# برنامه نویسی پیشرفته
# C#

۱۲ و ۱۴ آبان ۹۸

ملکی مجد

# topics

- Review
- Inheritance
- Create a derived class from a base class
- Call base class constructor/method
- Hiding method
- virtual method
- override method
- polymorphism

# review

- Class

- filed
- method
- public and private
- static
- Object – Instance
- constructor

- You can use **inheritance** as a tool to **avoid repetition** when defining different classes that have a number of features **in common** and are quite clearly **related** to one another

- For example,
  - *managers, manual workers*, and *all employees* of a factory.
  - have different responsibilities and perform different tasks

- Inheritance in programming is all about **classification**
  - it's a relationship between classes

- Mammal پستاندار
  - نفس کشیدن
  - شیردادن به طفل
  - خون گرم
- Horse اسب
  - سم
  - یورتمه رفتن
- Whale نهنگ
  - باله شنا
  - شنا کردن

- inherit from *Mammal*
  - *Horse, Whale, Aardvark, Human*

# Syntax - Using inheritance

```
class BaseClass
{
 ...
}

class DerivedClass : BaseClass
{
 ...
}

class DerivedSubClass : DerivedClass
{
 ...
}
```

```
class Mammal
{
        public void Breathe()
        {
        ...
        }
        public void SuckleYoung()
        {
        ...
        }
        ...
}
```

```
class Horse : Mammal
{
        ...
        public void Trot()
        { ... }
}
class Whale : Mammal
{
        ...
        public void Swim()
        { ... }
}
```

```
Horse myHorse = new Horse();
myHorse.Trot();
myHorse.Breathe();
myHorse.SuckleYoung();
```

- In C#, a class is allowed to derive from, at most, one base class; a class is *not allowed* to derive from two or more classes

- All structures actually inherit from an abstract class named *System.ValueType*.
  - cannot define your own inheritance hierarchy with structures,
  - and you cannot define a structure that derives from a class or another structure.

# System.Object

- All classes implicitly derive from *System.Object*.
  - all classes that you define automatically inherit all the features of the *System.Object* class
    - E.g., *ToString*

- the C# compiler silently rewrites the *Mammal* class as the following code

```
class Mammal : System.Object
{
 ...
}
```

# Calling base-class constructors

- A derived class automatically contains all the fields from the base class (In addition to the methods that it inherits)
- a constructor in a derived class
  - call the constructor for its base class as part of the initialization,

```
class Mammal // base class
{

    public Mammal(string name) //constructor for base class
    {
     ...
     }
}
```

```
class Horse :  Mammal // derived  class
{

    public Horse(string name)
            : base(name) // calls Mammal(name)
 {
     ...
     }}
```

# Calling base-class constructors (2)

- If you don't explicitly call a base-class constructor in a derived-class constructor
  - the **compiler** attempts to silently insert a **call to the base class's default constructor**
- Example:

```
Class Horse : Mammal
{

    public Horse (string name)
    {
     ...
     }
}
```

```
class Horse :  Mammal
{

    public Horse(string name)
            : base()
 {
     ...
     }}
```

This works if *Mammal* has a public default constructor!

# Assigning classes

Horse myHorse = new Horse(...);
Whale myWhale = myHorse;  // **error** - different types

Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // **legal**, Mammal is the base class of Horse

- higher up the inheritance hierarchy
  - all *Horse*s are *Mammal*s
  - think of a *Horse* simply as a special type of *Mammal*

# Assigning classes(2)

- Mammal myMammal = new  Mammal(...);
- Horse myHorse = myMammal; // **error**

- Note
  - not all *Mammal* objects are *Horse*s

- You can assign a *Mammal* object to a *Horse* variable as long as you first check that the *Mammal* is really a *Horse*, by using the ***as*** **or** ***is*** operator or by using a **cast**

# Casting data safely

- By **using a cast**, you can specify that, *in your opinion*, the data referenced by an object has a specific type and that it is safe to reference the object by using that type
- The C# **compiler will not** check that this is the case, but the **runtime will**
- If the type of object in memory does not match the cast, the runtime will throw an ***InvalidCastException***,
- C# provides very **useful operators** that can help you perform casting in a much more elegant manner: the ***is*** and ***as*** **operators**

# The *is* operator

- You can use the *is* operator to verify that the type of an object is what you expect it to be

```
WrappedInt wi = new WrappedInt();
…
object o = wi;
if (o is WrappedInt)
{
WrappedInt temp = (WrappedInt)o; // This is safe; o is a WrappedInt
}
```

# The *as* operator

- The *as* operator fulfills a similar role to *is* but in a slightly truncated manner
  - The runtime attempts to cast the object to the specified type. If the cast is **successful**, the **result** is returned. If the cast is **unsuccessful**, the *as* operator evaluates to the ***null***

```
WrappedInt wi = new WrappedInt();
…
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp != null)
{
      ... // Cast was successful
}
```

- the *as* operator
  - to check that *myMammal* refers to a *Horse*, and if it does, the assignment to *myHorseAgain* results in *myHorseAgain* referring to the same *Horse* object.
  - If *myMammal* refers to some other type of *Mammal*, the *as* operator returns *null* instead.

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // myMammal refers to a Horse
…
Horse myHorseAgain = myMammal as Horse; // OK - myMammal was a Horse
…
Whale myWhale = new Whale(...);
myMammal = myWhale;
…
myHorseAgain = myMammal as Horse; // returns null - myMammal was a Whale
```

- Any additional methods defined by the *Horse* or *Whale* class are not visible through the *Mammal* class.

- Horse myHorse = new Horse(...);
- Mammal myMammal = myHorse;
- myMammal.Breathe(); // **OK** - Breathe is part of the Mammal
- classmyMammal.Trot(); // **error** - Trot is not part of the Mammal class

# Declaring new methods

- If a base class and a derived class happen to declare two methods that have the same signature, you will receive a **warning** when you compile the application
  - A method in a derived class **masks (or hides)** a method in a base class that has the same signature

- The method signature refers to the name of the method and the number and types of its parameters, but not its return type

# example

```
class Mammal
{
    public void Talk() // assume that all mammals can talk
    {…}
}

class Horse : Mammal
{
    public void Talk() // horses talk in a different way from other mammals!
    {…}
}
```

the compiler generates a warning message informing you that *Horse.Talk* hides the inherited method *Mammal.Talk*

sure that the two methods to have the same signature, silence the warning by using the *new* keyword

```
class Mammal
{
        public void Talk()
        {…}
}

class Horse : Mammal
{
        new public void Talk()
        {…}
}
```

# virtual methods

- A method that is intended to be overridden is called a *virtual* method
  - Overriding a method is a mechanism for providing **different implementations of the same method +** the methods are all related because they are intended to perform the same task
  - Hiding a method is a means of replacing one method with another—the methods are usually unrelated and might perform totally different tasks

- Overriding a method is a useful programming concept
- hiding a method is often an error

- You can mark a method as a virtual method by using the ***virtual* keyword**. For example, the *ToString* method in the *System.Object* class is defined like this:

```
namespace System
{
    class Object
    {
        public virtual string ToString() {…}
    }
}
```

# *override* methods

- If a base class declares that a method is virtual,
  - a derived class can use the ***override*** **keyword** to declare another implementation of that method

```
class Horse : Mammal
{
    …
    public override string ToString()
    { …
    }
}
```

- The new implementation of the method in the derived class can call the **original** implementation of the method in the base class by using **the *base* keyword**

```
class Horse : Mammal
{
        …
        public override string ToString()
        {
            string temp = base.ToString();
        }
}
```

# Polymorphic (many forms) methods

- There are some important rules you must follow when you declare polymorphic methods
  - (by using the *virtual* and *override* keywords)

1. A virtual method cannot be private
   - Similarly, override methods cannot be private
2. The signatures of the virtual and override methods must be identical
   - both methods must return the same type

3. You can only override a virtual method
4. If the derived class does not declare the method by using the *override* keyword, it does not override the base class method; it hides the method (a compile-time warning)
5. An override method is implicitly virtual and can itself be overridden in a further derived class

# Virtual methods and polymorphism

- Using virtual methods, you can **call different versions of the same method**, based on the **object type determined dynamically** at run time

```
class Mammal // base class
{
    public virtual string GetTypeName()
    {
        return "This is a mammal" ;
    }
}
```

```
class Horse : Mammal
{
    public override string GetTypeName()
    {
        return "This is a horse";
    }
}
```

```
class Whale : Mammal
{
    public override string GetTypeName()
    {
        return "This is a whale";
    }
}
```

```
class Aardvark : Mammal
{
}
```

the *override* keyword used by the *GetTypeName* method in the *Horse* and *Whale* classes,
the *Aardvark* class does not have a *GetTypeName* method.

```
Mammal myMammal;
Horse myHorse = new Horse(...);
Whale myWhale = new Whale(...);
Aardvark myAardvark = new Aardvark(...);

myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName()); // ???
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // ???
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // ???
```

the *override* keyword used by the *GetTypeName* method in the *Horse* and *Whale* classes,
that the *Aardvark* class does not have a *GetTypeName* method.

```
Mammal myMammal;
Horse myHorse = new Horse(...);
Whale myWhale = new Whale(...);
Aardvark myAardvark = new Aardvark(...);

myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName()); // Horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // Whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // Mammal
```

What if  GetTypeName() is not virtual method?

# *protected* access

- it is useful for a base class to **allow derived classes to access some of its members while also hiding these members from classes** that are not part of the inheritance hierarchy

- If a **class A is derived from another class B**, it can **access** the **protected** class members of class B. In other words, inside the derived class A, a protected member of class B is effectively public.
- If a **class A is not derived from another class B**, it **cannot access** any protected members of class B. So, within class A, a protected member of class B is effectively private