MT UNIT-2

By

K. Bhaskara Rao

Asst. Prof, IT Dept.

Topics

- Types of Parameters → value parameters, ref, out, parameter array
- Method Overloading and Constructor overloading
- Types of constructors → default, parameterized, private, static, copy constructor
- Default constructor → a constructor with out any parameters. This is called when class is instantiated
-

Classes & Objects

- A *class is a template that defines the form of an*
- object. It specifies both the data and the code that will operate on that data.
- C# uses a class specification to construct *objects.*
- *Objects are instances of a class.* class is a logical abstraction.
- It is not until an object of that class has been created that a physical representation of that class exists in memory.
- When you define a class, you declare the data that it contains and the code that operates on it. While very simple classes might contain only code or only data, most real-world classes contain both.
- data is contained in *data members defined by the class, and code is* contained in *function members.*
- **data members** (also called *fields) include instance variables and static variables.* ***Function members*** *include methods,* constructors, destructors, indexers, events, operators, and properties

General Form of a class

class *classname {*
// declare instance variables
access type var1;
access type var2;
// ...
access type varN;
// declare methods
access ret-type method1(parameters) {
// body of method
}
access ret-type method2(parameters) {
// body of method
}
// ...
access ret-type methodN(parameters) {
// body of method
}
}

Class example

```
class Class1
{
        static void Main(string[] args)
        {
                // Your code would go here, e.g.
                Console.WriteLine("hi");
        }
        /* We can define other methods and vars for the class */
        // Constructor
        Class1()
        {
                // Code
        }
        // Some method, use public, private, protected
        // Use static as well just like Java
        public void foo()
        {
                // Code
        }
        // Instance, Static Variables
        private int m_number;
        public static double m_stuff;
}
```

Class Example

```
class Building {
public int Floors; // number of floors
public int Area; // total square footage of building
public int Occupants; // number of occupants
}
```

Object creation:

```
Building house = new Building();     // create an object of type
                                        building
```

Accessing members :

```
house.Floors = 2;
```

Class Example

```
public class BankAccount
{
        public double m_amount;
        BankAccount(double d) {
                m_amount = d;
        }
        public virtual string GetInfo() {
                return “Basic Account”;
        }
}

public class SavingsAccount : BankAccount
{
        // Savings Account derived from Bank Account
        // usual inheritance of methods, variables
        public double m_interest_rate;
        SavingsAccount(double d) : base(100) { // $100 bonus for signup
                m_interest_rate = 0.025;
        }
        public override string GetInfo() {
                string s = base.GetInfo();
                return s + “ and Savings Account”;
        }
}
```

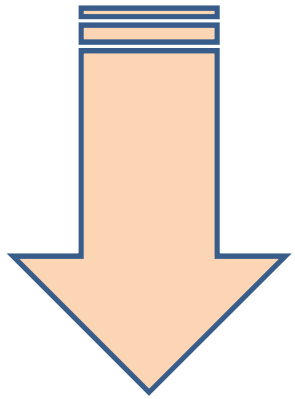
Sample Class Usage

```
SavingsAccount a = new SavingsAccount(0.05);
CoConsole.WriteLine(a.m_interest_rate);
Console.WriteLine(a.GetInfo());
```

Then the output is:

```
        100
        0.05
        Basic Account and Savings Account
nsole.WriteLine(a.m_amount);
```

Constructors



Constructor

A *constructor* *initializes an object when it is created. It has the same name as its class and* is syntactically similar to a method. However, constructors have no explicit return type.

- ✓ The general form of a constructor is shown here:

```
access class-name(param-list) {
// constructor code
}
```

- ✓ Typically, you will use a constructor to give initial values to the instance variables defined by the class.
- ✓ usually, *access is* ***public because constructors are normally called from outside*** their class. The *param-list can be empty, or it can specify one or more parameters.*
- C# automatically provides a default constructor that causes all member variables to be initialized to their default values. For most value types, the default value is zero. For **bool, the default is false.** For reference types, the default is null.

Parameterized Constructors

- ✓ Most often you will need a constructor that has one or more parameters.
- ✓ Parameters are added to a constructor in the same way they are added to a method: just
- ✓ Declare them inside the parentheses after the constructor's name.

Constructor

```
access class-name(param-list) {
// constructor code
}
```

Ex:

```
// A parameterized constructor.
using System;
class MyClass {
public int x;
public MyClass(int i) {
x = i;
}
}
class ParmConsDemo {
static void Main() {
MyClass t1 = new MyClass(10);
MyClass t2 = new MyClass(88);
Console.WriteLine(t1.x + " " + t2.x);
}
}
```

Parameters to methods

- value types → built in types, structs, enum etc…
- reference types → arrays, class instances etc…
- ref type
- out type
- params → allows variable no of parameters to methods

const, readonly keywords

```
class A{
 public const int i=10;   //static by default
public readonly int j;
void m1()
{
   j=20;   //after assignment j becomes const
}
}
```

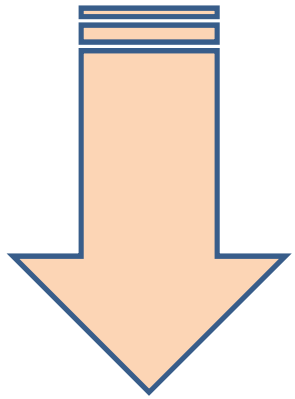
constructors

- default constructor
- parameterized constructor
- private constructor
- static constructor
- copy constructor
- overloaded constructor

Invoke an Overloaded Constructor Through this keyword:

constructor-name(parameter-list1) : this(parameter-list2) {
// ... body of constructor, which may be empty
}

base keyword



base Keyword – calling base class constructor

- ✓ When a derived class specifies a **base** clause, it is calling the constructor of its immediate base class.
- ✓ **base always refers** to the base class immediately above the calling class
- ✓ You pass arguments to the base constructor by specifying them as arguments to **base.**
- ✓ A reference variable of a base class can be assigned a reference to an object of any class derived from that base class.

Syntax:

derived-constructor(parameter-list) : base(arg-list) {

// body of constructor

}

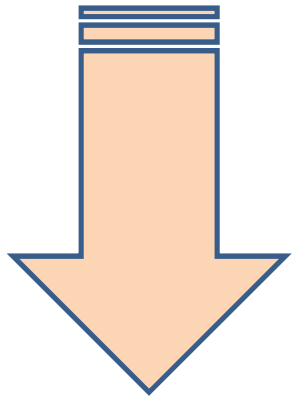
arg-list → *specifies any arguments needed by the constructor in the base class*

base Keyword – calling base class constructor

Example:

```
class A {
    public int i = 0;
}
// Create a derived class.
class B : A {
    new int i; // this i hides the i in A
    public B(int a, int b) {
    base.i = a; // this uncovers the i in A
    i = b; // i in B
}
```

Member access

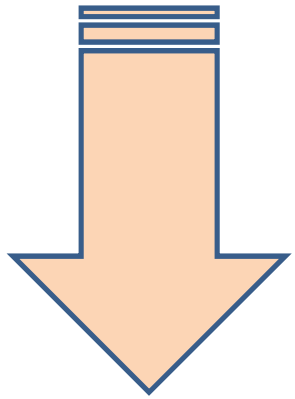


Class Member Access

	public	protected	internal	protected internal	private	private protected
Entire program	Yes	No	No	No	No	No
Containing class	Yes	Yes	Yes	Yes	Yes	Yes
Current assembly	Yes	No	Yes	Yes	No	No
Derived types	Yes	Yes	No	Yes	No	No
Derived types within current assembly	Yes	Yes	Yes	Yes	No	Yes

- ✓
- ✓
- ✓
- ✓

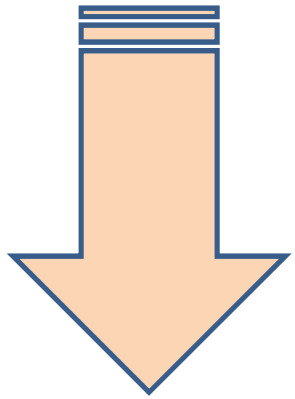
Method overloading & Constructor overloading



Method overriding using inheritance

```
using System;
class a //base class
{
    public void cname()
    {
        Console.WriteLine("This is first method");
    }
}
```

Member access



Method Overloading

- ✓ In C#, two or more methods within the same class can share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- ✓ Method overloading is one of the ways that C# implements polymorphism.
- ✓ In declaring Overloaded Methods, It is not sufficient for two methods to differ only in their return types. They must differ in the types or number of their parameters.
- ✓ When an overloaded method is called, the version of the method executed is the one whose parameters match the arguments.

```
class b : a //drived class
{
    new public  void cname()
// new keyword is used to hiding method cname of
class a
    {
        Console.WriteLine("This is second method");
    }
    public static void Main()
    {
        b obj = new b();
        b.cname();
        Console.Read();
    } }
```

Arrays

Method or property	Purpose
Reverse( )	Overloaded public static method that reverses the order of the elements in a one-dimensional array.
Sort( )	Overloaded public static method that sorts the values in a one-dimensional array.
IsFixedSize	Required because Array implements ICollection. With arrays, this will always return true (all arrays are of a fixed size).
IsReadOnly	Public property (required because Array implements IList) that returns a Boolean value indicating whether the array is read-only.
IsSynchronized	Public property (required because Array implements ICollection) that returns a Boolean value indicating whether the array is thread-safe.
Length	Public property that returns the length of the array.
Rank	Public property that returns the number of dimensions of the array.

Class member visibility

Class Member Visibility Levels

Visibility Keyword	Description
public	This member is accessible from both inside the assembly and outside the assembly, as well as by members inside and outside the class.
private	This member is accessible only from code within the class definition itself.
protected	This member is accessible only to derived types, both inside and outside the assembly.
internal	This member is accessible by any code within the assembly.
public protected	This member is accessible by all code within the assembly, but only by derived types outside the assembly.
private protected	This member is protected within the assembly (accessible only to derived types), but is inaccessible outside the assembly.

Class member visibility

Class Member Visibility:

Object Oriented Programming

- Object Oriented Programming (OOP) is a programming model where programs are organized around objects and data rather than action and logic.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.

Abstraction :

Representing essential features without representing the background details

✓ Abstraction lets you focus on what the object does instead of how it does it.

Object Oriented Programming

```
abstract class MobilePhone
    {
        public void Calling();
        public void SendSMS();
    }

    public class Nokia1400 : MobilePhone
    {
    }
```

Object Oriented Programming

Encapsulation: Wrapping up a data member and a method together into a single unit (in other words class) is called Encapsulation.

Example for encapsulation is class.

```
class Bag
  {
      book;
      pen;
      ReadBook();
  }
```

Encapsulation means hiding the internal details of an object, in other words how an object does something.

Object Oriented Programming

```
class Demo
  {
     private int _mark;

     public int Mark
     {
        get { return _mark; }
        set { if (_mark > 0) _mark = value; else _mark = 0; }
     }
  }
```

Ex: Mobile user learns how to use the phone but not how it works.

Numeric Custom Formatting Specifiers

Numeric Custom Format Specifiers

Specifier	Description
0	The zero placeholder.
#	The digit placeholder. If the given value has a digit in the position indicated by the # specifier, that digit is displayed in the formatted output.
.	Decimal point.
,	Thousands separator.
%	Percentage specifier. The value being formatted will be multiplied by 100 before being included in the formatted output.
E0/E+0/e/e+0/e-0/E	Scientific notation.
'XX' or "XX“	Literal strings. These are included literally in the formatted output without translation in their relative positions.
;	Section separator for conditional formatting of negative, zero, and positive values.

Arrays

Initialization:

int[] nums;

nums = new int[] { 99, 10, 100, 18, 78, 23,63, 9, 87, 49 };

Multi dimensiona arrays:

int[,] table = new int[10, 20];

type[, ...,] name = new type[size1, size2, ..., sizeN];

int[,,] multidim = new int[4, 10, 3];

multidim[2, 4, 1] = 100;

Initialization:

type[,] array_name = {

{ *val, val, val, ..., val },*

{ *val, val, val, ..., val },*

...

{ *val, val, val, ..., val }*

};

Arrays

System.Array methods and properties:

Method or property	Purpose
BinarySearch( )	Overloaded public static method that searches a one-dimensional sorted array.
Clear( )	Public static method that sets a range of elements in the array either to 0 or to a null reference.
Copy( )	Overloaded public static method that copies a section of one array to another array.
CreateInstance( )	Overloaded public static method that instantiates a new instance of an array.
IndexOf( )	Overloaded public static method that returns the index (offset) of the first instance of a value in a one-dimensional array.
LastIndexOf( )	Overloaded public static method that returns the index of the last instance of a value in a one-dimensional array.

Object Oriented Programming

Inheritance:

✓ When a class includes a property of another class it is known as inheritance.
✓ Inheritance is a process of object reusability.

Ex:
```
public class ParentClass
    {
        public ParentClass()
        {
            Console.WriteLine("Parent Constructor.");
        }
        public void print()
        {
            Console.WriteLine("I'm a Parent Class.");
        }
    }
public class ChildClass : ParentClass
    {
        public ChildClass()
        {
            Console.WriteLine("Child Constructor.");
        }
        public static void Main()
        {
            ChildClass child = new ChildClass();
            child.print();
        }
    }
```

inheritance

Syntax :

```
class derived-class-name : base-class-name {
   // body of class
}
```

inheritance

```
class TwoDShape {
public double Width;
public double Height;
public void ShowDim() {
      Console.WriteLine("Width and height are " +
      Width + " and " + Height);
}
}
// Triangle is derived from TwoDShape.
class Triangle : TwoDShape {
      public string Style; // style of triangle
      // Return area of triangle.
      public double Area() {
      return Width * Height / 2;
      }
/     / Display a triangle's style.
      public void ShowStyle() {
      Console.WriteLine("Triangle is " + Style);
      }
}
```

Object Oriented Programming

Polymorphism: Polymorphism means one name, many forms.

•One function behaves in different forms.

•Many forms of a single object is called Polymorphism.

Ex: A person behaves the son in a house at the same time that the person behaves an employee in an office.

Ex: Your mobile phone, one name but many forms:

•As phone

•As camera

•As mp3 player

•As radio

Class: It is template or blueprint that describes the details of an object.

Object:

static keyword

- Applies for classes, methods, variables and constructor.
- Normally, a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static.**
- When a member is declared static, it can be accessed before any objects of its class are created and without reference to any object.
- You can declare **both methods and variables to be static.**
- Because **static fields are independent of any specific object, they are useful when you** need to maintain information that is applicable to an entire class.
- The most common example of a static member is **Main( ),** which is declared static because it must be called by the operating system when your program begins.
- Outside the class, to use a static member, you must specify the name of its class followed by the dot operator. No object needs to be created. In fact, a static member cannot be accessed through an object reference. It must be accessed through its class name. For example, if you want to assign the value 10 to a static variable called count that is part of a class called Timer, use this line:

```
Timer.count = 10;
```

- A **static method can be called in the same way—by use** of the dot operator on the name of the class.

static keyword

- Variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, **all instances of the class share the same static variable.**
- **A static variable is initialized before its class is used.** If no explicit initializer is specified, it is initialized to zero for numeric types, null in the case of reference types, or false for variables of type bool. Thus, a static variable always has a value.
- The difference between a static method and a normal method is that the static method can be called through its class name, without any instance of that class being created. You have seen an example of this already: the Sqrt( ) method, which is a static method within C#'s System.Math class.

static methods and static variables

- Methods declared with static keyword are called static methods..

Rules:

- • A static method does not have a this reference. This is because a static method does not execute relative to any object.
- • A static method can directly call only other static methods of its class. It cannot
- directly call an instance method of its class. The reason is that instance methods
- operate on specific objects, but a static method is not called on an object. Thus, on what object would the instance method operate?
- • A similar restriction applies to static data. A static method can directly access only other static data defined by its class. It cannot operate on an instance variable of its class because there is no object to operate on.
- Error occurs when trying to call a non-**static method from within a static** method of the same class.

static method accessing a non static method - Error

```
using System;
class AnotherStaticError {
// A non-static method.
void NonStaticMeth() {
Console.WriteLine("Inside NonStaticMeth().");
}
/* Error! Can't directly call a non-static method
from within a static method. */
static void staticMeth() {
NonStaticMeth(); // won't compile
}
}
```

static classes

- A class can be declared **static. There are two key features of a static class. First, no object of** a **static class can be created. Second, a static class must contain only static members. A static** class is created by modifying a class declaration with the keyword **static, shown here:**

  static class *class-name { // ... }*

- Within the class, all members must be explicitly specified as **static. Making a class static** does not automatically make its members **static.**

Use of static classes:

- static classes have two primary uses. First, a static class is required when creating an *extension method.*
- Second, a static class is used to contain a collection of related static methods.

static constructor

- A constructor can also be specified as **static. A static constructor is typically used to** initialize features that apply to a class rather than an instance.
- Thus, it is used to initialize aspects of a class before any objects of the class are created.

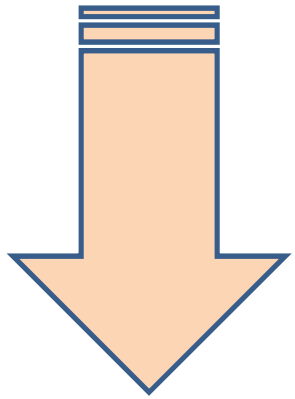
Topics

- Properties - examples
- Base reference – Derived objects
- Using abstract classes
- Exception Handling

Base reference – Derived objects

- C# is a strongly typed language. Aside from the standard conversions and automatic promotions that apply to its value types, type compatibility is strictly enforced.
- Therefore, a reference variable for one class type cannot normally refer to an object of another class type.
- In general, an object reference variable can refer only to objects of its type. There is, however, an important exception to C#'s strict type enforcement. A reference
- variable of a base class can be assigned a reference to an object of any class derived from that base class. This is legal because an instance of a derived type encapsulates an instance of the base type. Thus, a base class reference can refer to it.

virtual Methods & Overriding



Virtual methods & Overriding – virtual

- A *virtual method is a method that is declared as* ***virtual in a base class.***
- You declare a method as virtual inside a base class by preceding its declaration with the keyword **virtual.**
- ***The defining*** characteristic of a virtual method is that it can be redefined in one or more derived classes. Thus, each derived class can have its own version of a virtual method.
- Virtual keyword allows redefining the methods in the derived classes that have the same base class.
- when one is called through a base class reference. In this situation, C# determines which version of the method to call based upon the *type of the* object *referred to by the reference—and this determination is made at runtime. Thus, when* different objects are referred to, different versions of the virtual method are executed.

- In Either words, it is the type of the object being referred to (not the type of the reference) that determines which version of the virtual method will be executed. Therefore, if a base class contains a virtual method derived from that base class, then when different types of objects are referred to through a base class reference, different versions of the virtual method can be executed..

Overriding

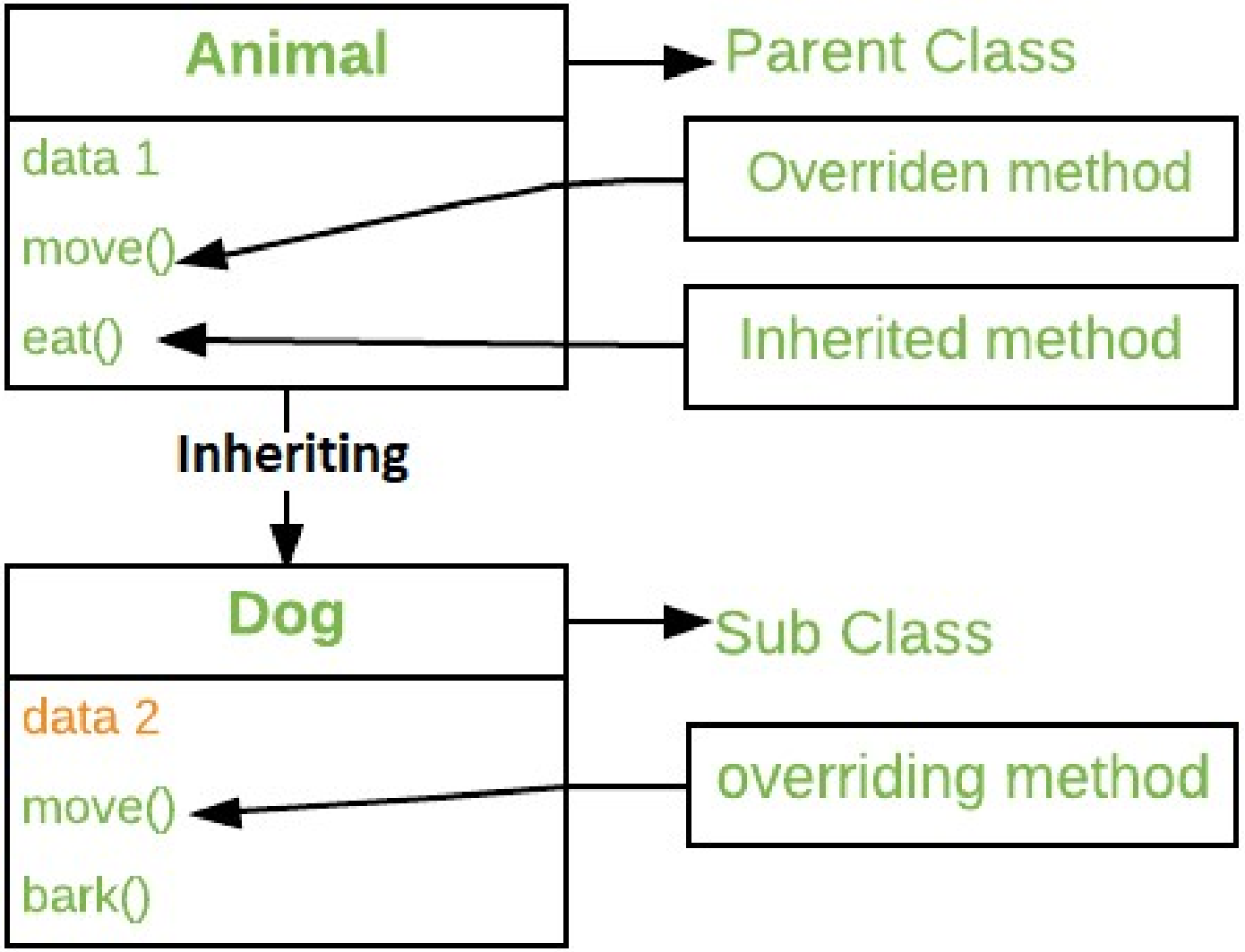
- Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.
- When a virtual method is redefined by a derived class, the **override modifier**

is used.

- Thus, the process of redefining a virtual method inside a derived class is called *method overriding.*
- When overriding a method, the name, return type, and signature of the overriding
- method must be the same as the virtual method that is being overridden. Also, a virtual method cannot be specified as **static or abstract .**

- Method overriding forms the basis for one of C#'s most powerful concepts: ***dynamic method dispatch****. Dynamic method dispatch is the mechanism by which a call to an overridden* method is resolved at runtime, rather than compile time. Dynamic method dispatch is important because this is how C# implements **runtimr polymorphism.**
- Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class. Method overriding is one of the ways by which C# achieve *Run Time Polymorphism(Dynamic Polymorphism)*.

Overriding


Animal
Parent Class
data 1
move()
eat()
Overriden method
Inherited method
Inheriting
Dog
Sub Class
data 2
move()
bark()
overriding method

Interfaces

- you can fully separate a class' interface from its implementation by using the keyword **interface.**
- Interfaces are syntactically similar to abstract classes. However, in an interface, no method can include a body.
- one class can implement any number of interfaces.
- More than one class can implement the same interface. So code can use any object of these classes because the interface to these objects is same.
- can have events, indexers, methods & properties declarations.
- They are useful for putting together plug-and-play architectures, where components can be interchanged at will.

Definition of interface:

interface *name {*

ret-type method-name1(param-list);

ret-type method-name2(param-list);

// ...

ret-type method-nameN(param-list);

}

Interfaces

- In an interface, no method can have an implementation. Thus, each class that includes an interface must

  implement all of the methods.

Example:

```
public interface ISeries {
int GetNext(); // return next number in series
void Reset(); // restart
void SetStart(int x); // set starting value
}
```

Definition of class implementing the interface:

```
class class-name : interface-name {
// class-body
}
```

Interfaces

- An interface reference can refer to any object of a class that implements the interface.

// interface property

type name {

get;

set;

}

- interfaces can be inherited

Interfaces

- An interface in C# is much like an interface in Java
- An interface states what an object can do, but not how it is done.
  - It looks like a class definition but we cannot implement any methods in the interface nor include any variables.
- Here is a sample interface:

Interface example

```
public interface IDrivable {
        void Start();
        void Stop();
        void Turn();
}
```

```
public class SportsCar : IDriveable {
        void Start() {
          // Code here to implement start
        }
        void Stop() {
          // Code here to implement stop
        }
        void Turn() {
          // Code here to implement turn
        }
}
```

Method that uses the Interface:

```
void GoForward(IDrivable d)
{
        d.Start();
        // wait
        d.Stop();
}
```

Interfaces

An interface is a contract that guarantees to a client how a class or struct will behave.

When a class implements an interface, it tells any potential client "I guarantee I'll support the methods, properties, events, and indexers of the named interface.“

An interface offers an alternative to an abstract class for creating contracts among classes and their clients.

Syntactically, an interface is like a class that has only abstract methods.

Syntax for interface :

```
[attributes] [access-modifier] interface interface-name [:base-list] {
        interface-body
}
```

The base-list lists the interfaces that this interface extends.

The interface-body is the implementation of the interface

The purpose of an interface is to define the capabilities that you want to have available in a class.

Interfaces

- Classes can implement more than one interface
- Example : public class Document : IStorable, Icompressible
- It is possible to extend an existing interface to add new methods or members, or to modify how existing members work
- An implementing class is free to mark any or all of the methods that implement the interface as virtual. Derived classes can override or provide new implementations

```
E xample interface :
 interface I C reature
 {
   int Num Legs {  put;  get;  }
   string Color {  get; set; }
 }
```

- Note : There are no accessibility kewords in the interface.
- - In the class that implementing interface, all the methods should have an access specifier ‘public’.

THANK YOU