**Lecture 5 – Inheritance and Polymorphism**

Polymorphism, deduction, multiple inheritance. Partial class. Nullable data types

The **abstract** modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the **abstract**modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

## [Example](javascript:void(0))

In this example, the class Square must provide an implementation of Area because it derives from ShapesClass:

C#

abstract class ShapesClass

{

abstract public int Area();

}

class Square : ShapesClass

{

int side = 0;

public Square(int n)

{

side = n;

}

// Area method is required to avoid

// a compile-time error.

public override int Area()

{

return side \* side;

}

static void Main()

{

Square sq = new Square(12);

Console.WriteLine("Area of the square = {0}", sq.Area());

}

interface I

{

void M();

}

abstract class C : I

{

public abstract void M();

}

}

// Output: Area of the square = 144

Abstract classes have the following features:

* An abstract class cannot be instantiated.
* An abstract class may contain abstract methods and accessors.
* It is not possible to modify an abstract class with the [sealed (C# Reference)](http://msdn.microsoft.com/en-us/library/88c54tsw.aspx) modifier because the two modifers have opposite meanings. The **sealed** modifier prevents a class from being inherited and the**abstract** modifier requires a class to be inherited.
* A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the **abstract** modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

* An abstract method is implicitly a virtual method.
* Abstract method declarations are only permitted in abstract classes.
* Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:
* public abstract void MyMethod();

The implementation is provided by an overriding method[override (C# Reference)](http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx), which is a member of a non-abstract class.

* It is an error to use the [static](http://msdn.microsoft.com/en-us/library/98f28cdx.aspx) or [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

* It is an error to use the **abstract** modifier on a static property.
* An abstract inherited property can be overridden in a derived class by including a property declaration that uses the [override](http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) modifier.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173150.aspx).

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

C#

interface I

{

void M();

}

abstract class C : I

{

public abstract void M();

}

In this example, the class DerivedClass is derived from an abstract class BaseClass. The abstract class contains an abstract method, AbstractMethod, and two abstract properties, X and Y.

C#

abstract class BaseClass // Abstract class

{

protected int \_x = 100;

protected int \_y = 150;

public abstract void AbstractMethod(); // Abstract method

public abstract int X { get; }

public abstract int Y { get; }

}

class DerivedClass : BaseClass

{

public override void AbstractMethod()

{

\_x++;

\_y++;

}

public override int X // overriding property

{

get

{

return \_x + 10;

}

}

public override int Y // overriding property

{

get

{

return \_y + 10;

}

}

static void Main()

{

DerivedClass o = new DerivedClass();

o.AbstractMethod();

Console.WriteLine("x = {0}, y = {1}", o.X, o.Y);

}

}

// Output: x = 111, y = 161

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

BaseClass bc = new BaseClass(); // Error

you will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

**C# Interface**

An interface contains definitions for a group of related functionalities that a [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or a [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx) can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the [interface](http://msdn.microsoft.com/en-us/library/87d83y5b.aspx) keyword, as the following example shows.

C#

interface IEquatable<T>

{

bool Equals(T obj);

}

Any class or struct that implements the [IEquatable<T>](http://msdn.microsoft.com/en-us/library/ms131187.aspx) interface must contain a definition for an [Equals](http://msdn.microsoft.com/en-us/library/ms131190.aspx) method that matches the signature that the interface specifies. As a result, you can count on a class that implementsIEquatable<T> to contain an Equals method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of IEquatable<T> doesn’t provide an implementation for Equals. The interface defines only the signature. In that way, an interface in C# is similar to an abstract class in which all the methods are abstract. However, a class or struct can implement multiple interfaces, but a class can inherit only a single class, abstract or not. Therefore, by using interfaces, you can include behavior from multiple sources in a class.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members](http://msdn.microsoft.com/en-us/library/ms173150.aspx).

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. For links to examples, see [Related Sections](http://msdn.microsoft.com/en-us/library/ms173156.aspx#bkmk_relatedsections). An interface can't contain constants, fields, operators, instance constructors, destructors, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be [static](http://msdn.microsoft.com/en-us/library/98f28cdx.aspx).

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the IEquatable<T> interface. The implementing class, Car, must provide an implementation of the [Equals](http://msdn.microsoft.com/en-us/library/ms131190.aspx) method.

C#

public class Car : IEquatable<Car>

{

public string Make {get; set;}

public string Model { get; set; }

public string Year { get; set; }

// Implementation of IEquatable<T> interface

public bool Equals(Car car)

{

if (this.Make == car.Make &&

this.Model == car.Model &&

this.Year == car.Year)

{

return true;

}

else

return false;

}

}

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a [get](http://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor. The class that implements the interface can declare the same property with both a **get** and [set](http://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see [Explicit Interface Implementation (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173157.aspx) and [Interface Properties (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/64syzecx.aspx).

Interfaces can implement other interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces implement. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (class ClassName : InterfaceName). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement the interface members instead of using the inherited implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see [Polymorphism](http://msdn.microsoft.com/en-us/library/ms173152.aspx).

## [Interfaces Summary](javascript:void(0))

An interface has the following properties:

* An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
* An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
* Interfaces can contain events, indexers, methods, and properties.
* Interfaces contain no implementation of methods.
* A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

Partial Class

There are several situations when splitting a class definition is desirable:

* When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
* When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
* To split a class definition, use the [partial](http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx) keyword modifier, as shown here:

C#

public partial class Employee

{

public void DoWork()

{

}

}

public partial class Employee

{

public void GoToLunch()

{

}

}

The **partial** keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the **partial** keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as **public**, **private**, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

C#

class Container

{

partial class Nested

{

void Test() { }

}

partial class Nested

{

void Test2() { }

}

}

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

C#

[SerializableAttribute]

partial class Moon { }

[ObsoleteAttribute]

partial class Moon { }

They are equivalent to the following declarations:

C#

[SerializableAttribute]

[ObsoleteAttribute]

class Moon { }

The following are merged from all the partial-type definitions:

* XML comments
* interfaces
* generic-type parameter attributes
* class attributes
* members

For example, consider the following declarations:

C#

partial class Earth : Planet, IRotate { }

partial class Earth : IRevolve { }

They are equivalent to the following declarations:

C#

class Earth : Planet, IRotate, IRevolve { }

### [**Restrictions**](javascript:void(0))

There are several rules to follow when you are working with partial class definitions:

* All partial-type definitions meant to be parts of the same type must be modified with **partial**. For example, the following class declarations generate an error:

C#

public partial class A { }

//public class A { } // Error, must also be marked partial

* The **partial** modifier can only appear immediately before the keywords **class**, **struct**, or **interface**.
* Nested partial types are allowed in partial-type definitions as illustrated in the following example:

C#

partial class ClassWithNestedClass

{

partial class NestedClass { }

}

partial class ClassWithNestedClass

{

partial class NestedClass { }

}

* All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
* The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
* The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
  + [public](http://msdn.microsoft.com/en-us/library/yzh058ae.aspx)
  + [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx)
  + [protected](http://msdn.microsoft.com/en-us/library/bcd5672a.aspx)
  + [internal](http://msdn.microsoft.com/en-us/library/7c5ka91b.aspx)
  + [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5.aspx)
  + [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw.aspx)
  + base class
  + [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx) modifier (nested parts)
  + generic constraints

For more information, see [Constraints on Type Parameters (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/d5x73970.aspx).

Nullable types

Nullable types are instances of the [System.Nullable<T>](http://msdn.microsoft.com/en-us/library/b3h38hb0.aspx) struct. A nullable type can represent the correct range of values for its underlying value type, plus an additional **null** value. For example, a Nullable<Int32>, pronounced "Nullable of Int32," can be assigned any value from -2147483648 to 2147483647, or it can be assigned the **null** value. A Nullable<bool> can be assigned the values [true](http://msdn.microsoft.com/en-us/library/eahhcxk2.aspx) [false](http://msdn.microsoft.com/en-us/library/67bxt5ee.aspx), or [null](http://msdn.microsoft.com/en-us/library/edakx9da.aspx). The ability to assign**null** to numeric and Boolean types is especially useful when you are dealing with databases and other data types that contain elements that may not be assigned a value. For example, a Boolean field in a database can store the values **true** or **false**, or it may be undefined.

C#

class NullableExample

{

static void Main()

{

int? num = null;

// Is the HasValue property true?

if (num.HasValue)

{

System.Console.WriteLine("num = " + num.Value);

}

else

{

System.Console.WriteLine("num = Null");

}

// y is set to zero

int y = num.GetValueOrDefault();

// num.Value throws an InvalidOperationException if num.HasValue is false

try

{

y = num.Value;

}

catch (System.InvalidOperationException e)

{

System.Console.WriteLine(e.Message);

}

}

}

The example will display the output:

num = Null

Nullable object must have a value.

For more examples, see [Using Nullable Types (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx)

## [Nullable Types Overview](javascript:void(0))

Nullable types have the following characteristics:

* Nullable types represent value-type variables that can be assigned the value of **null**. You cannot create a nullable type based on a reference type. (Reference types already support the **null** value.)
* The syntax **T?** is shorthand for [Nullable<T>](http://msdn.microsoft.com/en-us/library/b3h38hb0.aspx), where **T** is a value type. The two forms are interchangeable.
* Assign a value to a nullable type just as you would for an ordinary value type, for example int? x = 10; or double? d = 4.108. A nullable type can also be assigned the value **null**: int? x = null.
* Use the [Nullable<T>.GetValueOrDefault](http://msdn.microsoft.com/en-us/library/58x571cw.aspx) method to return either the assigned value, or the default value for the underlying type if the value is **null**, for example int j = x.GetValueOrDefault();
* Use the [HasValue](http://msdn.microsoft.com/en-us/library/sksw8094.aspx) and [Value](http://msdn.microsoft.com/en-us/library/ydkbatt6.aspx) read-only properties to test for null and retrieve the value, as shown in the following example: if(x.HasValue) j = x.Value;
  + The **HasValue** property returns **true** if the variable contains a value, or **false** if it is **null**.
  + The **Value** property returns a value if one is assigned. Otherwise, a [System.InvalidOperationException](http://msdn.microsoft.com/en-us/library/system.invalidoperationexception.aspx) is thrown.
  + The default value for **HasValue** is **false**. The **Value** property has no default value.
  + You can also use the **==** and **!=** operators with a nullable type, as shown in the following example: if (x != null) y = x;
* Use the **??** operator to assign a default value that will be applied when a nullable type whose current value is **null** is assigned to a non-nullable type, for example int? x = null; int y = x ?? -1;
* Nested nullable types are not allowed. The following line will not compile: Nullable<Nullable<int>> n;

Questions

1. What problems does abstract class solve?
2. What abstract class can have?
3. Can the abstract class contain a virtual method?
4. What does new operator defines in method description?
5. What problems does interface solve?
6. How class implements interface?
7. What can interface hold?
8. What are the methods to get reference to interface?
9. What problems explicit interface solve?