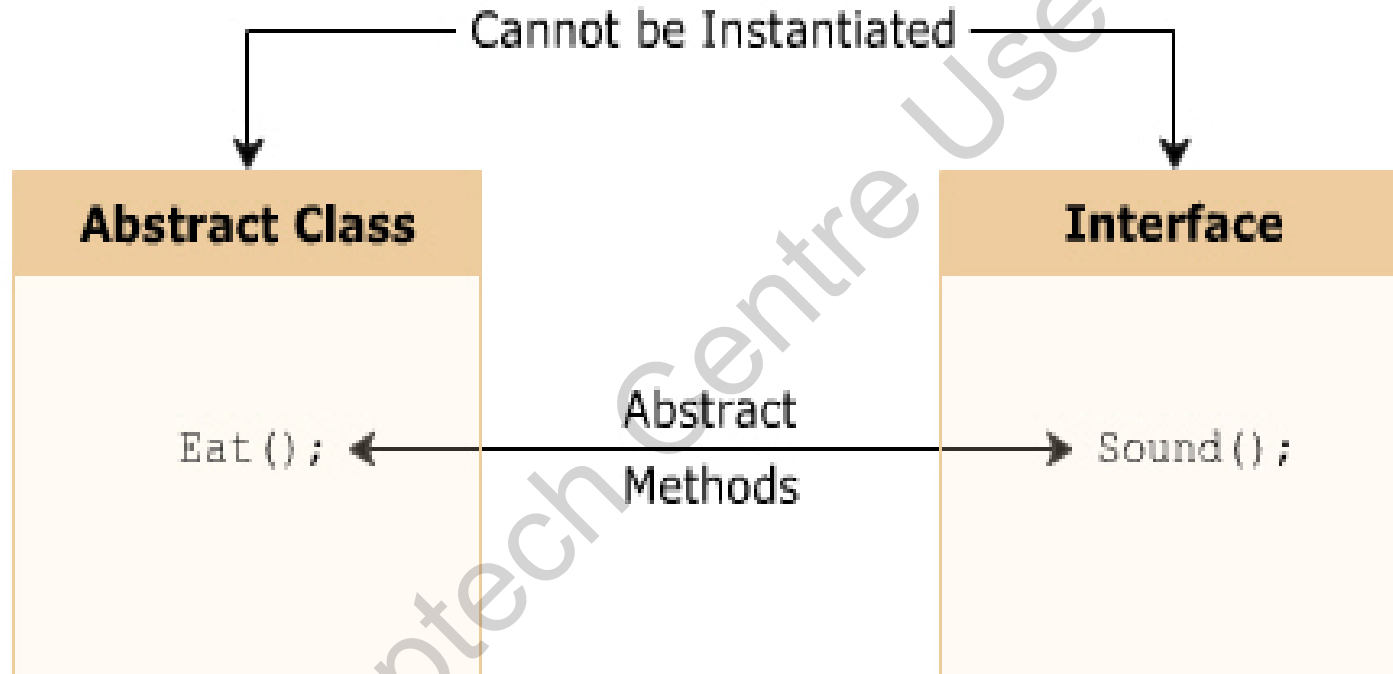
Differences Between an Abstract Class and an Interface

◆ Abstract classes and interfaces are similar because both contain abstract methods that are implemented by the inheriting class.

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 must implement common methods and declare common abstract methods.	An interface is a better option when you must declare only abstract methods.
An abstract class can declare constructors and destructors.	An interface cannot declare constructors or destructors.

Recommendations for Using Abstract Classes and Interfaces

◆ An abstract class can inherit another class whereas an interface cannot inherit a class.

◆ 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. • Unlike abstract classes, interfaces cannot be changed once they are created. • A new interface must 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.

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 must 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.

◆ Re-implementation occurs when the method declared in the interface is implemented in a class using the `virtual` keyword and this virtual method is then, overridden in the derived class.

◆ The is operator is used to check the compatibility between two types or classes and as operator returns null if the two types or classes are not compatible with each other.