

# Introduction to the C# object model

January 14, 2008

# Goals of the C# object system.

**polymorphism** the ability to write code which operates on many types—realized by inheritance, interfaces, and overloading

**encapsulation** the ability to make separate a class's behavior from its implementation details—realized with access modifiers

**extensibility** the ability to extend class functionality—realized with inheritance and virtual methods.

# Terminology

- Field: a variable declared in a class.
- Method: a procedure associated with a class.
- Member: a field or method.
  
- Instance of <class>: an object of type <class>

# Inheritance

- All classes inherit from a base class (default is `System.Object`).
- Derived classes automatically include the members of their base classes (e.g. `ToString()` from `System.Object`).
- Child classes *extend* base classes by adding new members, and overriding virtual methods.
- Can treat an instance of a derived class as an instance of its base class.

## Basic inheritance example

```
using System;

class BaseSimple{
    public void Print(){
        Console.Out.WriteLine( "BaseSimple" );}
}

class ChildSimple : BaseSimple { }

class Runner{
    public static void Main( string [] s){
        (new BaseSimple()).Print(); // "BaseSimple"
        (new ChildSimple()).Print(); // "BaseSimple"
    }
}
```

*Static Dispatch:* New *overloaded* methods are called using an object's compile-time type.

```
class BaseNew{  
    public void Print(){  
        Console.Out.WriteLine( "BaseNew" );} }  
  
class ChildNew : BaseNew {  
    new public void Print(){  
        Console.Out.WriteLine( "ChildNew" );} }  
  
class Runner{  
    public static void Main( string [] s ){  
        ChildNew c = new ChildNew();  
        BaseNew b = c;  
  
        c.Print();    // "ChildNew"  
        b.Print();    // "BaseNew"  
    } }
```

*Dynamic Dispatch: Virtual methods called using an object's run-time type.*

```
class BaseVirt{
    public virtual void Print(){
        Console.Out.WriteLine("BaseVirt");} }

class ChildVirt : BaseVirt {
    public override void Print(){
        Console.Out.WriteLine("ChildVirt");} }

class Runner{
    public static void Main(string[] s){
        ChildVirt c = new ChildVirt();
        BaseVirt b = c;

        c.Print(); // "ChildVirt"
        b.Print(); // "ChildVirt"
    } }
```

# Overriding Rules

- Base classes may mark methods with `virtual`. Such methods are virtual and may be overridden by derived classes.
- Derived classes must mark methods with `override` to override them.
- Derived classes can mark methods with `sealed` prevent subclasses from overriding the methods.
  - By default methods are sealed.
  - A derived class can seal a virtual method to stop further overriding.
- Compiler will raise an error if there's a chance of ambiguity.



## Calling base class methods with `base`

Sometimes we need to call a base class's methods explicitly.

```
class ChildVirt : BaseVirt {  
    public override void Print(){  
        Console.Out.WriteLine("ChildVirt_says_Hi!")  
        Console.Out.WriteLine("Base_virt_says:");  
        // calls base method  
        base.Print();  
    }  
  
    // calls base constructor  
    public ChildVirt(int x): base(x) {}  
}
```

Without the `base` keyword, there would be no way to access such methods!

# Class Modifiers and Static Members

- Static members
  - One copy of member per class (as opposed to per instance).
  - Example: `Foo.NumberOfFoos`
- Class modifiers
  - Marking a class `abstract` means it can't be instantiated, only derived from.
  - Marking a class `sealed` means it can't be derived from, only instantiated.
  - Marking a class `static` means a class is both `sealed` and `abstract`. (Can only contain static members, and can be initialized with a zero-argument static constructor.)

## Static class example

```
using System.Collections.Generic;

public static class Logger{
    private static List<string> myList;

    static Logger() { myList = new List<string>(); }

    public static void Append(string s) {
        myList.Add(s); }
}
```

# Interfaces declare contracts that a class must follow.

- Interfaces list methods which must appear in a class.
- Methods may use interface names for argument and result types (bounded polymorphism).
- Classes can implement interfaces in two ways
  - Implicitly (the normal way), interface methods added directly to class and accessed as usual.
  - Explicitly, interface members are declared with special syntax and accessed through casts. Useful in the case where two interfaces declare methods with the same name.

## Example: Implicit Interface Implementation

```
interface IWindow {  
    void Draw();  
}  
  
public class Display: IWindow {  
    // Implicit Interface Implementation  
    public void Draw(){ Console.Out.WriteLine ("A");  
}  
  
class Runner{  
    static void Main(string[] args){  
        Display c = new Display();  
        c.Draw(); // "A"  
    }  
}
```

## Multiple interfaces can conflict.

```
interface IWindow {  
    // Implementations should print to the screen  
    void Draw();  
}
```

```
interface ICowboy {  
    // Implementations should get out a gun  
    void Draw();  
}
```

```
// Trouble!  
public class WesternGame: IWindow, ICowboy {...}
```

## Example: Explicit Interface Implementation

```
class WesternGame: IWindow, ICowboy {  
    // Explicit Interface Implementations  
    void IWindow.Draw(){  
        Console.Out.WriteLine ("Drawing_Picture"); }  
    void ICowboy.Draw(){  
        Console.Out.WriteLine ("Drawing_Six_Shooter");  
}  
  
class Runner{  
    static void Main(string[] args){  
        WesternGame w = new WesternGame();  
  
        // Error: w.Draw();  
        ((ICowboy) w).Draw(); // "Drawing Picture"  
        ((IWindow) w).Draw(); // "Drawing Six Shooter"  
    }}
```

```
string x = (string) someObject
```

- Up-casts:
  - Convert instances of a child class to a parent class or interface.
  - Always succeeds.
- Down-casts:
  - Convert instances of a parent class to a child class.
  - May fail and throw `InvalidCastException`
  - Use `as` or `is` to check if a cast is safe.
- Generics provide an elegant way to write (for example) collection classes without casting.



# Access modifiers protect class implementation details.

Access modifiers may be attached to class, field, and method declarations.

Modifier	Meaning
public	No visibility restrictions.
protected <sup>1</sup>	Visible to classes derived from the defining class
internal <sup>2</sup>	Visible anywhere in the same assembly.
protected internal <sup>1</sup>	Visible according to protected. Also, member visible according to internal.
private <sup>1</sup>	Visible only within defining class

---

<sup>1</sup>Only applicable to elements defined in a class (i.e. not to classes defined only in a namespace).

<sup>2</sup>internal is the default access modifier.