برنامه نویسی پیشرفته C#

۱۲و ۱۹ آبان ۹۸

ملکی مجد

topics

- Interface
- Abstract class
- Sealed class

- real power of inheritance comes from inheriting from an interface.

interface

- interface
  - does not contain any code or data
  - just specifies the methods and properties for the heirs

- By using an interface, you can **completely separate** the **names and signatures** of the methods of a class from the method’s **implementation**.

Abstract class

- similar to interfaces, however, abstract classes **can contain code and data**.

- Can specify certain methods of an abstract class as **virtual**
  - so that a class that inherits from the abstract class can optionally provide its own implementation of these methods

Understanding interfaces

- We want to define a new class in which you can **store collections of objects**, a bit like you would use an **array**.

- *RetrieveInOrder*
    - ordinary array !
    - retrieve objects in a sequence (depends on the type)?

```
int CompareTo(object obj)
{
        // return 0 if this instance is equal to obj
        // return < 0 if this instance is less than obj
        // return > 0 if this instance is greater than obj
        ...
}
```

Defining an interface

```
interface Icomparable
{
        int CompareTo(object obj);
}
```

Keyword interface + no access modifier + no implementation + no data filed

- a struct can implement an interface

Interface example

```
interface ILandBound
{
            int NumberOfLegs();
}


class Horse : ILandBound
{
            ...
            public int NumberOfLegs()
                        { return 4; }
}
```

inherit from another class and implement an interface at the same time

```
interface ILandBound
        { ...}

class Mammal
        { ...}

class Horse : Mammal , ILandBound
        { ...}
```

- The **base** class is always named **first**, followed by a comma, followed by the interface
- A class can inherit from several interfaces
- An interface, *InterfaceA*, can inherit from another interface, *InterfaceB*
  - any class that implements *InterfaceA* must provide implementations of all the methods in *InterfaceB* and *InterfaceA*.

Referencing a class through its interface

```
Horse myHorse = new Horse(...);
ILandBound iMyHorse = myHorse; // legal
```

```
interface ILandBound
        { ...}

class Mammal
        { ...}

class Horse
        : Mammal , ILandBound
        { ...}
```

- The technique of referencing an object through an interface is useful because you can use it to define methods that can take different types as parameters, as long as the types implement a specified interface

```
int FindLandSpeed(ILandBound landBoundMammal)
        { ...}
if (myHorse is ILandBound)
        { ILandBound iLandBoundAnimal = myHorse;}
```

Working with multiple interfaces

- A class can have at most one base class, but it is allowed to implement an unlimited number of interfaces. A class must implement all the methods declared by these interfaces.

```
class Horse : Mammal, ILandBound, Igrazable
{
      ...
}
```

Explicitly implementing an interface

```
class Horse : ILandBound, Ijourney
{
        ...
        int ILandBound.NumberOfLegs()
                { return 4; }
         int IJourney.NumberOfLegs()
                { return 3; }
}
```

The methods are private!

How to use private method

```
Horse horse = new Horse();
int legs = horse.NumberOfLegs();//compile err
IJourney journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();
```

Abstract classes

- parts of the derived classes to share common implementations!
- Duplication in code is a warning sign

```
class Horse : Mammal, ILandBound, Igrazable
{
        ...
        void IGrazable.ChewGrass()
       {
                Console.WriteLine("Chewing grass");
                        // code for chewing grass
       }
}
class Sheep : Mammal, ILandBound, Igrazable
{
       ...
        void IGrazable.ChewGrass()
       {
                Console.WriteLine("Chewing grass");
                        // same code as horse for chewing grass
       }
}
```

One solution: new class

```
class GrazingMammal : Mammal, Igrazable
{
             void IGrazable.ChewGrass()
{
            // common code for chewing grass
             Console.WriteLine("Chewing grass");
}}
class Horse : GrazingMammal, ILandBound
            { ...}
class Sheep : GrazingMammal, ILandBound
            { ...}
```

One solution: new class

```
class GrazingMammal : Mammal, Igrazable
{
                 void IGrazable.ChewGrass()
{
                // common code for chewing grass
                 Console.WriteLine("Chewing grass");
}}
class Horse : GrazingMammal, ILandBound
                { ...}
class Sheep : GrazingMammal, ILandBound
                { ...}
```

one thing is not quite right : instances of the *GrazingMammal* class !

The *GrazingMammal* class is an abstraction of common functionality rather than an entity in its own right.

- To declare that creating instances of a class **is not allowed**, you can declare that the class is **abstract** by using the *abstract* keyword,

```
abstract class GrazingMammal : Mammal, Igrazable
        { ...}

GrazingMammal myGrazingMammal = new GrazingMammal(...); // illegal
```

Abstract methods

- An abstract class can contain abstract methods
- A derived class *must* override this method
- An abstract method cannot be private

- An abstract method is useful if it **does not make sense to provide a default implementation** in the abstract class but you want to ensure that an inheriting **class provides its own implementation** of that method

Sealed classes

- you can use the *sealed* keyword to **prevent a class from being used as a base class** if you decide that it should not be

- If any class attempts to use *Horse* as a base class, a compile-time error will be generated
  - a sealed class cannot declare any virtual methods
  - an abstract class cannot be sealed

Sealed methods

- use the *sealed* keyword to declare that an individual method (in an unsealed class)
    - means that a derived class cannot override this method

    - You can seal only a method declared with the *override* keyword

- An interface introduces the name of a method.
- A virtual method is the first implementation of a method
- An override method is another implementation of a method.
- A sealed method is the last implementation of a method