C# 7.0

Lesson 04 : OOP Using C#

Lesson Objectives

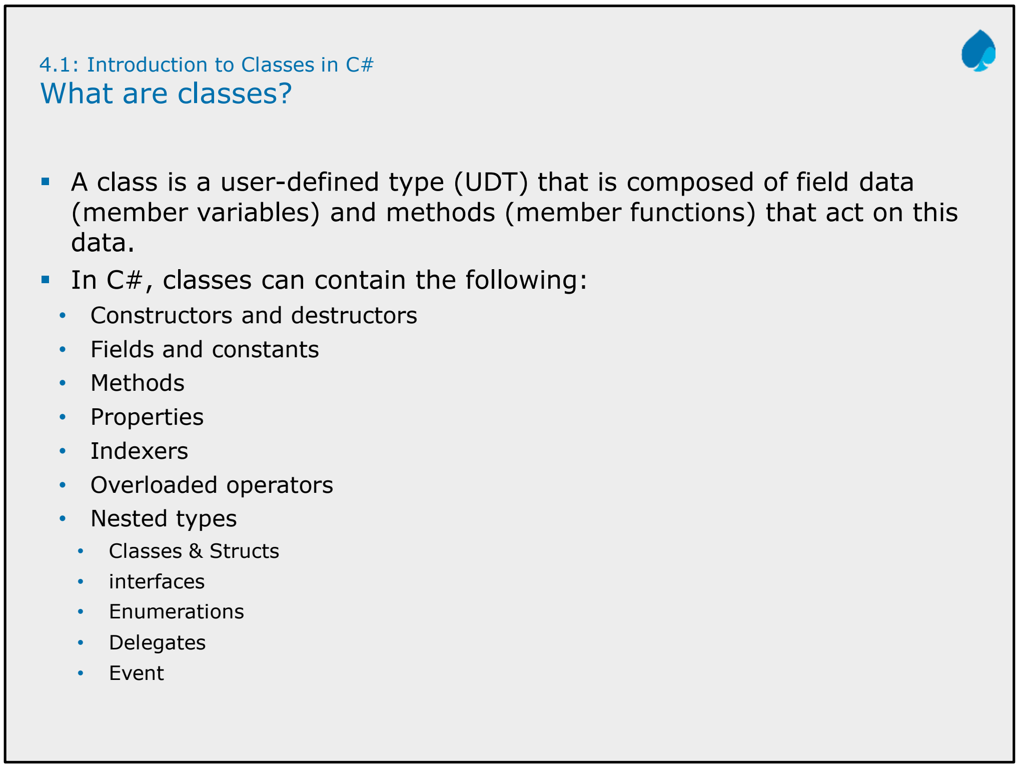
- What are classes?
- Classes in C# - An Example
- Definition and Types of Constructor
- Constructor Example
- Destructors in C#
- Local, Instance & Static Variables
- Definition of Method
- Example of a Class Method
- The Value Parameter
- The Params Parameter
- Method Overloading And Polymorphism
- Understanding Operator Overloading in C#

Lesson Objectives

- Function Overriding By Polymorphism
- Function Overriding By Polymorphism – Virtual Methods
- What Is A Static Constructor?
- Types Of Class Accessibility
- What Is Class Inheritance?
- Constructors And Their Inheritance
- Hiding Name Of Base Class Member
- Reference Of Derived Object To Base Variable
- Introduction to Access Modifiers in C#
  What Is An Abstract Class?
- An Example : Abstract Class
- Characteristics of Abstract Methods
- Abstract Class, Virtual and Abstract Methods
- Characteristics Of Sealed Class

Lesson Objectives

- Sealed Class : An Example
- Sealed Static Class
- What are Interfaces?
- Implementation of Interfaces
- Interfaces : An Example
- Use of Structs in C#
- Structs Vs Classes in C#
- Structs : An Example
- Introduction to Namespaces in C#



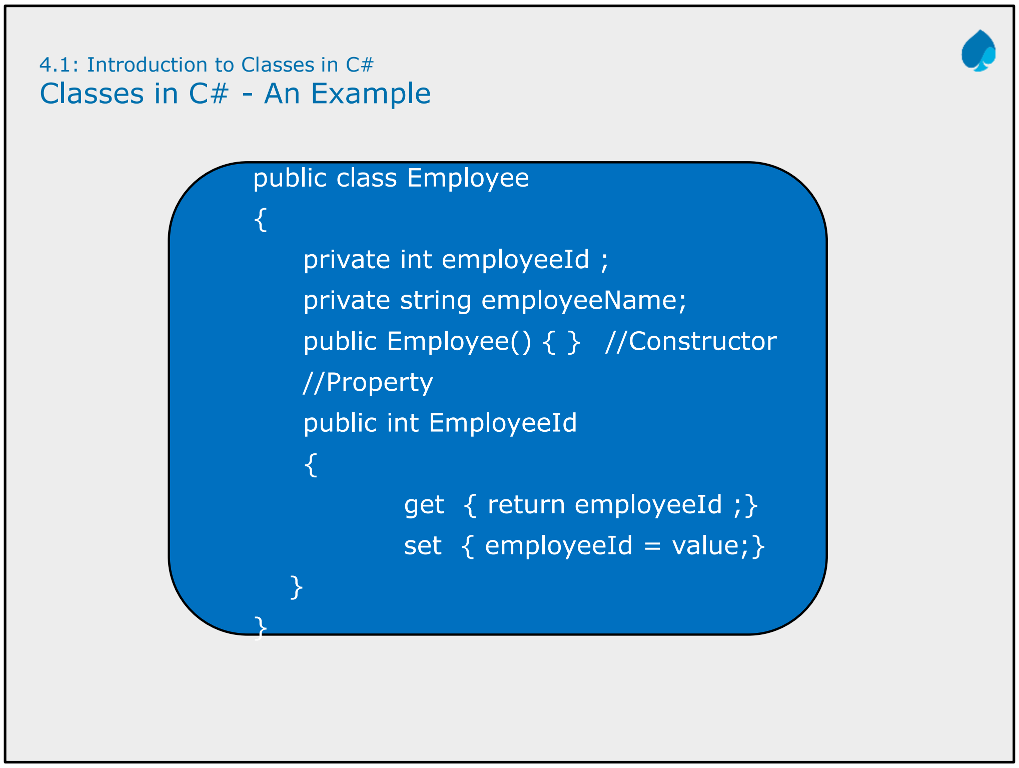
4.1: Introduction to Classes in C#

What are classes?

- A class is a user-defined type (UDT) that is composed of field data (member variables) and methods (member functions) that act on this data.
- In C#, classes can contain the following:
  - Constructors and destructors
  - Fields and constants
  - Methods
  - Properties
  - Indexers
  - Overloaded operators
  - Nested types
    - Classes & Structs
    - interfaces
    - Enumerations
    - Delegates
    - Event

Classes:

- **Template**: 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. Thus, a class is essentially a set of plans that specify how to build an object.
- **Logical Construction**: A class is a logical abstraction. Only after an object of that class has been created, a physical representation of that class exists in memory.
- **The Data and Code**: 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. In general terms, data is contained in data members defined by the class, and code is contained in function members. It is important to state at the outset that C# defines several specific types of data and function members.
- For example, data members (also called fields) include: instance variables and static variables.
- Function members include:
  - Methods
  - Constructors
  - Destructors
  - Indexers
  - Events
  - Operators
  - Properties



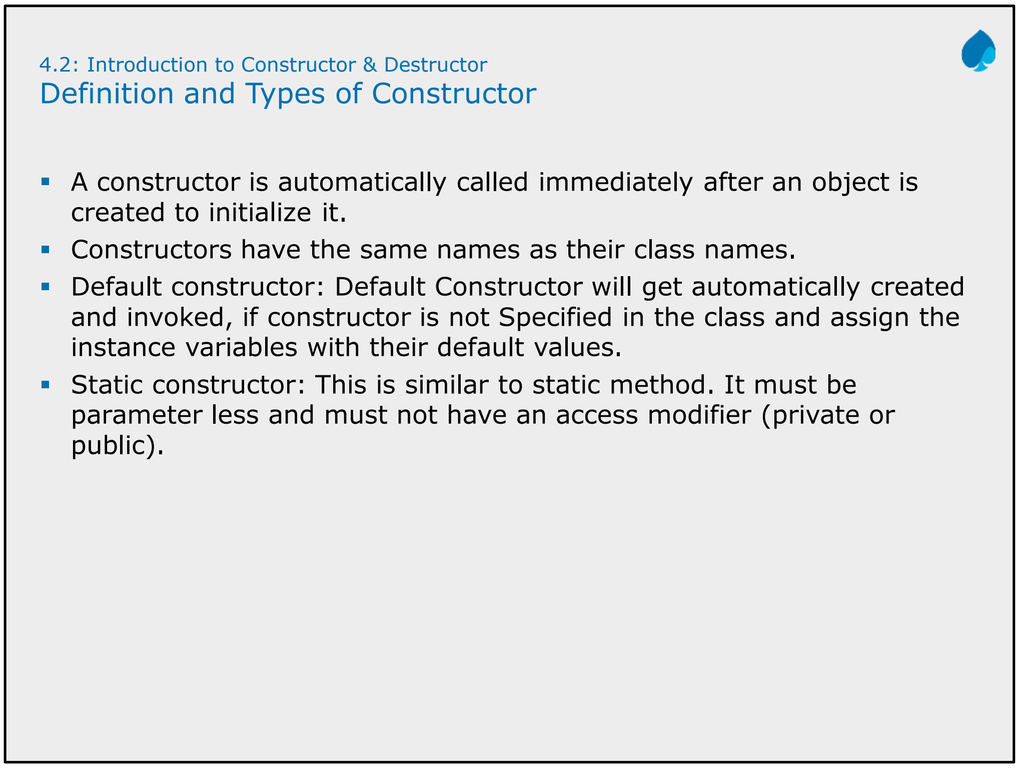
4.1: Introduction to Classes in C#

Classes in C# - An Example

```
public class Employee
{
    private int employeeId ;
    private string employeeName;
    public Employee() { }  //Constructor
    //Property
    public int EmployeeId
    {
            get  { return employeeId ;}
            set  { employeeId = value;}
    }
}
```

Classes (Contd.):

- A class is created using the keyword **class**. Given here is the general form of a simple class definition that contains only instance variables and methods.
- Notice that each declaration of variable or method is preceded with access. Here, **access** is an access specifier, such as **public**, which specifies how the member can be accessed. The access specifier determines what type of access is allowed. The access specifier is optional and if absent, then the member is private to the class. Members with private access can be used only by other members of their class.



4.2: Introduction to Constructor & Destructor

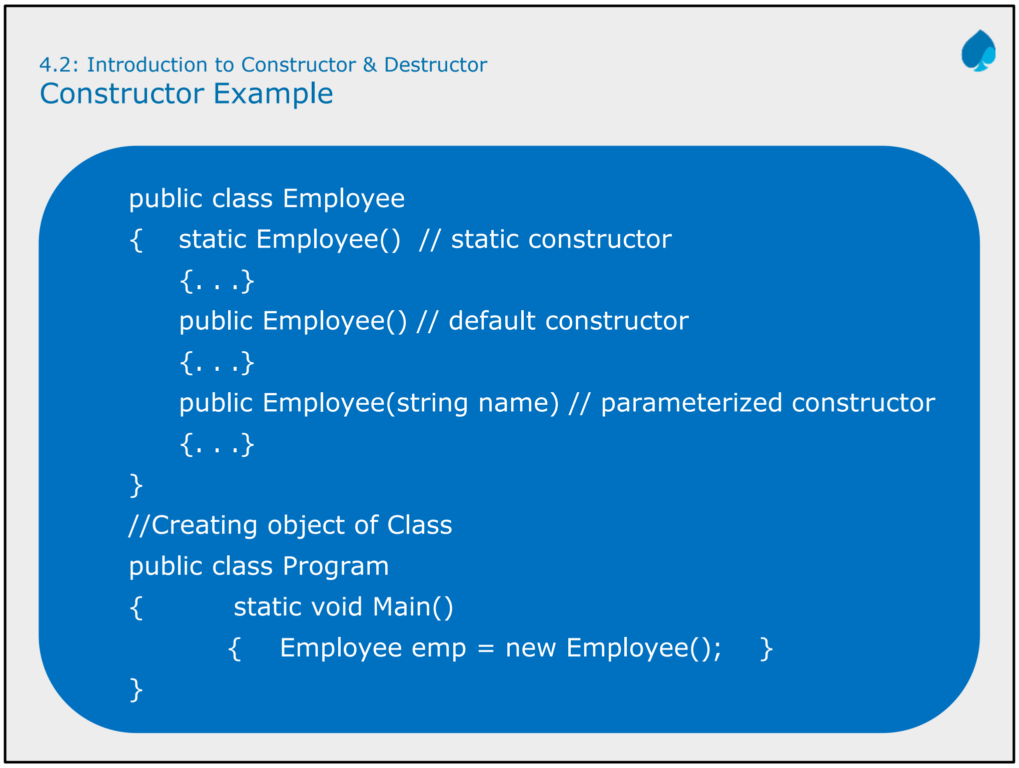
Definition and Types of Constructor

- A constructor is automatically called immediately after an object is created to initialize it.
- Constructors have the same names as their class names.
- Default constructor: Default Constructor will get automatically created and invoked, if constructor is not Specified in the class and assign the instance variables with their default values.
- Static constructor: This is similar to static method. It must be parameter less and must not have an access modifier (private or public).

<u>Constructors:</u>
A constructor initializes an object when it is created. Constructor has the same name as its class and is syntactically similar to a method. However, constructor has no explicit return type. The general form of constructor is shown here:

access class-name( ) { // constructor code }

Typically, you use a constructor to give initial values to the instance variables defined by the class. Also, you use it to perform any other startup procedures required to create a fully formed object. Usually, access is **public** because constructors are normally called from outside their class.
All classes have constructors, whether you define one or not. The reason is, C# automatically provides a default constructor that initializes all member variables to **zero** (for value types) or **null** (for reference types). However, once you define your own constructor, the default constructor is no longer used.



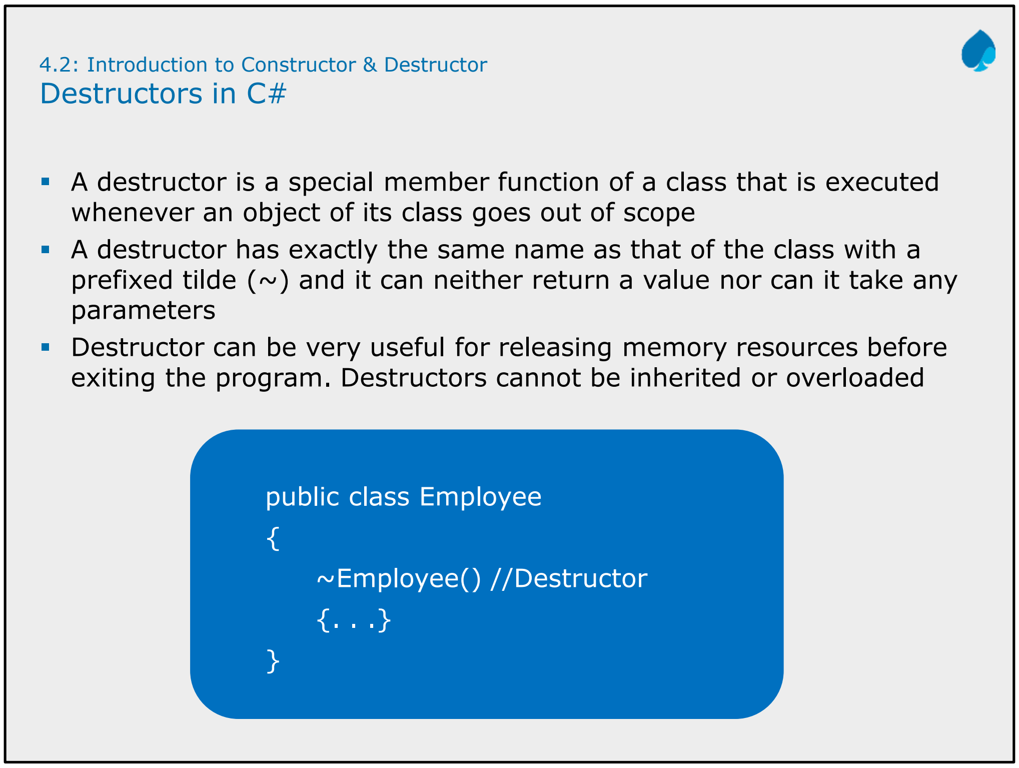
4.2: Introduction to Constructor & Destructor

Constructor Example

```
public class Employee
{    static Employee()  // static constructor
     {. . .}
     public Employee() // default constructor
     {. . .}
     public Employee(string name) // parameterized constructor
     {. . .}
}
//Creating object of Class
public class Program
{          static void Main()
          {    Employee emp = new Employee();    }
}
```

<u>Example of a Constructor:</u>

Static constructor is called only once before any object of that class gets instantiated.

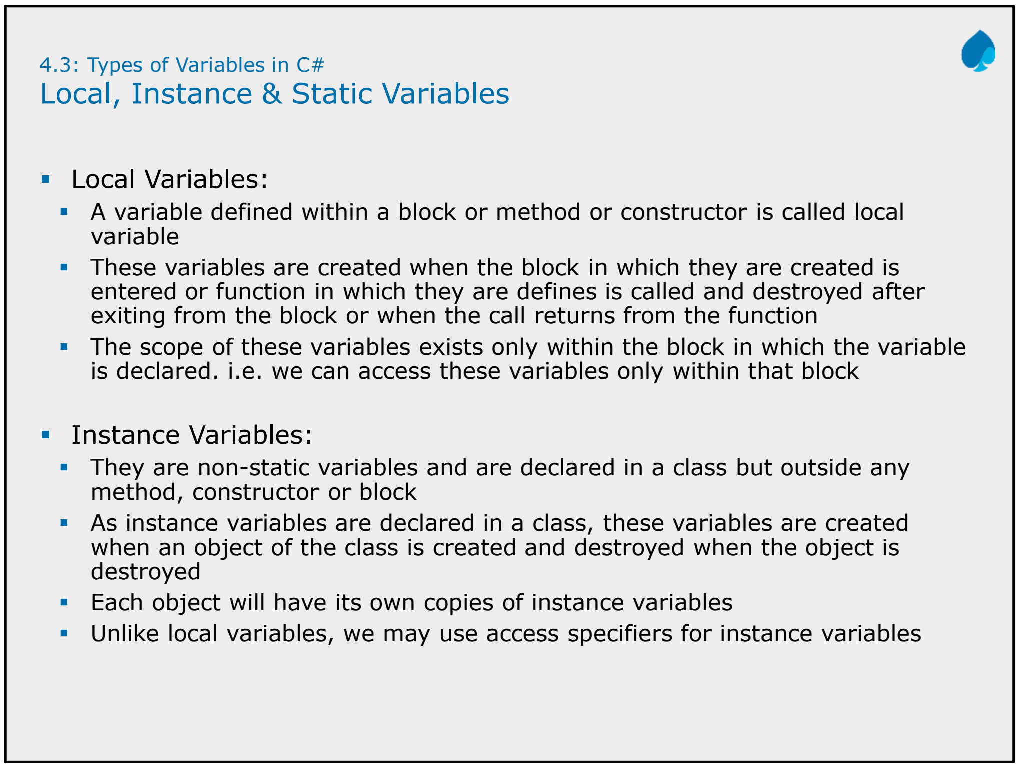


4.2: Introduction to Constructor & Destructor

Destructors in C#

- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope
- A destructor has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters
- Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded

```
public class Employee
{
    ~Employee() //Destructor
    {. . .}
}
```



4.3: Types of Variables in C#

Local, Instance & Static Variables

- Local Variables:
  - A variable defined within a block or method or constructor is called local variable
  - These variables are created when the block in which they are created is entered or function in which they are defines is called and destroyed after exiting from the block or when the call returns from the function
  - The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variables only within that block
- Instance Variables:
  - They are non-static variables and are declared in a class but outside any method, constructor or block
  - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed
  - Each object will have its own copies of instance variables
  - Unlike local variables, we may use access specifiers for instance variables

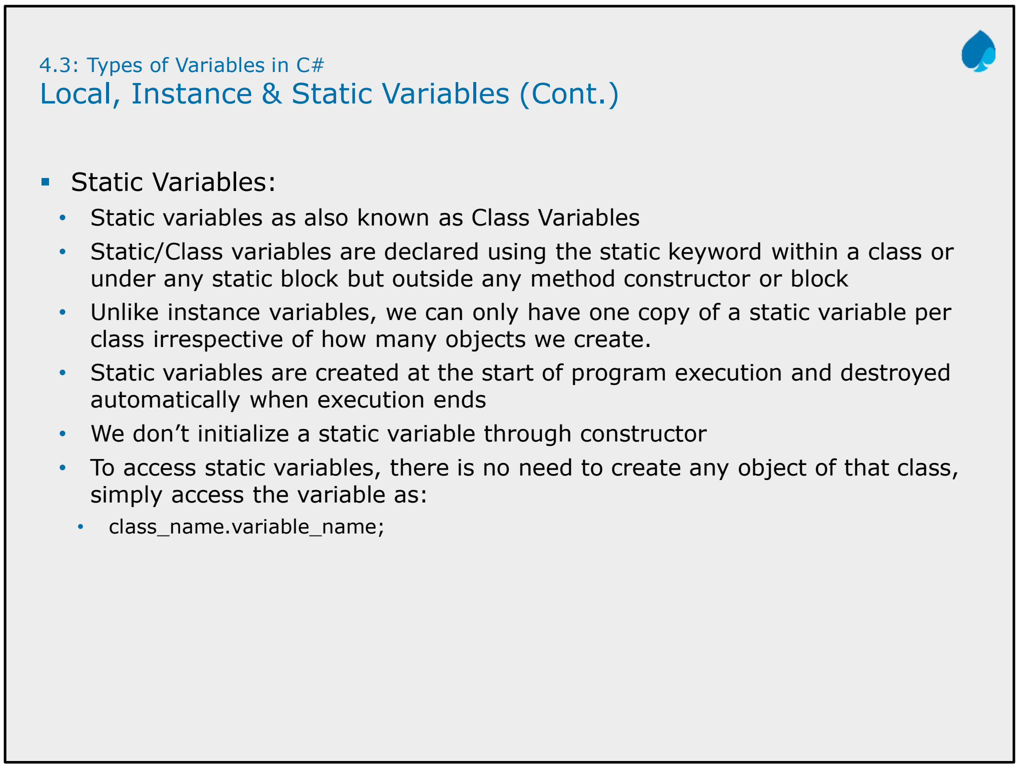
Example : Local Variables

```
class EmployeeDetails
{

    public void DisplayAge()
    {
        //local variable age
        int age = 0;

        age = age + 35;
        Console.WriteLine("Employee age is : " + age);
    }

    public static void Main(String[] args)
    {
        EmployeeDetails obj = new EmployeeDetails();
        obj.DisplayAge();
    }
}
```

Output :

```
Employee age is : 35
```

In the above program, the variable “age” is a local variable to the function DisplayAge(). If we use the variable age outside DisplayAge() function, the compiler will produce an error as shown in below program.



4.3: Types of Variables in C#

Local, Instance & Static Variables (Cont.)

- Static Variables:
  - Static variables as also known as Class Variables
  - Static/Class variables are declared using the static keyword within a class or under any static block but outside any method constructor or block
  - Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
  - Static variables are created at the start of program execution and destroyed automatically when execution ends
  - We don't initialize a static variable through constructor
  - To access static variables, there is no need to create any object of that class, simply access the variable as:
    - class_name.variable_name;

Example : Instance Variable

```
class OrderDetails{

    //Instance Variables
    int firstOrderValue;
    int secondOrderValue;
    int thirdOrderValue;

    // Main Method
    public static void Main(String[] args)
    {

        //First object
        OrderDetails obj1 = new OrderDetails();
        obj1.firstOrderValue = 2000;
        obj1.secondOrderValue = 1500;
        obj1.thirdOrderValue = 9000;

        //Second object
        OrderDetails obj2 = new OrderDetails();
        obj2.firstOrderValue = 5000;
        obj2.secondOrderValue = 2500;
        obj2.thirdOrderValue = 1000;
```

4.4: Introduction to Methods in C#

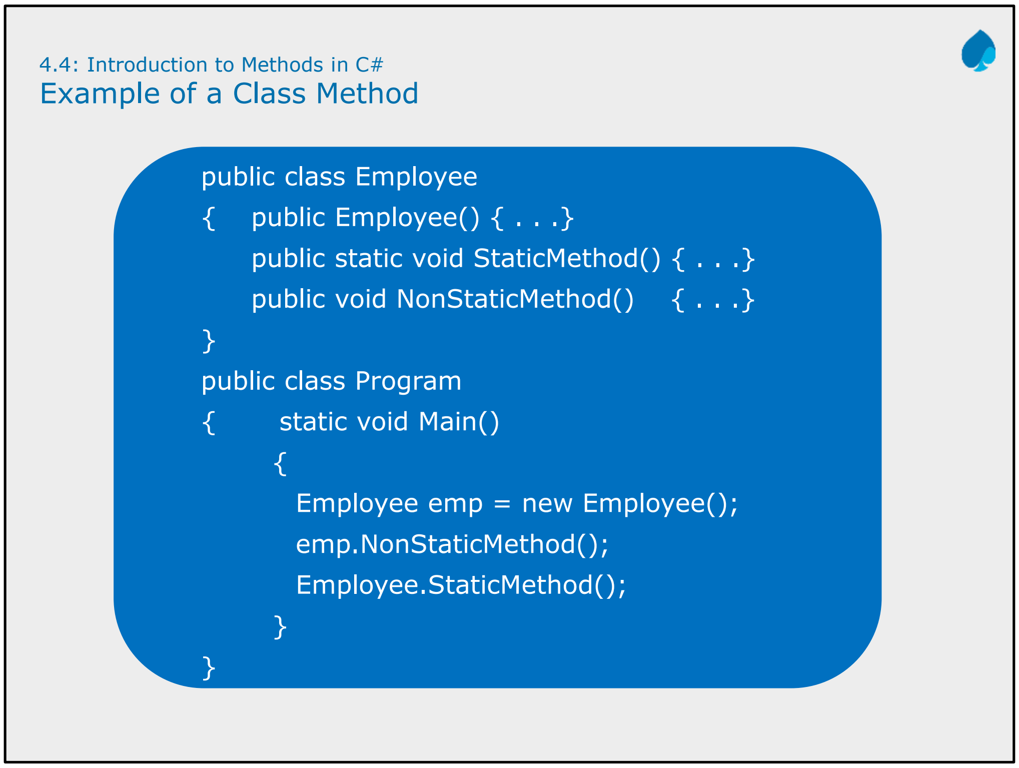
Definition of Method

- A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using the following method-declaration:

```
[attributes]
[method-modifiers] return-type method-name-
identifier ( [formal-parameter-list] )
{
    [statements]
}
```

- There are four kinds of parameters:
- out
- ref
- params
- value

value is a default parameter



4.4: Introduction to Methods in C#

Example of a Class Method

```
public class Employee
{    public Employee() { . . .}
     public static void StaticMethod() { . . .}
     public void NonStaticMethod()   { . . .}
}
public class Program
{       static void Main()
        {
          Employee emp = new Employee();
          emp.NonStaticMethod();
          Employee.StaticMethod();
        }
}
```

Class Method - Example :

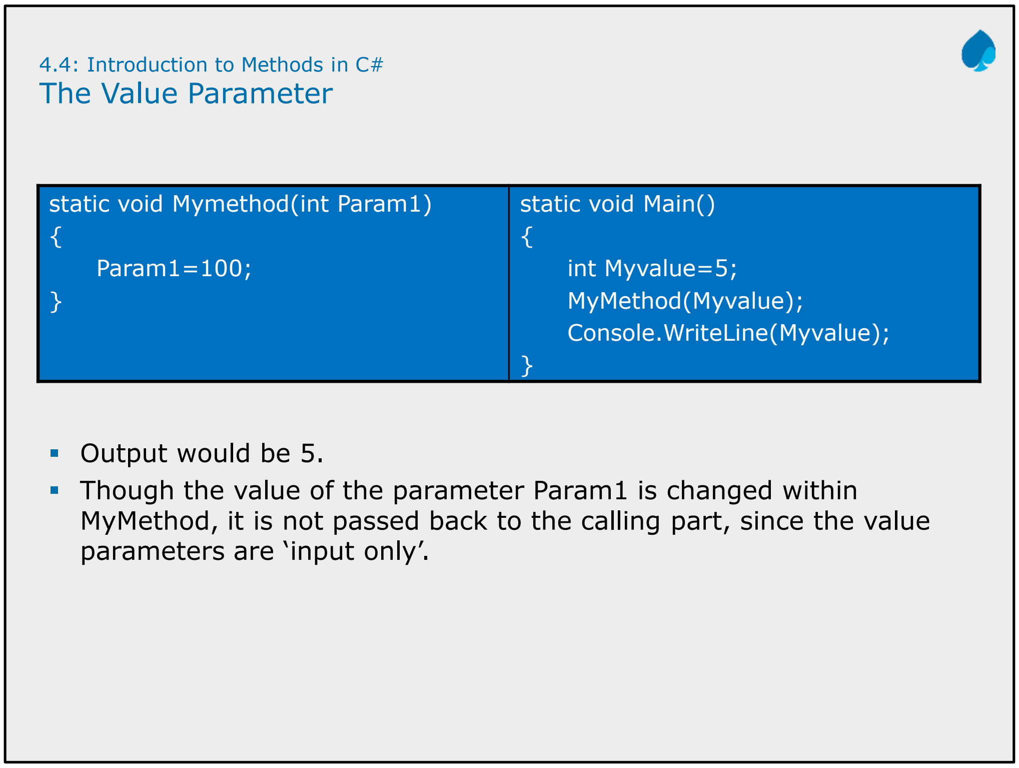
Methods are subroutines that manipulate the data defined by a class; and, in many cases, provide access to that data. Typically, other parts of your program interacts with a class through its methods.

The general form of a method is shown here:

```
access ret-type name(parameter-list)
 {
 // body of method
 }
```

A method contains one or more statements. In a well-written C# code, each method performs only one task.

Each method has a name, and this name is used for calling the method.



4.4: Introduction to Methods in C#

The Value Parameter

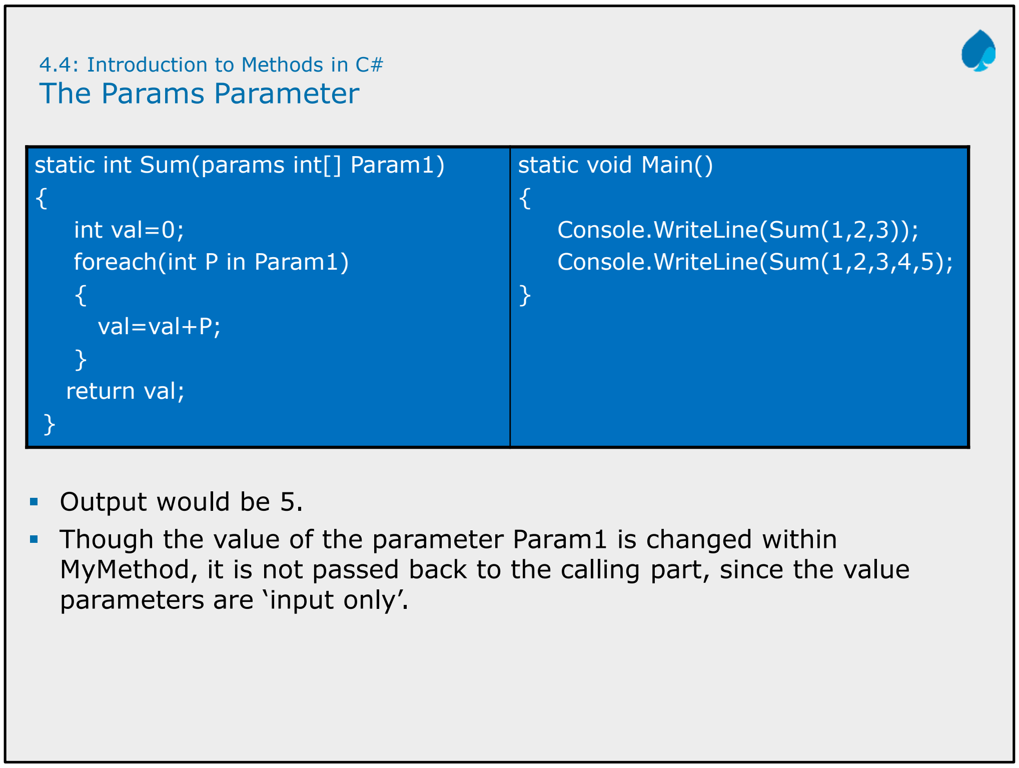
```
static void Mymethod(int Param1)
{
    Param1=100;
}
```

```
static void Main()
{
    int Myvalue=5;
    MyMethod(Myvalue);
    Console.WriteLine(Myvalue);
}
```

- Output would be 5.
- Though the value of the parameter Param1 is changed within MyMethod, it is not passed back to the calling part, since the value parameters are 'input only'.

The Value Parameters:

The **value** parameter is the default parameter type in C#.
If a parameter does not have any modifier, it is the **value** parameter by default.
When you use the **value** parameter, the actual value is passed to the function.
This means, changes made to the parameter are local to the function and are not passed back to the calling part.



4.4: Introduction to Methods in C#

The Params Parameter

```
static int Sum(params int[] Param1)
{
     int val=0;
     foreach(int P in Param1)
     {
        val=val+P;
     }
    return val;
 }
```

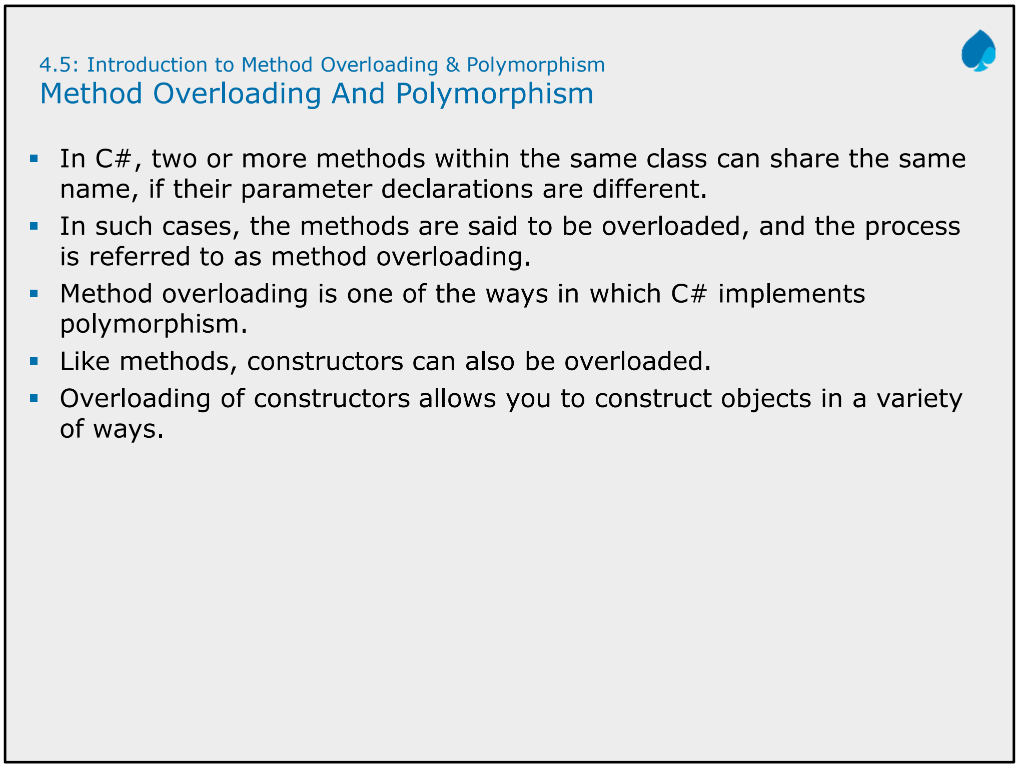
```
static void Main()
{
     Console.WriteLine(Sum(1,2,3));
     Console.WriteLine(Sum(1,2,3,4,5);
}
```

- Output would be 5.
- Though the value of the parameter Param1 is changed within MyMethod, it is not passed back to the calling part, since the value parameters are ‘input only’.

The Params Parameter:

The value passed for a **params** parameter can be either a comma-separated value list or a single dimensional array.

The **params** parameters are ‘input only’.



4.5: Introduction to Method Overloading & Polymorphism

Method Overloading And Polymorphism

- In C#, two or more methods within the same class can share the same name, if their parameter declarations are different.
- In such cases, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways in which C# implements polymorphism.
- Like methods, constructors can also be overloaded.
- Overloading of constructors allows you to construct objects in a variety of ways.

Method Overloading and Polymorphism:

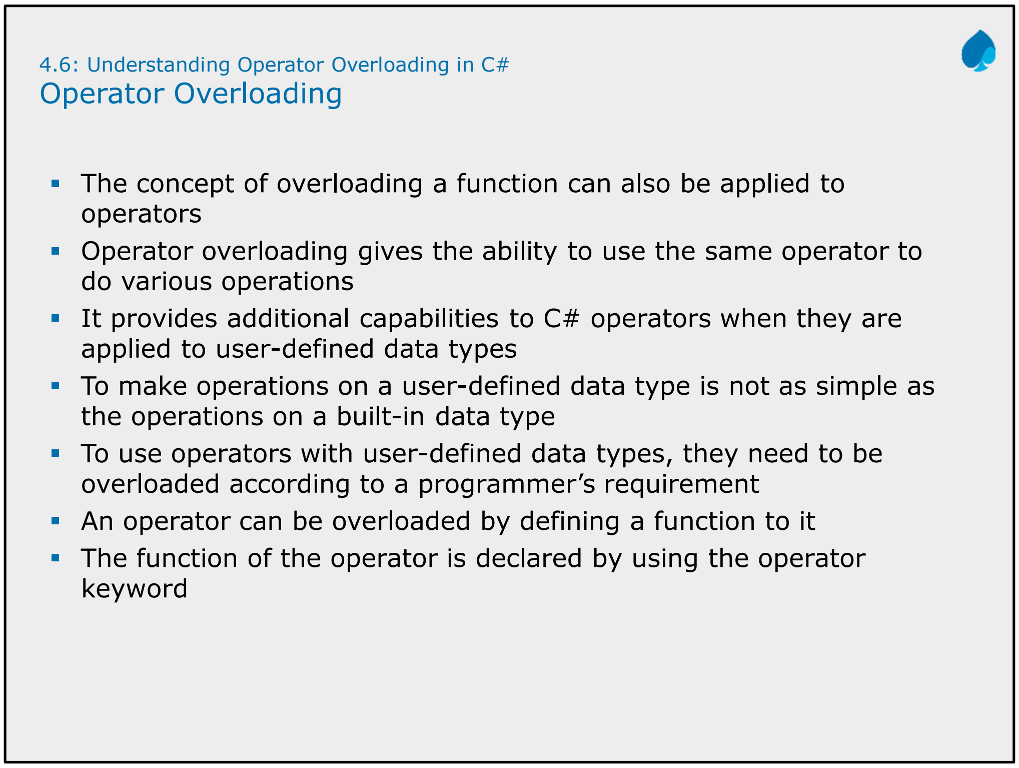
In general, to overload a method, we declare different versions of it. The compiler takes care of the rest.

We must observe one important restriction: the type and/or number of the parameters of each overloaded method must be unique.

It is not sufficient for two methods to differ only in their return types. They must differ in the types or numbers of their parameters. (Return types do not provide sufficient information in all cases for C# to decide which method to use.)

Of course, overloaded methods may differ in their return types also.

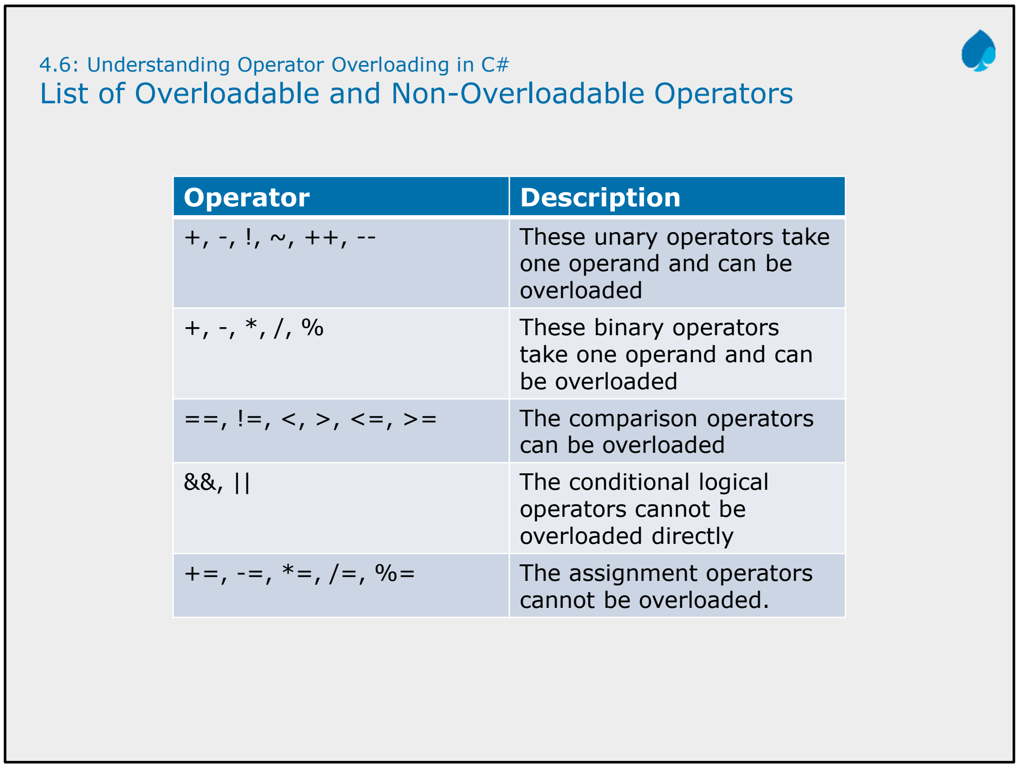
When an overloaded method is called, the version of the method executed is the one which has parameters that match the arguments.



4.6: Understanding Operator Overloading in C#

Operator Overloading

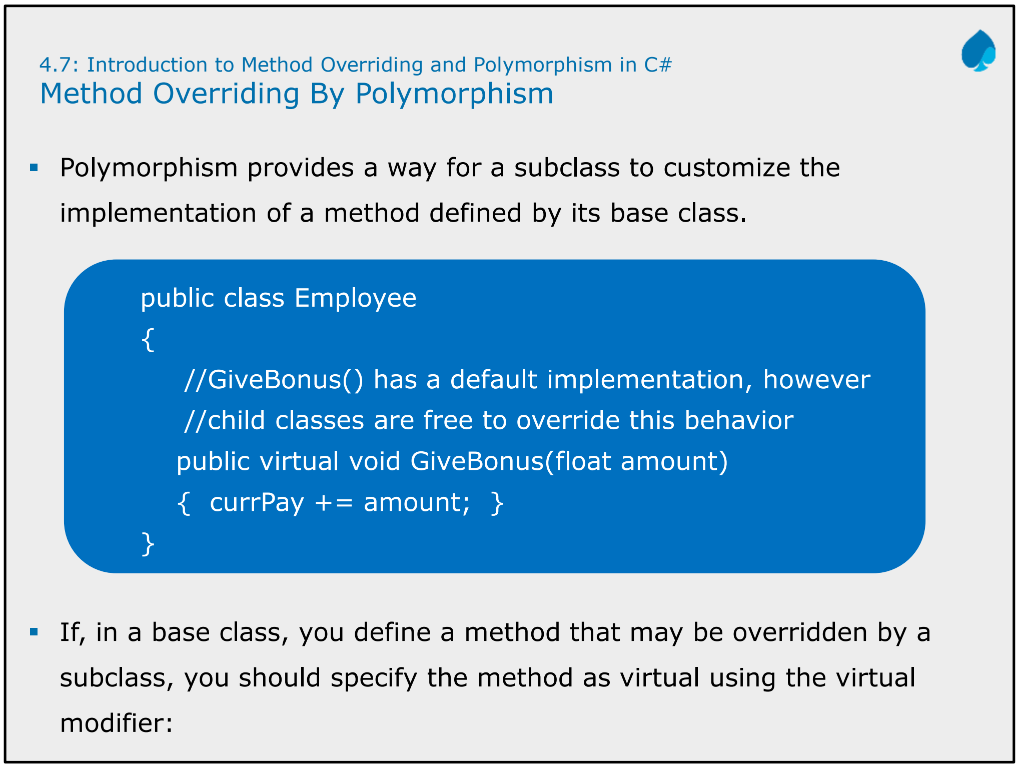
- The concept of overloading a function can also be applied to operators
- Operator overloading gives the ability to use the same operator to do various operations
- It provides additional capabilities to C# operators when they are applied to user-defined data types
- To make operations on a user-defined data type is not as simple as the operations on a built-in data type
- To use operators with user-defined data types, they need to be overloaded according to a programmer’s requirement
- An operator can be overloaded by defining a function to it
- The function of the operator is declared by using the operator keyword



4.6: Understanding Operator Overloading in C#

List of Overloadable and Non-Overloadable Operators

Operator	Description
+, -, !, ~, ++, --	These unary operators take one operand and can be overloaded
+, -, *, /, %	These binary operators take one operand and can be overloaded
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators cannot be overloaded directly
+=, -=, *=, /=, %=	The assignment operators cannot be overloaded.



4.7: Introduction to Method Overriding and Polymorphism in C#

Method Overriding By Polymorphism

- Polymorphism provides a way for a subclass to customize the implementation of a method defined by its base class.

```
public class Employee
{
    //GiveBonus() has a default implementation, however
    //child classes are free to override this behavior
   public virtual void GiveBonus(float amount)
   {  currPay += amount;  }
}
```

- If, in a base class, you define a method that may be overridden by a subclass, you should specify the method as virtual using the virtual modifier:

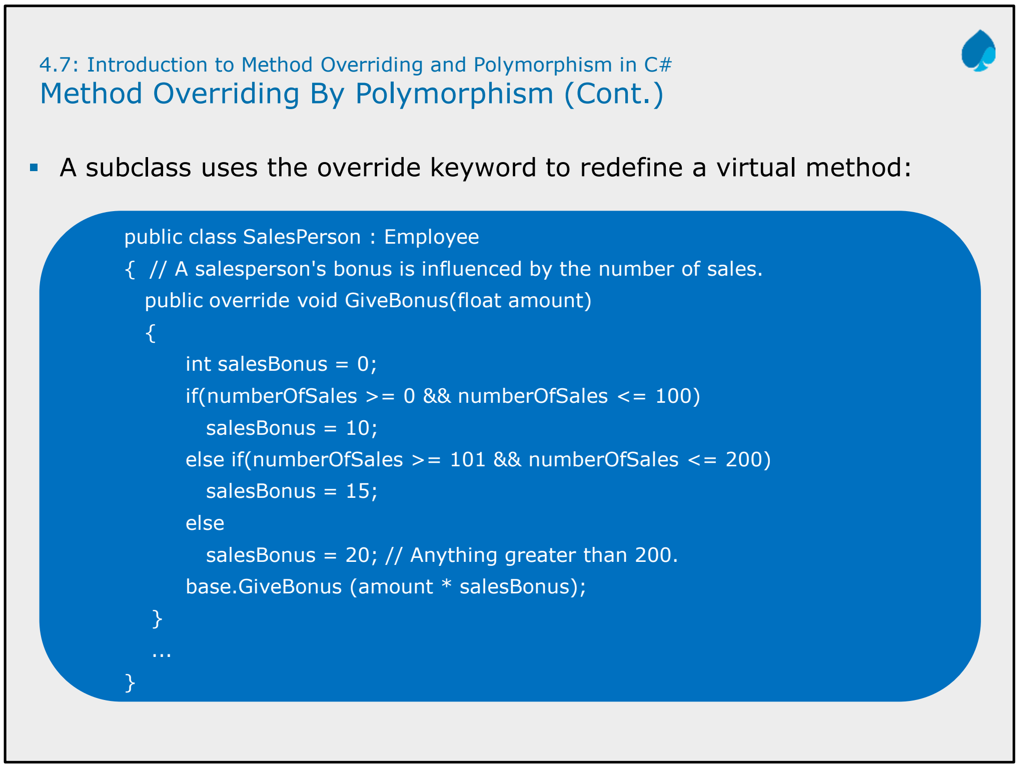
4.7: Introduction to Method Overriding and Polymorphism in C#

Method Overriding By Polymorphism (Cont.)

- A **virtual** method is a method that is declared as virtual in a base class and redefined in one or more derived classes.
- Each derived class can have its own version of a virtual method.
- You declare a method as virtual inside a base class by preceding its declaration with the keyword virtual.
- When a virtual method is redefined by a derived class, the **override** modifier is used.

Virtual Methods -Declaration and Redefinition:

Virtual methods are interesting because of what happens 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 types of objects are referred to, different versions of the virtual method are executed.



4.7: Introduction to Method Overriding and Polymorphism in C#

Method Overriding By Polymorphism (Cont.)

- A subclass uses the override keyword to redefine a virtual method:

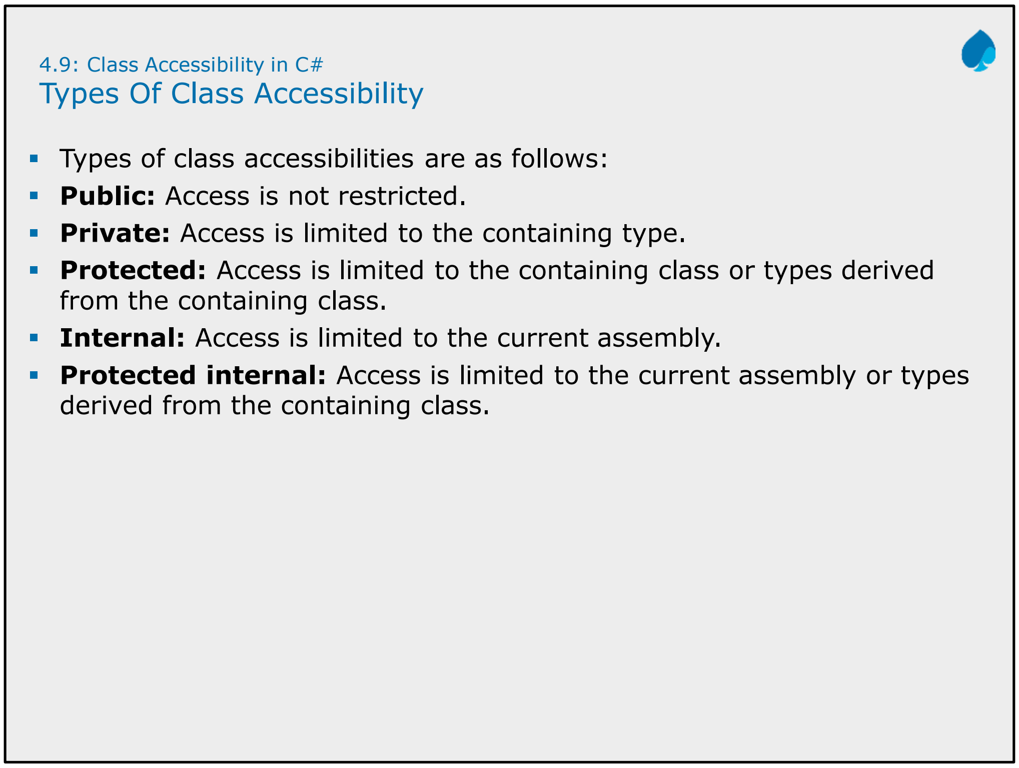
```
public class SalesPerson : Employee
{  // A salesperson's bonus is influenced by the number of sales.
   public override void GiveBonus(float amount)
   {
         int salesBonus = 0;
         if(numberOfSales >= 0 && numberOfSales <= 100)
            salesBonus = 10;
         else if(numberOfSales >= 101 && numberOfSales <= 200)
            salesBonus = 15;
         else
            salesBonus = 20; // Anything greater than 200.
         base.GiveBonus (amount * salesBonus);
    }
    …
}
```

4.8: Introduction to Static Members in C#

What Is A Static Constructor?

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

What is a static constructor?:
A static constructor is called automatically; it is called before the instance constructor is called.
In all the cases, the static constructor is executed before any instance constructor.
Furthermore, static constructors cannot have access modifiers (thus, they use default access) and cannot be called by your program.



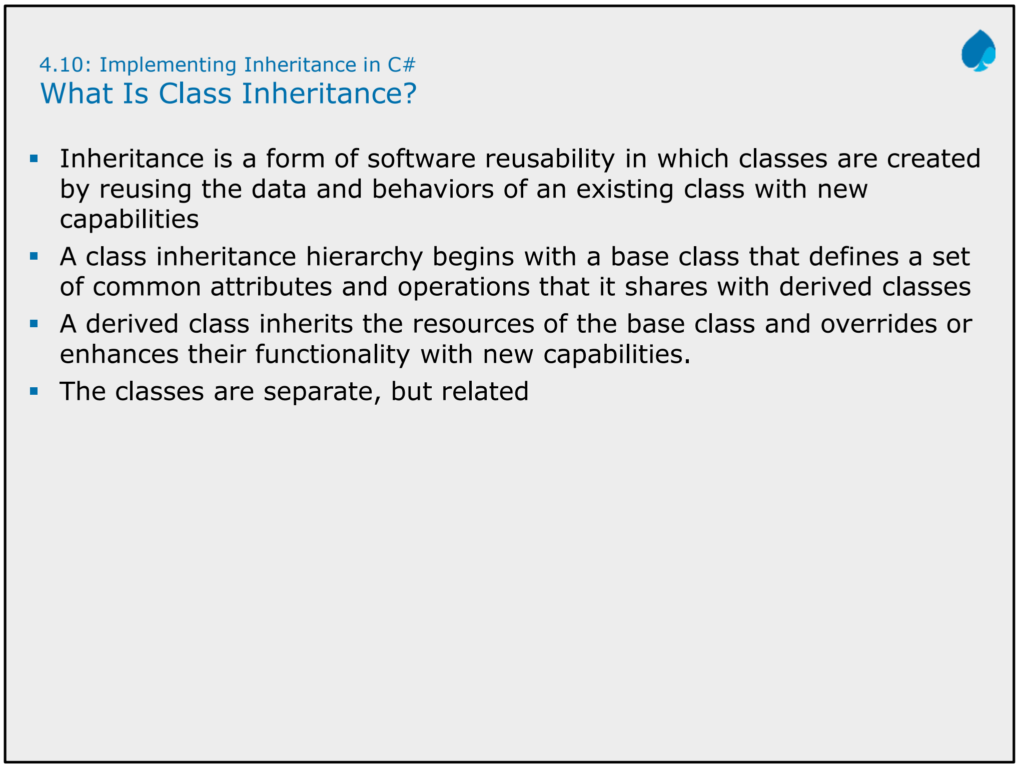
4.9: Class Accessibility in C#

Types Of Class Accessibility

- Types of class accessibilities are as follows:
- **Public:** Access is not restricted.
- **Private:** Access is limited to the containing type.
- **Protected:** Access is limited to the containing class or types derived from the containing class.
- **Internal:** Access is limited to the current assembly.
- **Protected internal:** Access is limited to the current assembly or types derived from the containing class.

Demo

- Defining and using Classes in C#



4.10: Implementing Inheritance in C#

What Is Class Inheritance?

- Inheritance is a form of software reusability in which classes are created by reusing the data and behaviors of an existing class with new capabilities
- A class inheritance hierarchy begins with a base class that defines a set of common attributes and operations that it shares with derived classes
- A derived class inherits the resources of the base class and overrides or enhances their functionality with new capabilities.
- The classes are separate, but related

What is class inheritance?

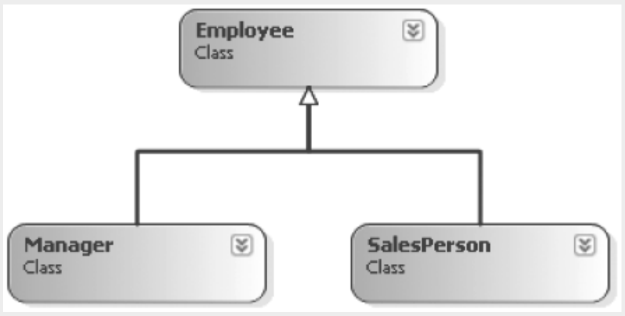
Inheritance is one of the three foundational principles of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding the things that are
unique to it.

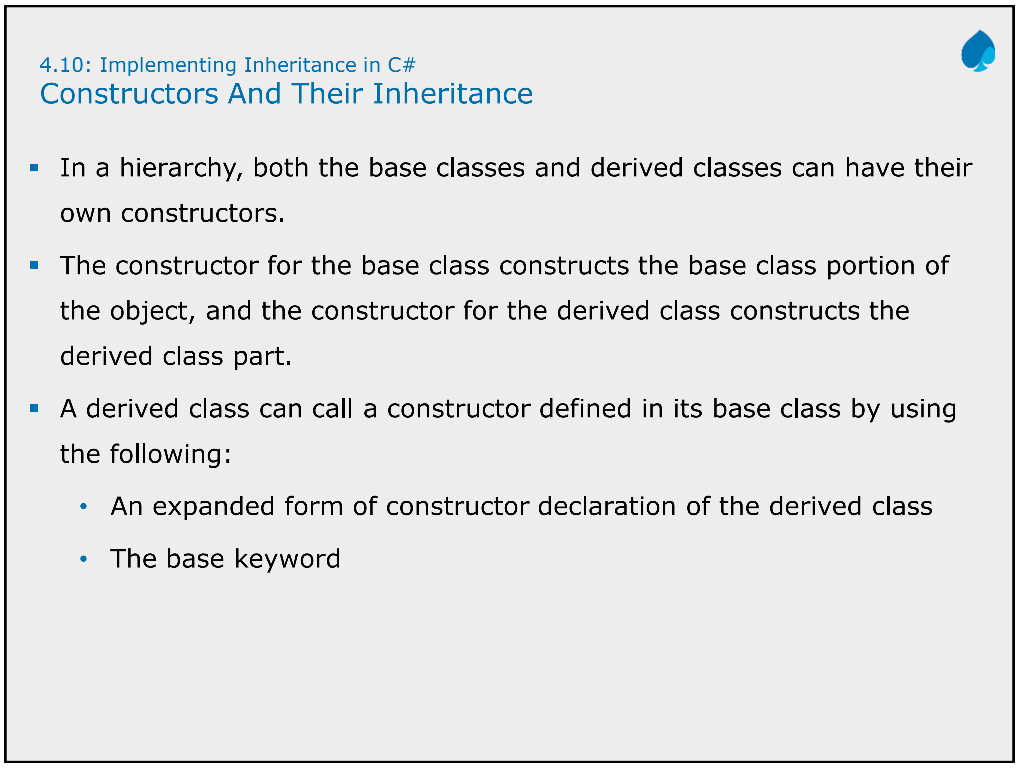
In the language of C#, a class that is inherited is called a base class. The class that performs the inheriting is called a derived class. Therefore, a derived class is a specialized version of a base class. It inherits all of the variables, methods, properties, operators, and indexers defined by the base class and adds its own unique elements.

4.10: Implementing Inheritance in C#

What Is Class Inheritance? (Cont.)

- Inheritance is also called 'is a' relationship
- A SalesPerson 'is-a' Employee (as is a Manager)
- Base classes (such as Employee) are used to define general characteristics that are common to all descendants.
- Derived classes (such as SalesPerson and Manager), the general functionalities, are extended while adding more specific behaviors.





4.10: Implementing Inheritance in C#

Constructors And Their Inheritance

- In a hierarchy, both the base classes and derived classes can have their own constructors.
- The constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part.
- A derived class can call a constructor defined in its base class by using the following:
  - An expanded form of constructor declaration of the derived class
  - The base keyword

Constructors and Their Inheritance:
In a hierarchy, both the base classes and derived classes can have their own constructors. This raises an important question: which constructor is responsible for building an object of the derived class? The one in the base class, the one in the derived class, or both? The answer is: the constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part. The reason is, the base class has no knowledge of or access to any element in a derived class. Thus, their construction is separate.

A derived class can call a constructor defined in its base class by using an expanded form of the derived class' constructor declaration and the base keyword. The general form of this expanded declaration is as follows:

```
derived-constructor(parameter-list) : base(arg-list) {
 // body of constructor
}
```

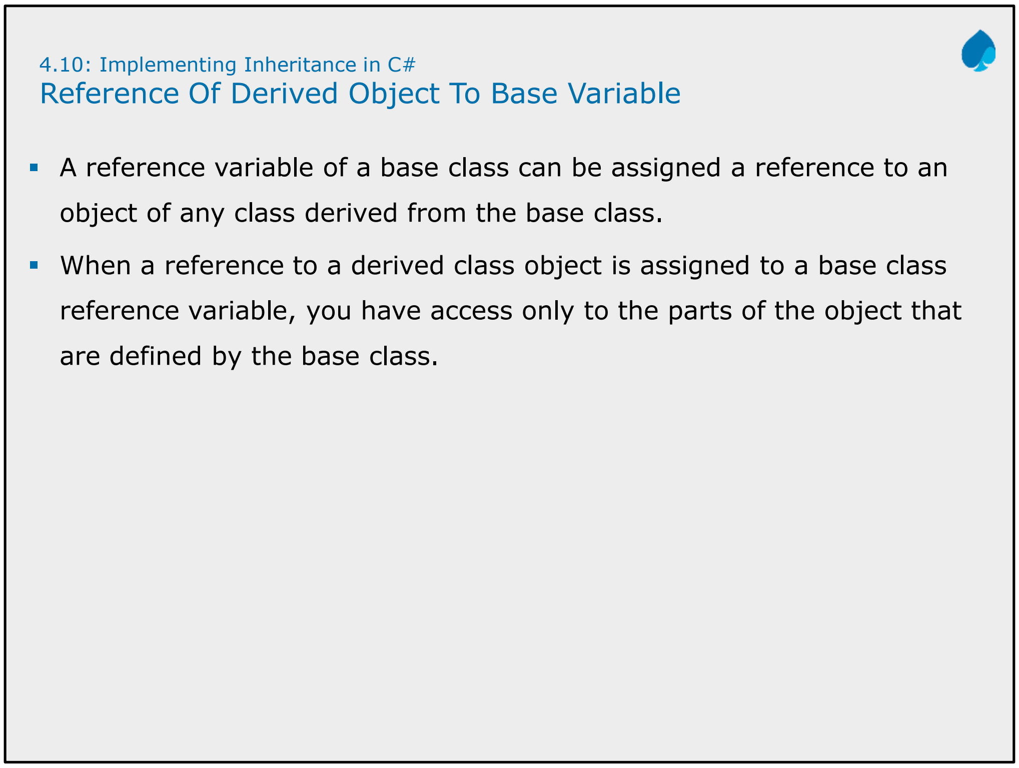
Here, arg-list specifies the arguments that are needed by the constructor defined in the base class.

4.10: Implementing Inheritance in C#

Hiding Name Of Base Class Member

- It is possible for a derived class to define a member that has the same name as a member in its base class.
- When this happens, the member in the base class is hidden within the derived class.
- Even though this is not technically an error in C#, the compiler issues a warning message.
- If you intended to hide a base class member purposely, then to prevent this warning, the derived class member must be preceded by the new keyword.

Hiding the Name of Base Class Member:
Understand that this use of the **new** keyword is separate and distinct from its use while creating an object instance.



4.10: Implementing Inheritance in C#

Reference Of Derived Object To Base Variable

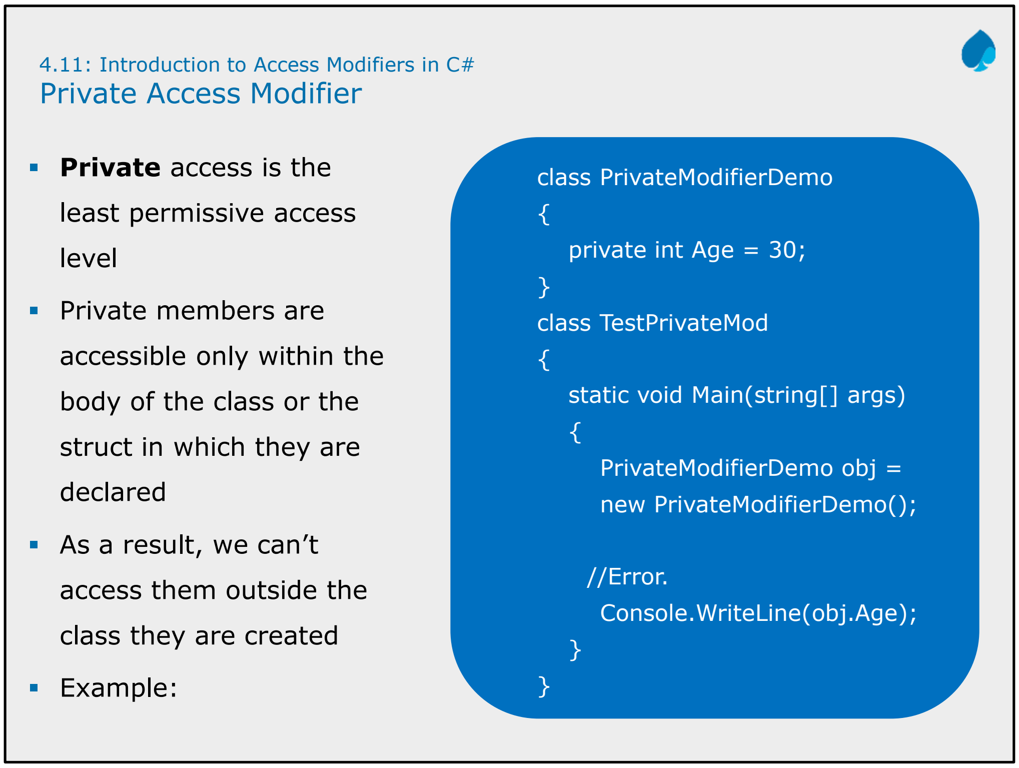
- A reference variable of a base class can be assigned a reference to an object of any class derived from the base class.
- When a reference to a derived class object is assigned to a base class reference variable, you have access only to the parts of the object that are defined by the base class.

Reference of Derived Object to Base Variable:
C# is a strongly typed language. The standard conversions and automatic promotions apply to all its value types. However, values are essentially compatible with their class types (type compatibility). Therefore, a reference variable for one class type cannot normally refer to an object of another class 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.

Demo

- Inheritance in C#

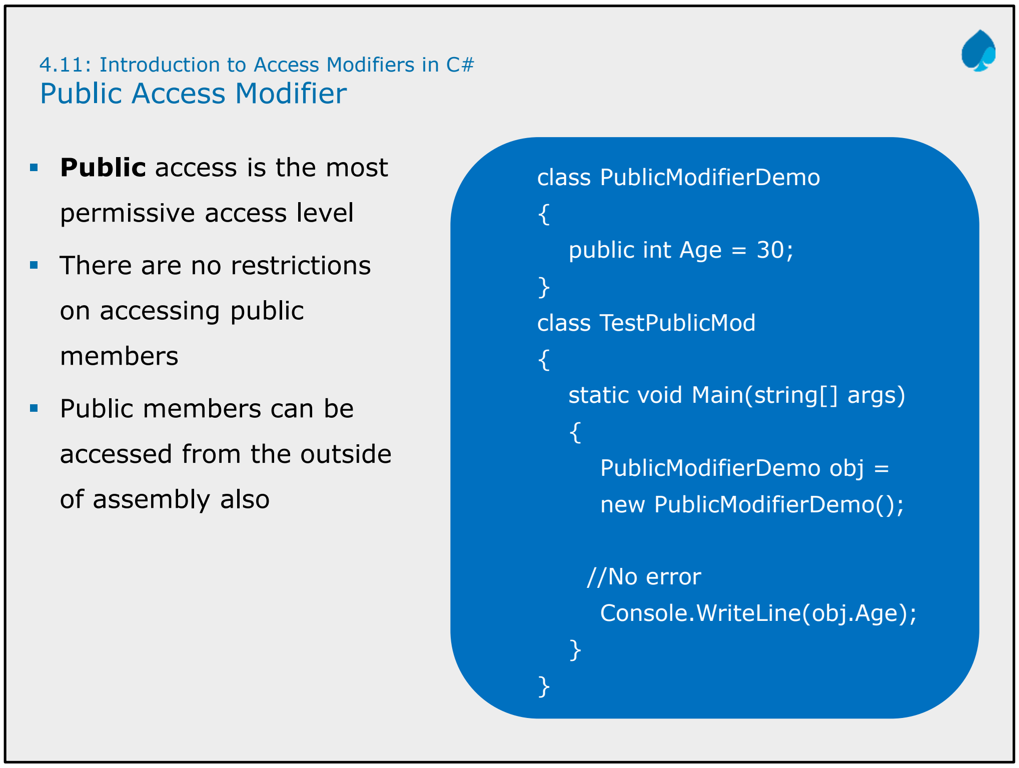


4.11: Introduction to Access Modifiers in C#

Private Access Modifier

- **Private** access is the least permissive access level
- Private members are accessible only within the body of the class or the struct in which they are declared
- As a result, we can't access them outside the class they are created
- Example:

```
class PrivateModifierDemo
{
    private int Age = 30;
}
class TestPrivateMod
{
    static void Main(string[] args)
    {
        PrivateModifierDemo obj =
        new PrivateModifierDemo();

      //Error.
        Console.WriteLine(obj.Age);
    }
}
```

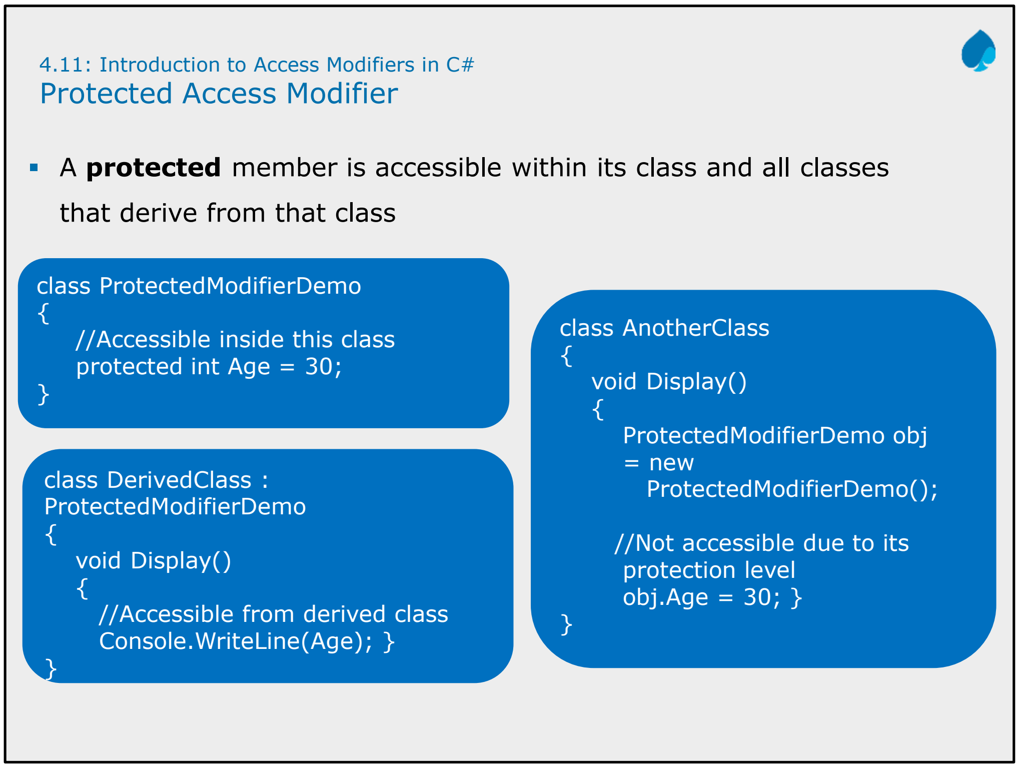


4.11: Introduction to Access Modifiers in C#

Public Access Modifier

- **Public** access is the most permissive access level
- There are no restrictions on accessing public members
- Public members can be accessed from the outside of assembly also

```
class PublicModifierDemo
{
    public int Age = 30;
}
class TestPublicMod
{
    static void Main(string[] args)
    {
        PublicModifierDemo obj =
        new PublicModifierDemo();

      //No error
        Console.WriteLine(obj.Age);
    }
}
```



4.11: Introduction to Access Modifiers in C#

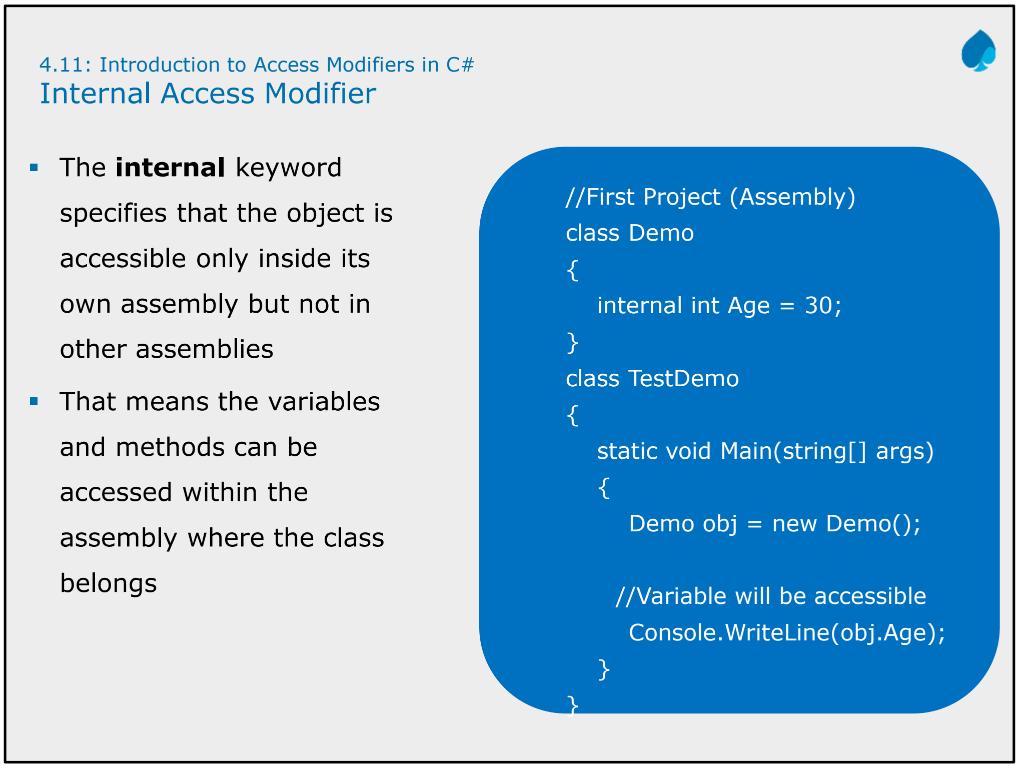
Protected Access Modifier

- A **protected** member is accessible within its class and all classes that derive from that class

```
class ProtectedModifierDemo
{
    //Accessible inside this class
    protected int Age = 30;
}
```

```
class DerivedClass :
ProtectedModifierDemo
{
    void Display()
    {
      //Accessible from derived class
      Console.WriteLine(Age); }
}
```

```
class AnotherClass
{
    void Display()
    {
        ProtectedModifierDemo obj
        = new
          ProtectedModifierDemo();

       //Not accessible due to its
        protection level
        obj.Age = 30; }
}
```

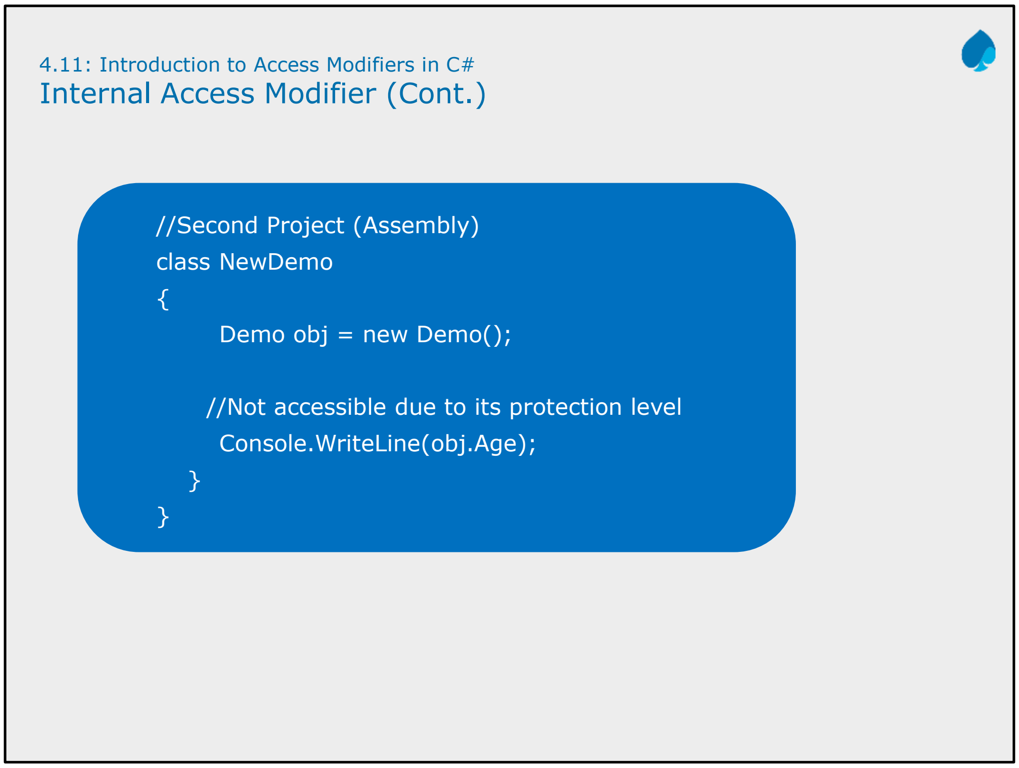


4.11: Introduction to Access Modifiers in C#

Internal Access Modifier

- The **internal** keyword specifies that the object is accessible only inside its own assembly but not in other assemblies
- That means the variables and methods can be accessed within the assembly where the class belongs

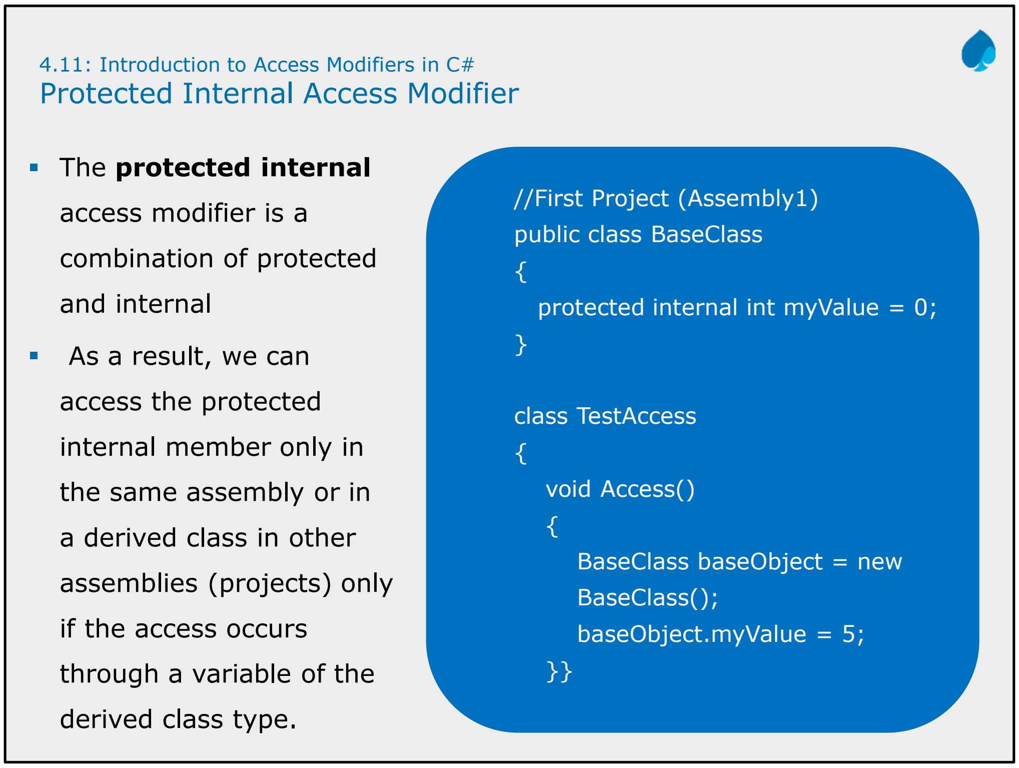
```
//First Project (Assembly)
class Demo
{
    internal int Age = 30;
}
class TestDemo
{
    static void Main(string[] args)
    {
        Demo obj = new Demo();

      //Variable will be accessible
        Console.WriteLine(obj.Age);
    }
}
```



4.11: Introduction to Access Modifiers in C#

Internal Access Modifier (Cont.)

```
//Second Project (Assembly)
class NewDemo
{
      Demo obj = new Demo();

    //Not accessible due to its protection level
      Console.WriteLine(obj.Age);
   }
}
```

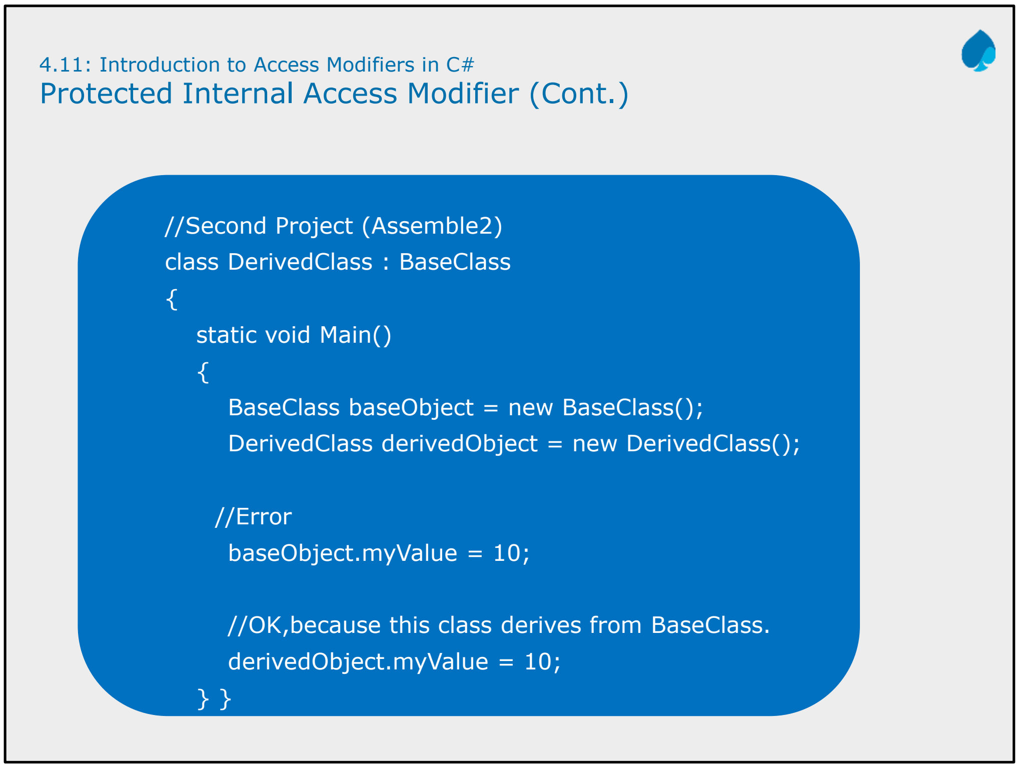


4.11: Introduction to Access Modifiers in C#

Protected Internal Access Modifier

- The **protected internal** access modifier is a combination of protected and internal
- As a result, we can access the protected internal member only in the same assembly or in a derived class in other assemblies (projects) only if the access occurs through a variable of the derived class type.

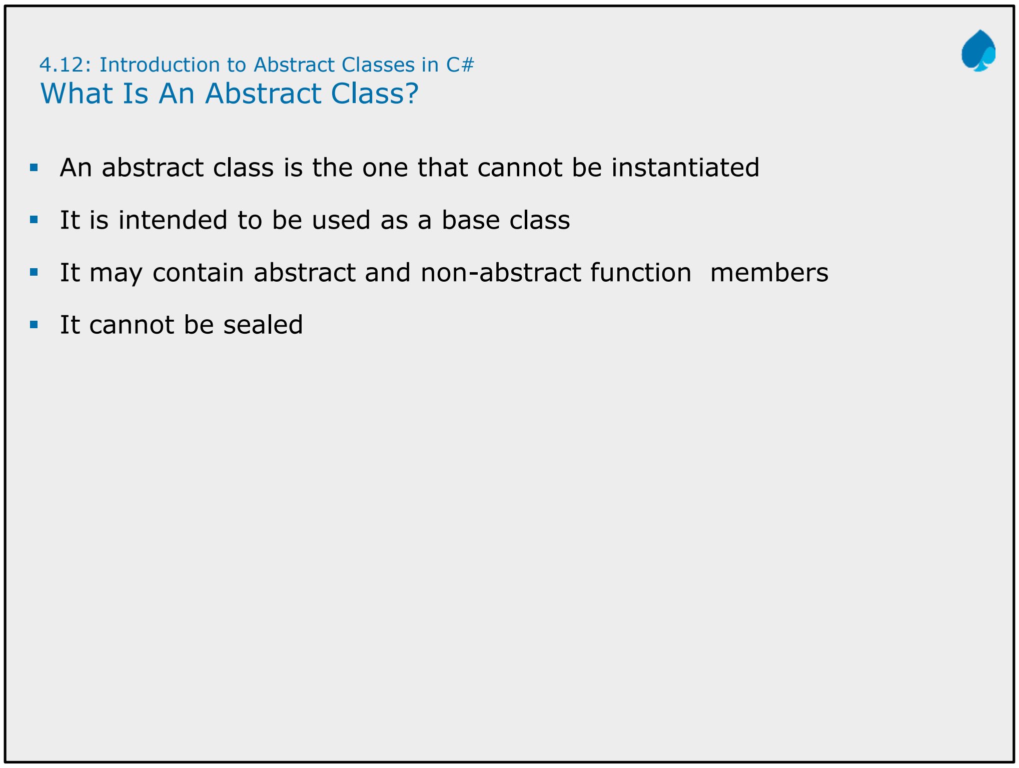
```
//First Project (Assembly1)
public class BaseClass
{
  protected internal int myValue = 0;
}

class TestAccess
{
   void Access()
   {
      BaseClass baseObject = new
      BaseClass();
      baseObject.myValue = 5;
   }}
```



4.11: Introduction to Access Modifiers in C#

Protected Internal Access Modifier (Cont.)

```
//Second Project (Assemble2)
class DerivedClass : BaseClass
{
   static void Main()
   {
      BaseClass baseObject = new BaseClass();
      DerivedClass derivedObject = new DerivedClass();

     //Error
      baseObject.myValue = 10;

      //OK,because this class derives from BaseClass.
      derivedObject.myValue = 10;
   } }
```



4.12: Introduction to Abstract Classes in C#

What Is An Abstract Class?

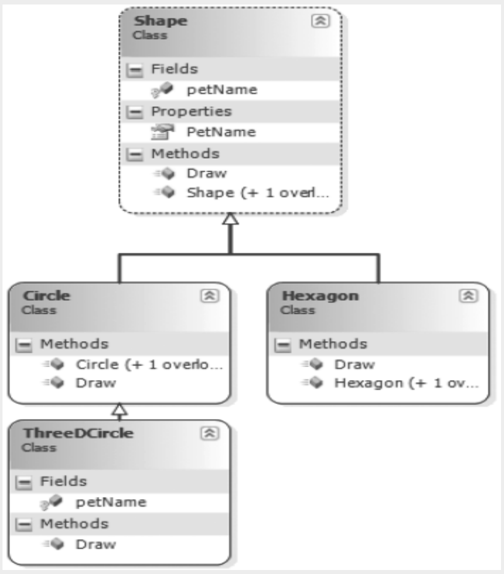
- An abstract class is the one that cannot be instantiated
- It is intended to be used as a base class
- It may contain abstract and non-abstract function members
- It cannot be sealed

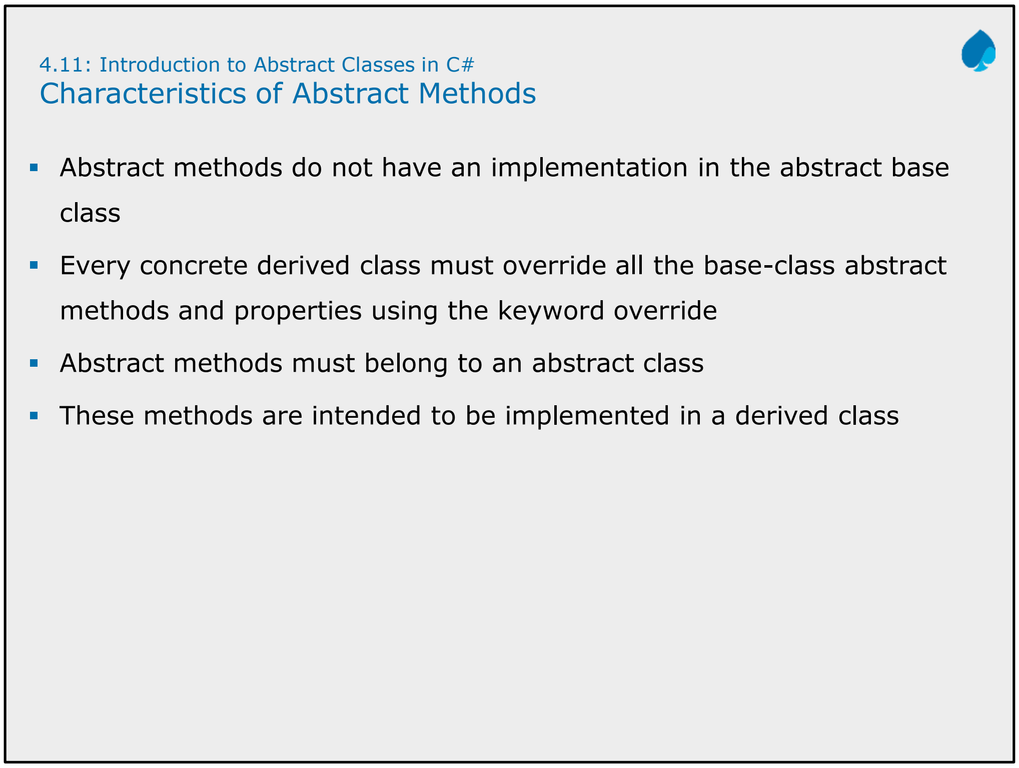
What is an abstract class?

Sometimes, you may require to create a base class that defines only a generalized form that can be shared by all of its derived classes, each derived class filling in its specific details. Such a class determines the nature of the methods that the derived classes must implement, but does not, by itself, provide an implementation of one or more of these methods. Such classes are declared as the abstract classes.

4.11: Introduction to Abstract Classes in C#

An Example : Abstract Class





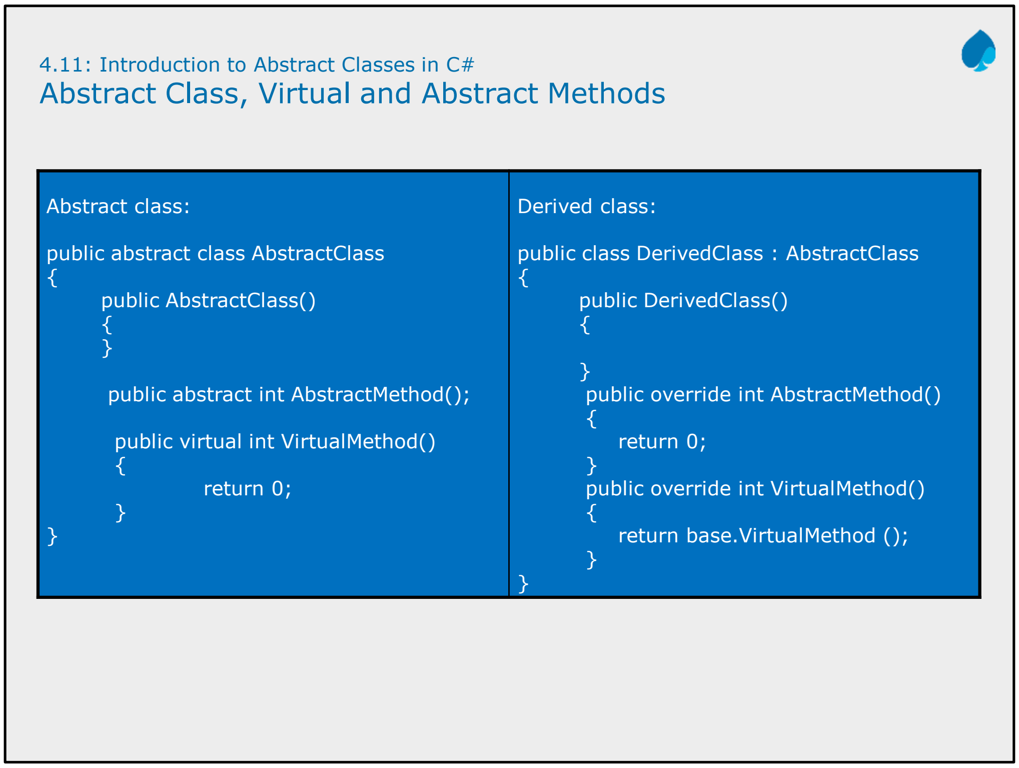
4.11: Introduction to Abstract Classes in C#

Characteristics of Abstract Methods

- Abstract methods do not have an implementation in the abstract base class
- Every concrete derived class must override all the base-class abstract methods and properties using the keyword override
- Abstract methods must belong to an abstract class
- These methods are intended to be implemented in a derived class

Characteristics of Abstract Methods:

When a class has been defined as an abstract base class, it may define any number of abstract members (which is analogous to a C++ pure virtual function). Abstract methods can be used whenever you wish to define a method that does not supply a default implementation.



4.11: Introduction to Abstract Classes in C#

Abstract Class, Virtual and Abstract Methods

Abstract class:

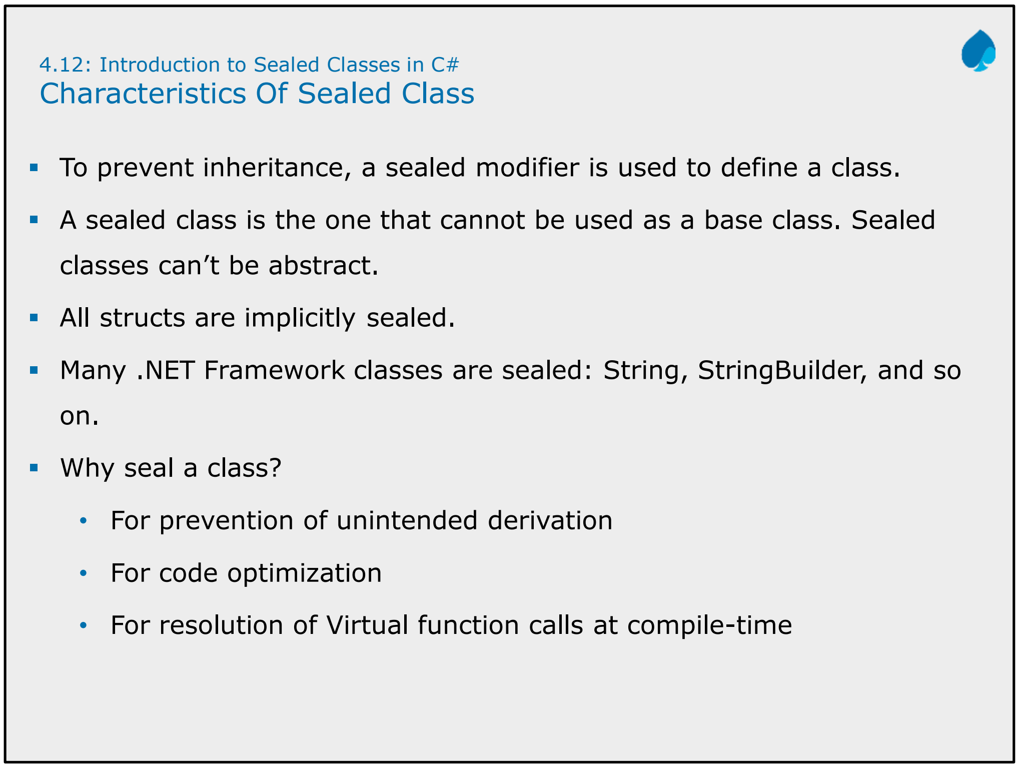
```
public abstract class AbstractClass
{
        public AbstractClass()
        {
        }

         public abstract int AbstractMethod();

          public virtual int VirtualMethod()
          {
                       return 0;
          }
}
```

Derived class:

```
public class DerivedClass : AbstractClass
{
         public DerivedClass()
         {

         }
          public override int AbstractMethod()
          {
               return 0;
          }
          public override int VirtualMethod()
          {
               return base.VirtualMethod ();
          }
}
```

Demo

- Function Overriding, Abstract Class and Abstract Methods



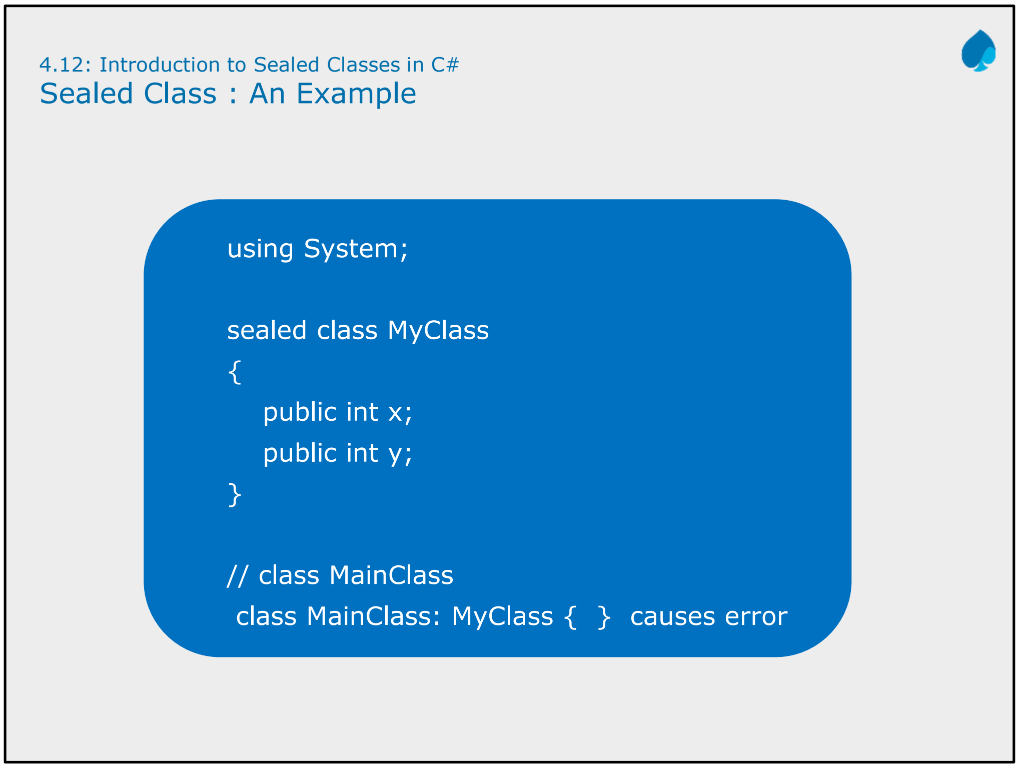
4.12: Introduction to Sealed Classes in C#

Characteristics Of Sealed Class

- To prevent inheritance, a sealed modifier is used to define a class.
- A sealed class is the one that cannot be used as a base class. Sealed classes can't be abstract.
- All structs are implicitly sealed.
- Many .NET Framework classes are sealed: String, StringBuilder, and so on.
- Why seal a class?
  - For prevention of unintended derivation
  - For code optimization
  - For resolution of Virtual function calls at compile-time

Characteristics of Sealed Class:

Inheritance is powerful and useful; however, sometimes you require to prevent it. For example, you might have a class that encapsulates the initialization sequence of some specialized hardware device, such as a medical monitor. In this case, you don't want users of your class to be able to change the way the monitor is initialized, setting the device incorrectly. Whatever the reason, in C#, it is easy to prevent a class from being inherited by using the keyword **sealed**.



4.12: Introduction to Sealed Classes in C#

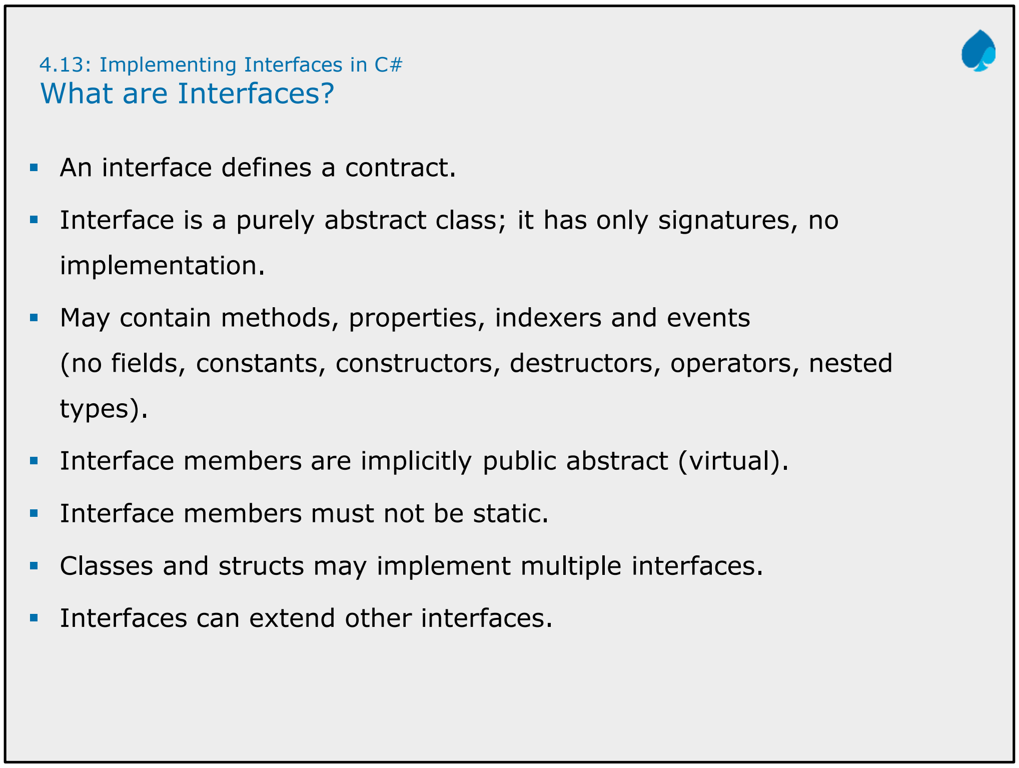
Sealed Class : An Example

```
using System;

sealed class MyClass
{
    public int x;
    public int y;
}

// class MainClass
 class MainClass: MyClass { } causes error
```

4.12: Introduction to Sealed Classes in C#

Sealed Static Class

- In C#, static class is created by using static keyword
- A static class can only contain static data members, static methods, and a static constructor.
- It is not allowed to create objects of the static class.
- Static classes are sealed



4.13: Implementing Interfaces in C#

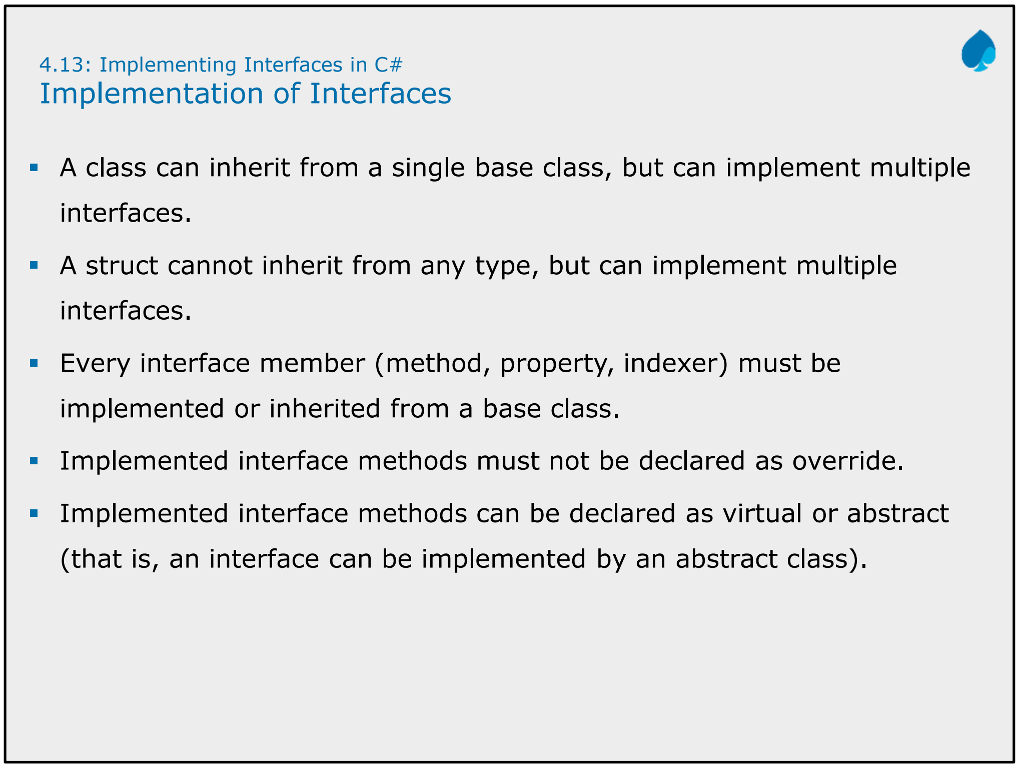
What are Interfaces?

- An interface defines a contract.
- Interface is a purely abstract class; it has only signatures, no implementation.
- May contain methods, properties, indexers and events (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly public abstract (virtual).
- Interface members must not be static.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

What are interfaces?:

In object-oriented programming, it is sometimes helpful to define what a class must do, but not how it will do it. You have already seen an example, the abstract method. While abstract classes and methods are useful, it is possible to take this concept a step further. In C#, you can fully separate specification of interface of a class 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. That is, an interface provides no implementation whatsoever. It specifies what must be done, but not how. Once an interface is defined, any number of classes can implement it. Also, one class can implement any number of interfaces.



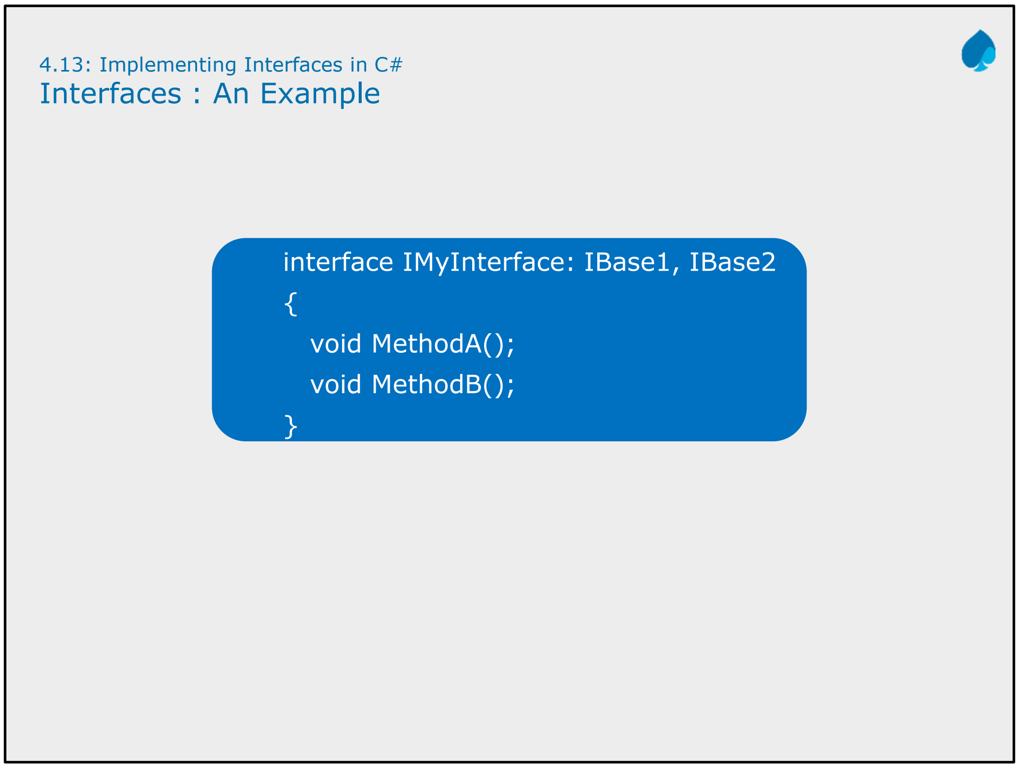
4.13: Implementing Interfaces in C#

Implementation of Interfaces

- A class can inherit from a single base class, but can implement multiple interfaces.
- A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, indexer) must be implemented or inherited from a base class.
- Implemented interface methods must not be declared as override.
- Implemented interface methods can be declared as virtual or abstract (that is, an interface can be implemented by an abstract class).

Implementation of Interfaces:

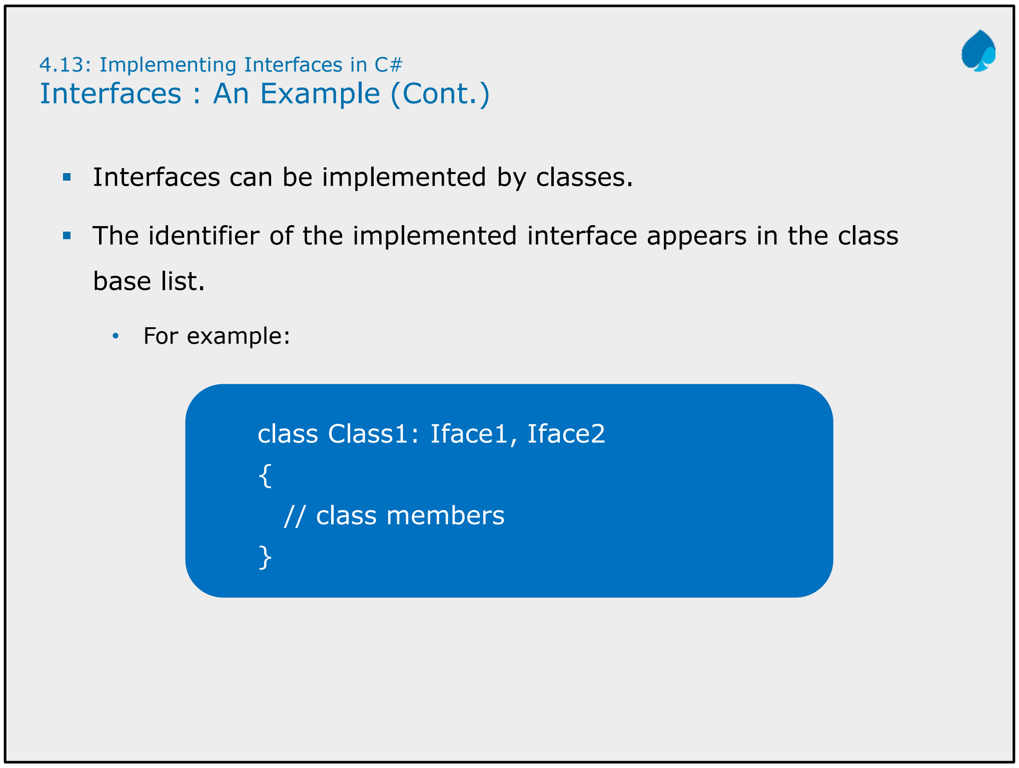
To implement an interface, a class must provide bodies (implementations) for the methods described by the interface. Each class is free to determine the details of its own implementation. Thus, two classes might implement the same interface in different ways, but each class still supports the same set of methods. Therefore, code that has knowledge of the interface can use objects of either class since the interface to those objects is the same. By providing the interface, C# allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.



4.13: Implementing Interfaces in C#

Interfaces : An Example

```
interface IMyInterface: IBase1, IBase2
{
   void MethodA();
   void MethodB();
}
```

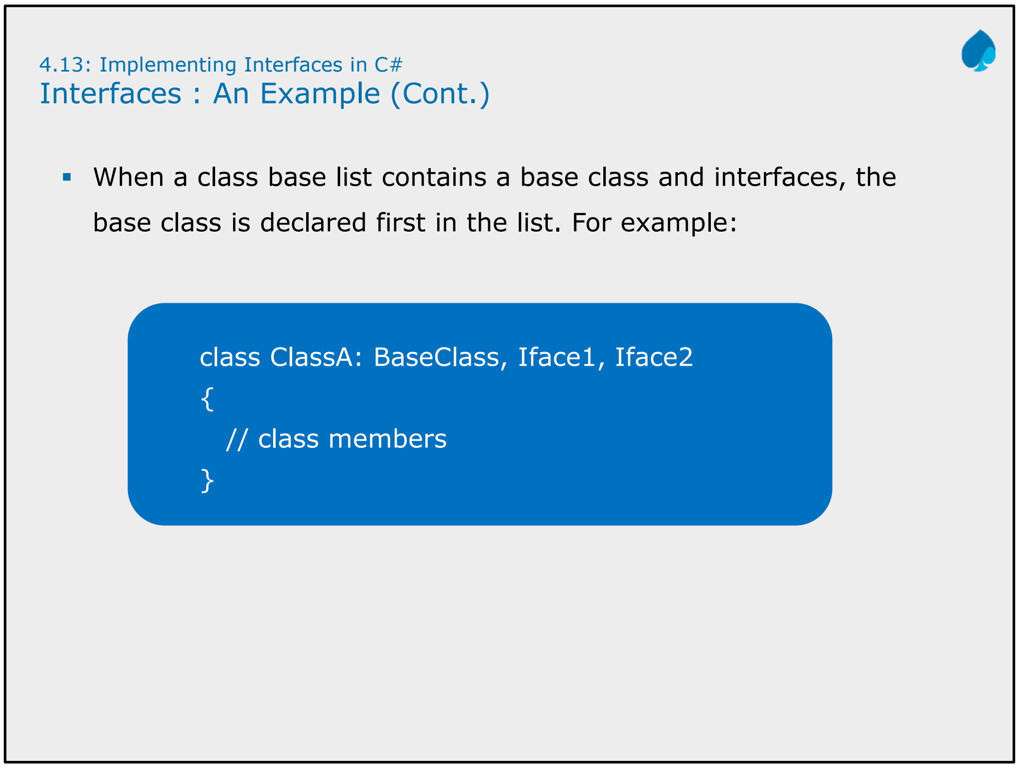


4.13: Implementing Interfaces in C#

Interfaces : An Example (Cont.)

- Interfaces can be implemented by classes.
- The identifier of the implemented interface appears in the class base list.
  - For example:

```
class Class1: Iface1, Iface2
{
   // class members
}
```

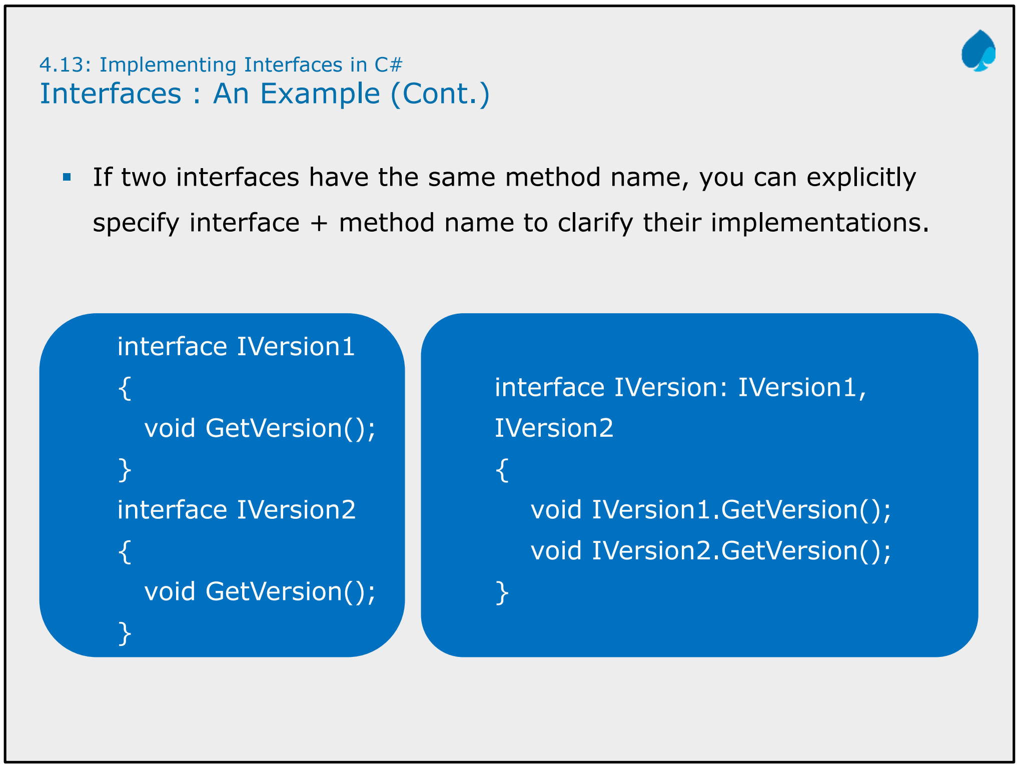


4.13: Implementing Interfaces in C#

Interfaces : An Example (Cont.)

- When a class base list contains a base class and interfaces, the base class is declared first in the list. For example:

```
class ClassA: BaseClass, Iface1, Iface2
{
   // class members
}
```



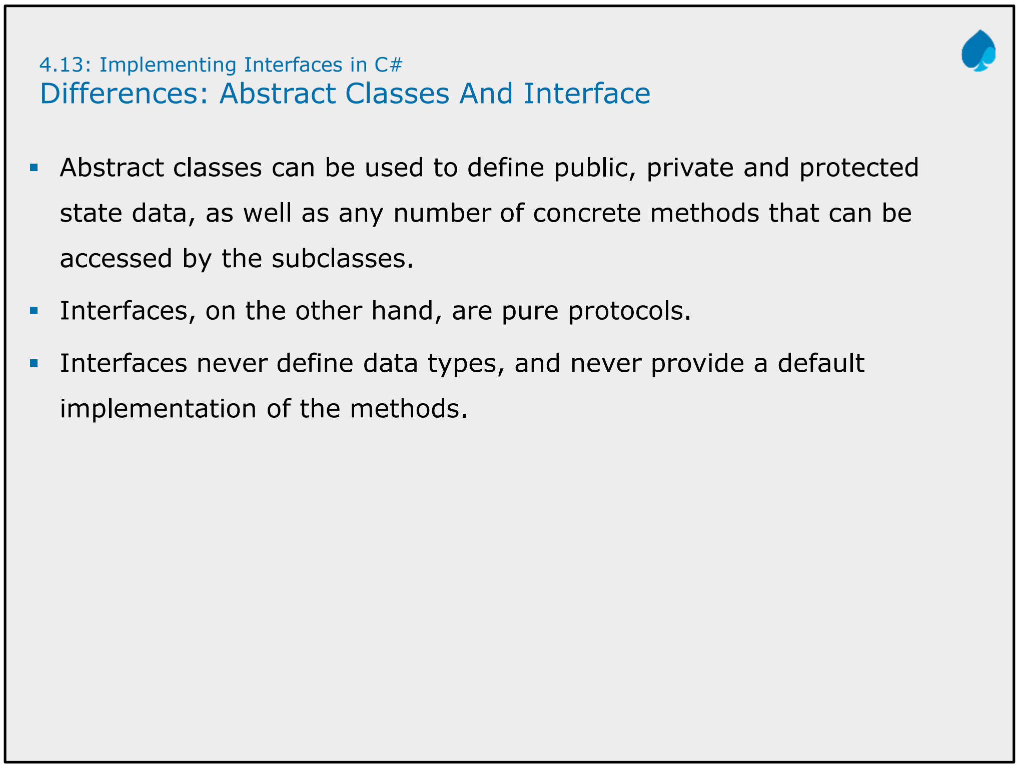
4.13: Implementing Interfaces in C#

Interfaces : An Example (Cont.)

- If two interfaces have the same method name, you can explicitly specify interface + method name to clarify their implementations.

```
interface IVersion1
{
   void GetVersion();
}
interface IVersion2
{
   void GetVersion();
}
```

```
interface IVersion: IVersion1,
IVersion2
{
    void IVersion1.GetVersion();
    void IVersion2.GetVersion();
}
```



4.13: Implementing Interfaces in C#

Differences: Abstract Classes And Interface

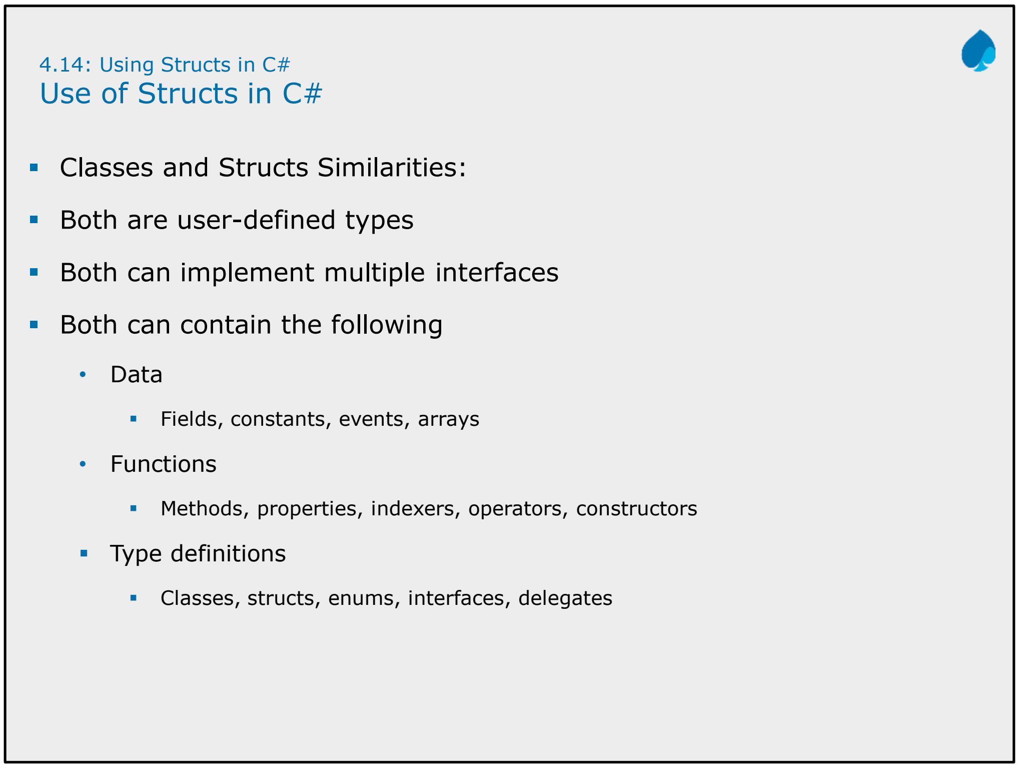
- Abstract classes can be used to define public, private and protected state data, as well as any number of concrete methods that can be accessed by the subclasses.
- Interfaces, on the other hand, are pure protocols.
- Interfaces never define data types, and never provide a default implementation of the methods.

Differences: Abstract Classes and Interface

One of the most challenging parts of C# programming involves ascertaining when to create an interface and when to use an abstract class when you want to describe functionality but not implementation. The general rule is: When you can fully describe the concept in terms of “what it does” without needing to specify any “how it does it,” then you should use an interface. If you need to include some implementation details, then you need to represent your concept using an abstract class.

Demo

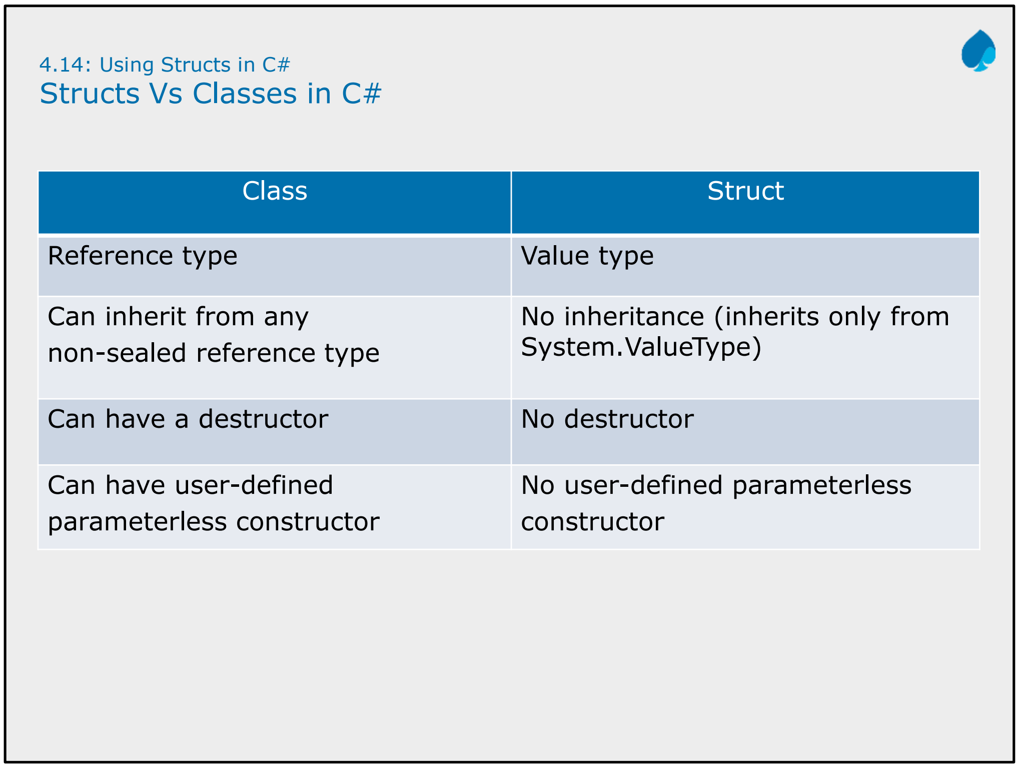
- Creating and using sealed classes and Interfaces in C#



4.14: Using Structs in C#

Use of Structs in C#

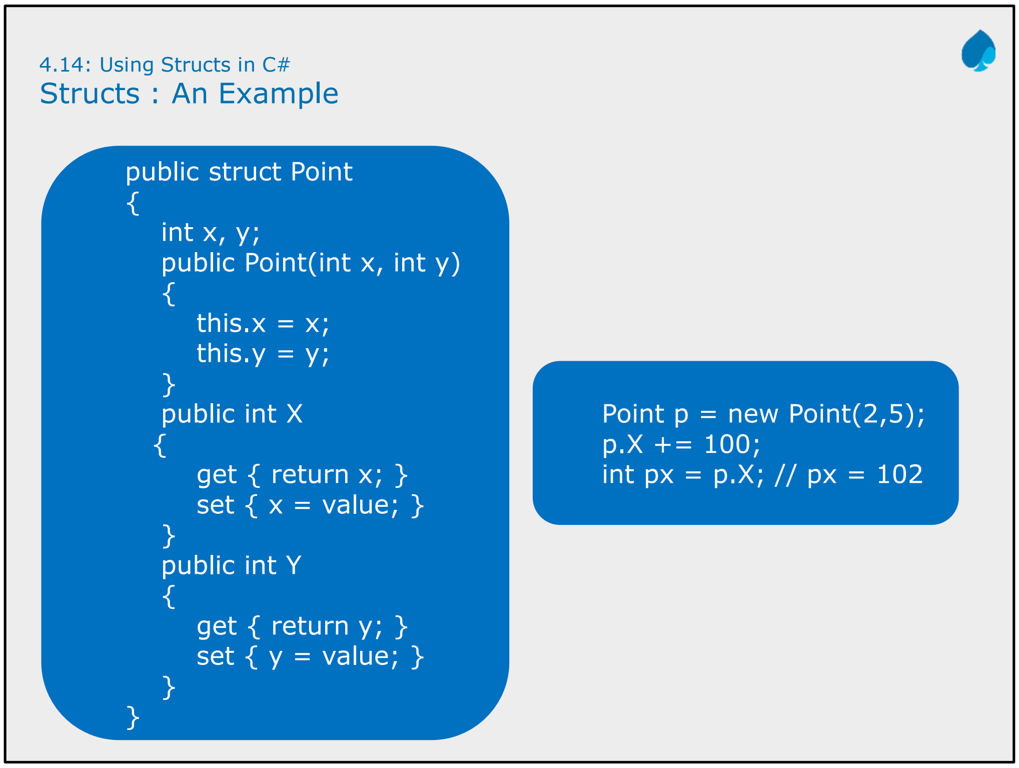
- Classes and Structs Similarities:
- Both are user-defined types
- Both can implement multiple interfaces
- Both can contain the following
  - Data
    - Fields, constants, events, arrays
  - Functions
    - Methods, properties, indexers, operators, constructors
  - Type definitions
    - Classes, structs, enums, interfaces, delegates



4.14: Using Structs in C#

Structs Vs Classes in C#

Class	Struct
Reference type	Value type
Can inherit from any non-sealed reference type	No inheritance (inherits only from System.ValueType)
Can have a destructor	No destructor
Can have user-defined parameterless constructor	No user-defined parameterless constructor

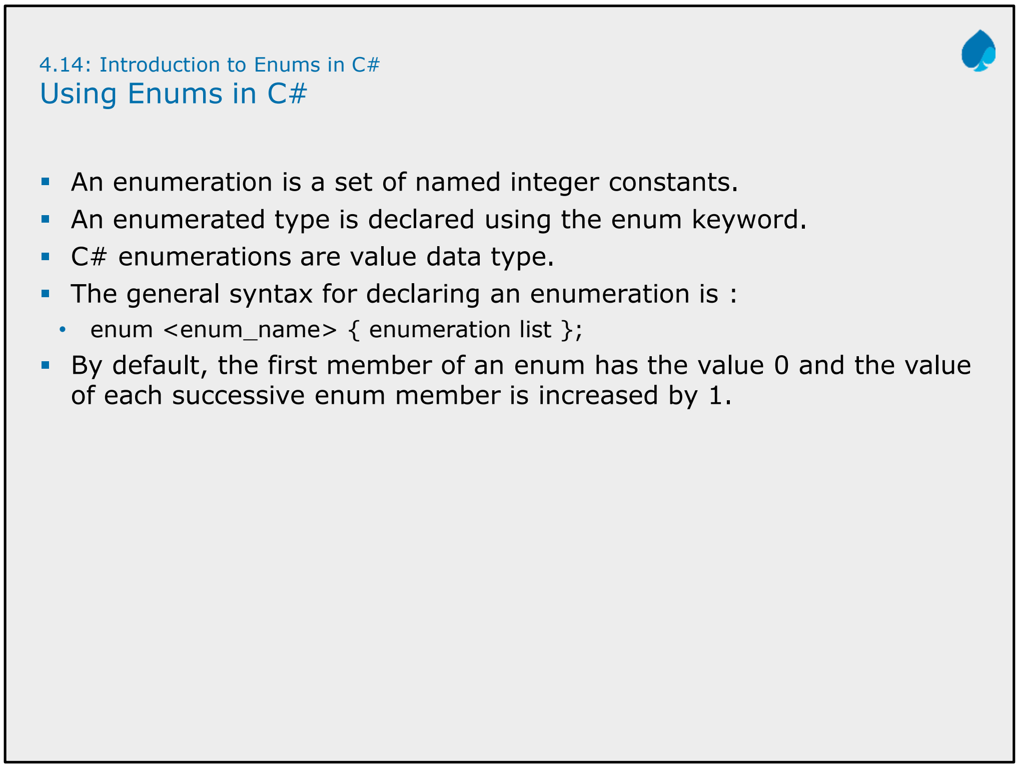


4.14: Using Structs in C#

Structs : An Example

```
public struct Point
{
    int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X
   {
        get { return x; }
        set { x = value; }
    }
    public int Y
    {
        get { return y; }
        set { y = value; }
    }
}
```

```
Point p = new Point(2,5);
p.X += 100;
int px = p.X; // px = 102
```

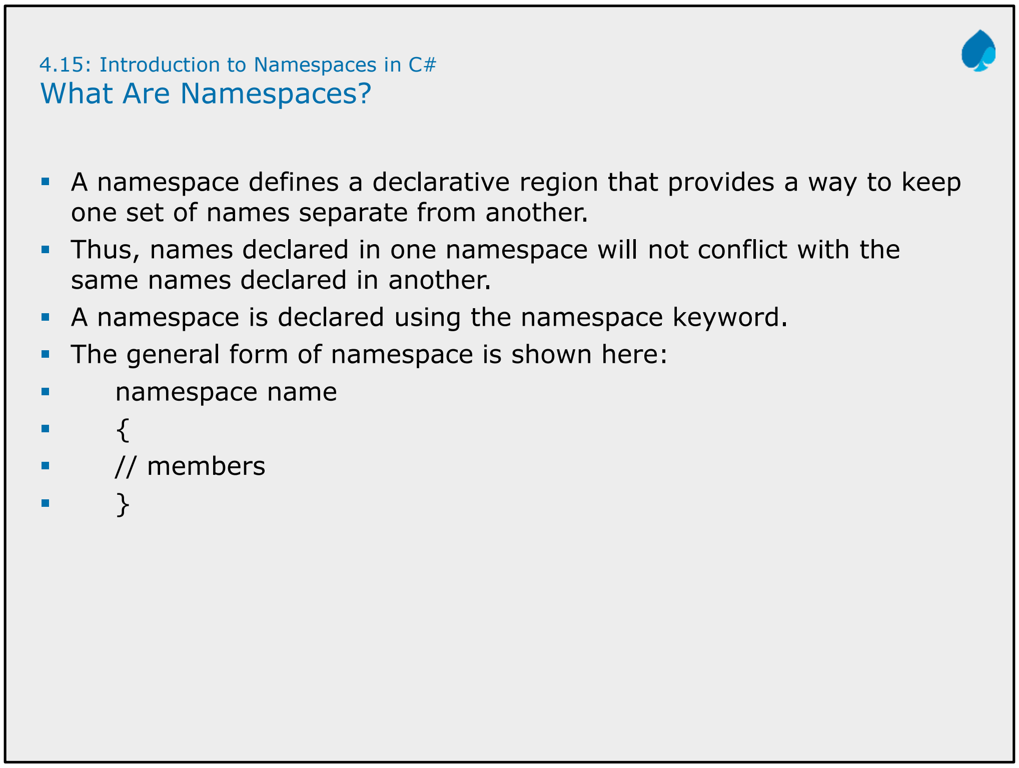


4.14: Introduction to Enums in C#

Using Enums in C#

- An enumeration is a set of named integer constants.
- An enumerated type is declared using the enum keyword.
- C# enumerations are value data type.
- The general syntax for declaring an enumeration is :
  - enum <enum_name> { enumeration list };
- By default, the first member of an enum has the value 0 and the value of each successive enum member is increased by 1.

```
using System;
namespace EnumApplication
{
            class EnumProgram
            {
                        enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
                        static void Main(string[] args)
                        {
                                    int WeekdayStart = (int)Days.Mon;
                                    int WeekdayEnd = (int)Days.Fri;
                                    Console.WriteLine("Monday: {0}",
WeekdayStart);
                                    Console.WriteLine("Friday: {0}",
WeekdayEnd);
                                    Console.ReadKey();
                        }
            }
 }
```



4.15: Introduction to Namespaces in C#

What Are Namespaces?

- A namespace defines a declarative region that provides a way to keep one set of names separate from another.
- Thus, names declared in one namespace will not conflict with the same names declared in another.
- A namespace is declared using the namespace keyword.
- The general form of namespace is shown here:
- namespace name
- {
- // members
- }

Namespaces:

Namespaces are important because there has been an explosion of variable, method, property, and class names over the past few years. These include library routines, third-party codes, and your own code. Without namespaces, all of these names would compete for slots in the global namespace and conflicts would arise. For example, if your program defines a class called **Finder**, it can conflict with another class called **Finder** supplied by a third-party library that your program uses. Fortunately, namespaces prevent this type of problem, because a namespace localizes the visibility of names declared within it.

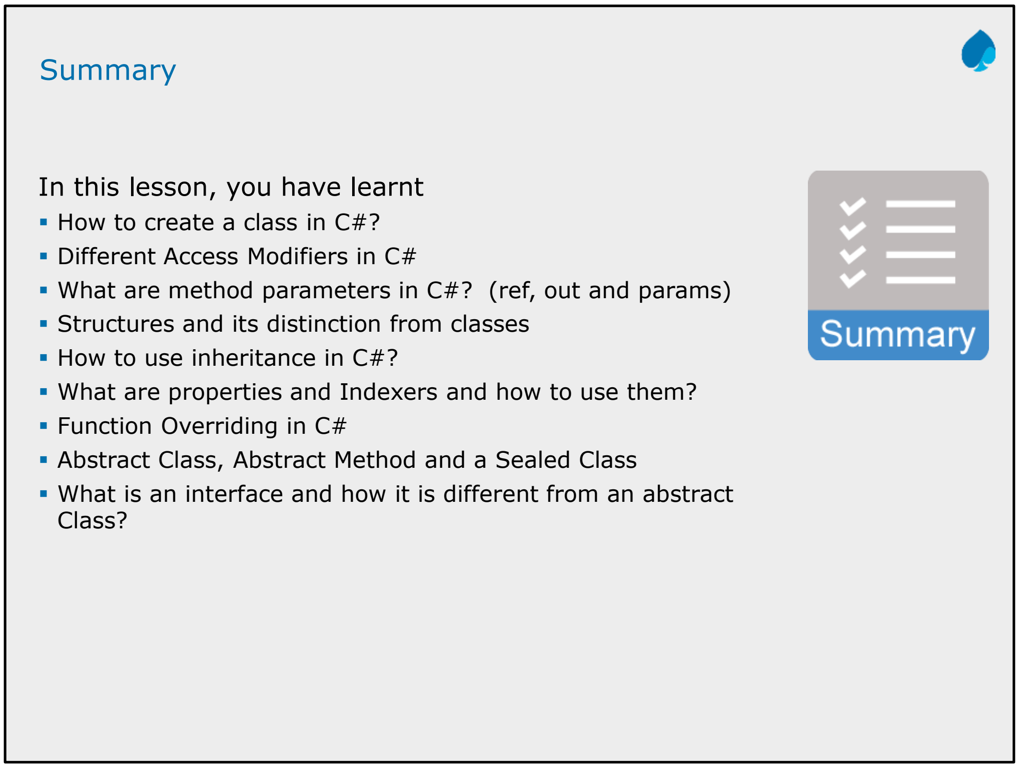
A namespace is declared using the **namespace** keyword. The general form of namespace is shown below:

namespace name { // members }

Here, **name** is the name of the namespace. Anything defined within a namespace is said to be within the scope of that namespace. Thus, namespace defines a scope. Within a namespace, you can declare classes, structures, delegates, enumerations, interfaces, or another namespace.

Demo

- Using Structs & Enums in C#



Summary

In this lesson, you have learnt

- How to create a class in C#?
- Different Access Modifiers in C#
- What are method parameters in C#? (ref, out and params)
- Structures and its distinction from classes
- How to use inheritance in C#?
- What are properties and Indexers and how to use them?
- Function Overriding in C#
- Abstract Class, Abstract Method and a Sealed Class
- What is an interface and how it is different from an abstract Class?

Summary

Add the notes here.

Review Question

- Question 1: How is class different from a structure?
- Question 2: Why is class called as a “is-a” relationship?
- Question 3: What are the different access specifiers in C#?
- Question 4: What is a Sealed Class in C#?
- Question 5: Can abstract class be sealed?
- Question 6: How is abstract class different from an interface?
- Question 7: How is function overriding implemented in C#?
- Question 8: What are the different method parameters in C#?