Chair of Software Engineering

C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 – May 2007

Lecture 3: OO Programming with C#

Lisa (Ling) Liu

Overview

- Reviewing the pillars of OOP
- C#'s encapsulation services
- C#'s inheritance support
- C#'s polymorphism support
- Interfaces and collections

Reviewing the Pillars of OOP

- Encapsulation
  - Encapsulate the inner details of implementation
  - Protect data
- Inheritance
  - Build new class based on existing class definitions
  - Embody the is-a relationship between types
- Polymorphism
  - Treat the related objects in the same way

Encapsulation support in C#

- An object's field data and implementation details should not be directly accessed.
  - ➢ Define a pair of traditional accessor and mutator methods
    - Making use of access modifier (private, public)
  - ➢ Define a named property

```
public class Employee
{
        private string fullName;
        .......

        // Accessor
        public string GetFullName()
        {
              return fullName;
         }
        //Mutator
        public void SetFullName (string s)
        {
              fullName = s;
        }
}
```

enforcing encapsulation using traditional accessors and mutators

Class properties

- NET languages use properties to enforce encapsulation
- Stimulate public accessible data

```
static void Main()
{
    Employee p = new Employee();
    p.Name = “Tom”;
    Console.WriteLine (p.Name);
}
```

Define a property

- A C# property is composed of a get block and set block
- value represents the implicit parameter used during a property assignment

```
public class Employee
{
        private string fullName;
        ...

        // Property for fullName
        public string Name
        {
                get { return fullName; }
                set { fullName = value; }
        }

}
```

Internal representation of C# properties

- Under the hood, the properties always map to "real" accessors and mutators

```
C:\Eclipse3.1\Csharp\BookExamples\Code\Ch_04 Code\Employee...
File  View  Help
get_Benefits : class Employees.Employee/BenefitPackage()
get_Company : string()
get_ID : int32()
get_Name : string()
get_Pay : float32()
get_SocialSecurityNumber : string()
set_Age : void(int32)
set_Benefits : void(class Employees.Employee/BenefitPackage)
set_Company : void(string)
set_ID : void(int32)
set_Name : void(string)
set_Pay : void(float32)
set_SocialSecurityNumber : void(string)
Age : instance int32()
Benefits : instance class Employees.Employee/BenefitPackage()
.assembly Employees
```

Property visibility

```
// The get and set logic is both public,
// given the declaration of the property.
public string SocialSecurityNumber
{
        set { return empSSN; }
        get { empSSN = value; }
}
```

```
// Object users can only get the value,
// however derived types can set the value.
public string SocialSecurityNumber
{
        set { return empSSN; }
         protected get { empSSN = value; }
}
```

```
public string SocialSecurityNumber
{
        // Now as a read-only property

        get { empSSN = value; }
}
```

```
public string SocialSecurityNumber
{
        // Now as a write-only property

        set { return empSSN; }
}
```

Static property

```
public class Employee
{
    private static string fullName;
    ...

    // Property for fullName
    public static string Name
    {
        get { return fullName; }
        set { fullName = value; }
    }

}
```

Inheritance support in C#

- Define a subclass (using colon operator ":")
- A subclass gains all non-private data and members from its base class
- A subclass does not inherit constructors from the base class

```
public class Manager : Employee
{
        private ulong numberOfOptions;
        ...
        public Manager ( )
         ...
}
```

Subclass constructor

- By default, any subclass constructor first calls the *default constructor* (parameterless constructor) of a base class.

- Call special base class constructor with keyword base

```
public class A
{
        public A (int n)
        {
                …
        }

}
```

```
public class B : A
{
        public B (int n) : base (n)
        {
                …
        }

}
```

this keyword: self-reference

- Represent current object
  this.FullName, this.EmpID

- Forward constructor calls

```
public class Employee
{
        …
        public Employee ()
        {
                …
        }

        public Employee (int n) : this ()
        {
        }
}
```

Single base class and sealed class

- In C#, a given class has exactly one direct base class
- sealed keyword is used to define a class that cannot be inherited

```
public sealed class A
{
        …
}
```

```
//Compiler error
public class B : A
{
        …
}
```

Another form of reuse

- has-a relationship (also known as the *containment/delegation* model)

```
public class B
{
        ...
        public void Mb1()
        {
                ...
        }
        private void Mb2 ()
        {
                ...
        }
}
```

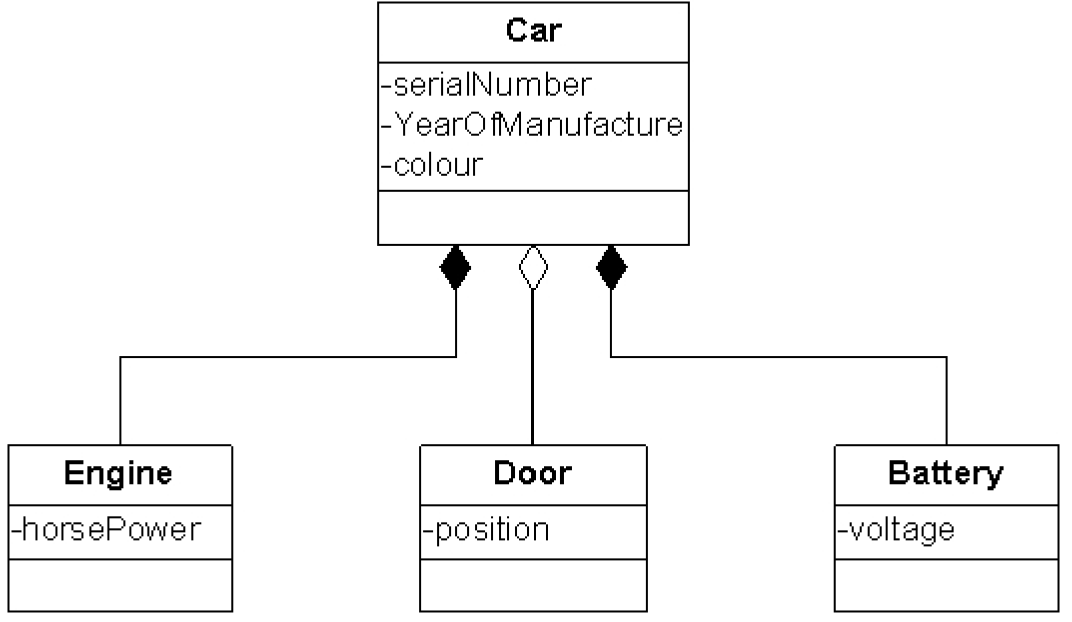
```
public class A
{
        ...
        B b = new B ();
        public void Ma1 ()
        {
                b.Mb1 ();
        }
}
```

reuse another class's functionality through contained object

How to reuse method Mb2 in Class A?

Nested class

- Nested class is a member of containing class
- The relationship between containing class and nested class is similar to composition relationship, except the containing class can completely access the nested class
- Nested class can also access the private member of containing class
- Usually, nested class acts as a helper for the containing class, and is not designed for outside world



In the diagram above, the battery and the engine have no meaning outside of the car, as the car cannot work without either of them, so the relationship is formed using composition. However, a car can work without doors, so the relationship is formed using aggregation.

```
namespace MyCars
{
	public class Car
	{
		// Aggregation uses instance of class outside of this class
		protected Door FrontRight;
		protected Door FrontLeft;
		protected Door RearRight;
		protected Door RearLeft;

		// inner class used to create objects
		// that are intrinsically linked to the class car
		protected class Engine
		{
			public int horsePower;

		}
		protected class Battery
		{
			public int voltage;
		}

		// Composition uses instances of objects of inner classes
		protected Engine TheEngine;
		protected Engine The Battery;

		public Car ()
		{
			TheEngine = new Engine ();
			TheBattery = new Battery ();
		}
	}

	public class Door
	{
		public int position;
	}
}
```

Define and use nested class

- A nested class is defined in the same manner as a normal class
- A nested class can access any member of the containing class
- The this keyword reference in the nested class only holds a reference to the nested class

Polymorphism support in C#

Through inheritance, a class can be used as more than one type; it can be used as its own type, any base types, or any interface if it implements interfaces. This is called polymorphism.

- Two choices to change the data and behavior of a base class
  - Replace the base member with a new derived member
  - Override a virtual base member (achieve the polymorphism)

Completely take over a base class member

- Base class member is declared as virtual
- Derived class member is declared as override
- Fields cannot be virtual; only methods, properties, events and indexers can be virtual
- When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class

```
public class BaseClass
{
    public virtual void DoWork () { }
    public virtual int WorkProperty
    {
        get {return 0;}
    }
}
```

```
public class DerivedClass : BaseClass
{
    public override void DoWork () { }
    public override int WorkProperty
    {
        get {return 1;}
    }
}
```

```
DerivedClass B = new DerivedClass ();

B.DoWork (); // calls the new method

BaseClass A = (BaseClass) B;

A.DoWork (); //also calls the new method
```

Leverage base class members

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the base keyword

```
public class BaseClass
{
    public virtual void DoWork ()
    {
        …
    }
    …
}
```

```
public class DerivedClass : BaseClass
{
    public override void DoWork ()
    {
        base.DoWork();
    }
}
```

Preventing further overriding

- Stop virtual inheritance by declaring an override as sealed (put the sealed keyword before the override keyword)

Overriding and method selection

- C# compiler selects the best method to call if more than one method is compatible with the call

```
public class A
{
    public virtual void Foo (int n)
    {
        Console.WriteLine ("A: Foo (int)");
    }
    public class B : A
    {
        public override void Foo (int n)
        {
            Console.WriteLine ("B: Foo (int)");
        }
        public void Foo (double n)
        {
            Console.WriteLine ("B: Foo (double)");
        }
}
```

```
public class Test
{
    public static void Main ()
    {
        B b;
        b = new B();
        b.Foo (5);
    }
}
```

Replace a base class member

- Replacing a member of a base class with a new derived member requires the new keyword
- When the new keyword is used, the new class members are called instead of the base class members that have been replaced. Those base class members are called hidden members
- Hidden class members can still be called if an instance of the derived class is cast to an instance of the base class

```
public class BaseClass
{
    public void DoWork () { }
    public int WorkField;
    public int WorkProperty
    {
        get {return 0;}
    }
}
```

```
public class DerivedClass : BaseClass
{
    public new void DoWork () { }
    public new int WorkField;
    public new int WorkProperty
    {
        get {return 1;}
    }
}
```

```
DerivedClass B = new DerivedClass ();

B.DoWork (); // calls the new method

BaseClass A = (BaseClass) B;

A.DoWork (); //calls the old method
```

Difference between new and override

- new members are the members of this class, while override members are the members of the base class and be redefined in this class
- C# allows a subclass' member has the same name with that of the base class. In this case, the member in subclass is seemed as a new member

Abstract class

- Abstract class is defined with abstract keyword
- In C#, it is not allowed to create an instance of an abstract class
- An abstract class may define any number of abstract members
- If a class inherit from an abstract class, it must override the abstract method (enforce polymorphic activities)

```
abstract  public  class Shape
{
    ...
    // Draw() is now completely abstract (note semicolon).
     public abstract void Draw();
     ...
 }


public class Circle : Shape
{
        public Circle() { }
        public Circle(string name): base(name) { }

        // Now Circle must decide how to render itself.
        public override void Draw()
        {
            Console.WriteLine("Drawing {0} the Circle", PetName);
        }
 }
```

Casting

- Implicit casting
- Explicit casting

```
int i = 10;
float f = 0;
f = i; // An implicit conversion, no data will be lost.
i = (int) f; // An explicit conversion. Information will be lost.
```

Implicit casting takes place when passing a derived class object to a base class variable.

Determine a variable's type

is keyword

```
public static void FireThisPerson (Employee e)
{
    if (e is SalesPerson)
    { ... }
    if (e is Manager)
    { ... }
}
```

Define an interface

- An interface is a named collection of semantically related abstract methods
- An interface contains only the signatures of methods, delegates or events
- The implementation of the methods in an interface is done in the class that implements the interface

```
public interface IPointy
{
    // Implicitly public and abstract
    byte Points ();
}
```

- A class (or struct) can implement several interfaces, which are declared in the comma-delimited type list
- Base class must be the first item in the type list

```
public class Triangle : Shape, IPointy
{
    …
    public override void Draw ()
    {
        …
    }

    // IPointy Implementation
    public byte Points
    {
        get {return 3;}
    }
}
```

A class must implement all members defined in the interface

Contrasting interfaces to abstract classes

- Interfaces are pure protocol. Interfaces never define state data and never provide an implementation of the methods
- Interface types are also quite helpful given that C# only support single inheritance
- Interfaces provide another way to inject polymorphic behavior into a system

Invoking interface members

- Directly invoking (if you know the type implements the interface member)
- Casting
- The as keyword
- The is keyword

```
Triangle t = new Triangle ();
byte n;

n = t.Points
```

Developer knows that Triangle implements interface IPointy

```
Triangle t = new Triangle ();
IPointy itPt;
byte n;

try
{
        itPt = (IPointy) t;
        n = itPt.Points;
}
catch (InvalidCastException e)
{
        ...
}
```

Using try/catch logic to determine if the type support the requested interface

```
Triangle t = new Triangle ();
IPointy itPt = t as IPointy;
byte n;

if (itPt != null)
{
        n = itPt.Points;
}
else
{
        ...
}
```

as is used to perform conversions between compatible types

```
Triangle t = new Triangle ();
byte n;

if (t is IPointy)
{
		n = t.Points;
}
else
{
		...
}
```

Explicit interface implementation

- An explicitly implemented member can only be accessed through an instance of the interface
- Syntax of explicit implementation of interface methods

  returnValue InterfaceName.MethodName (args)

```
public interface IDraw3D
{
    void Draw ();
}

public class Shape
{
    ...
    public virtual void Draw ()
    { ...}
}
```

```
public class Line : Shape, IDraw3D
{
    void IDraw3D.Draw ()
    {
        ...
    }
    public override void Draw ()
    {
        ...
    }
}
```

```
// This invokes the overriden Shape.Draw() method
Line myLine = new Line ();
myLine.Draw ();


// This invokes the IDraw3D.Draw() method
Line myLine = new Line ();
IDraw3D i3d = (IDraw3D) myLine;
i3d.Draw ();
```

Avoid name clashes through the explicit interface implementation

```
public interface ILeft
{
    int P ();
}

public interface IRight
{
    int P ();

}
```

```
public class Middle : ILeft, IRight
{
    int ILeft.P ()
    {
        ...
    }
    int IRight.P ()
    {
        ...
    }
}
```

Interface hierarchies

- Build interface hierarchies through inheritance
- Create an interface that derives from multiple base interfaces

```
public interface IDrawable
{
    void Draw ();
}
public interface IPrintable : IDrawable
{
    void Print ();
}
 public interface IRender : IPrintable
{
        void Render ();
}
```

```
public interface ICar
{
    void Drive ();
}
public interface IUnderwaterCar
{
    void Dive ();
}
 public interface IJamesBondCar :
ICar, IUnderwaterCar
{
        void TurboBoost();
}
```

Iterate custom collections

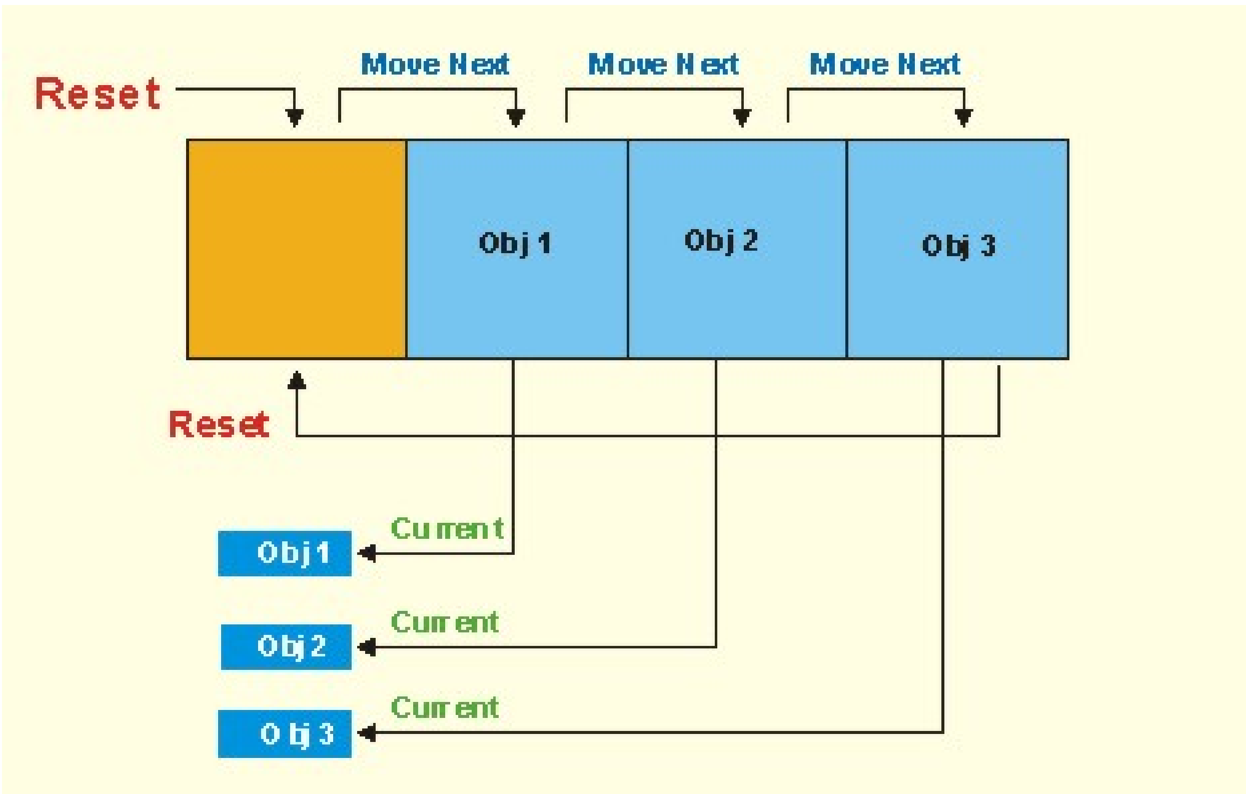
- IEnumerable and IEnumerator interface are used to iterate through a collection of custom types
- IEnumerable interface contain only one abstract member function called GetEnumerator() . This method will return an Interface called IEnumerator
- IEnumerator provides two abstract method and a property to access a particular element in a collection. Reset () , MoveNext() are the abstract methods and Current is the property

IEnumerable interface

```
public interface IEnumerable
{
        IEnumberator GetEnumerator ();
}
```

IEnumerator interface

- IEnumerator is the interface which helps you to iterate through any custom collection



Building cloneable objects

- Implement IClonable interface

```
public interface IClonable
{
    object Clone ();
}
```

IComparable interface

- Defines a generalized comparison method that a value type or class implements to create a type-specific comparison method

```
public interface IComparable
{
    int CompareTo (Object o);
}
```

```
public class Car : IComparable
{
    ...
    int IComparable.CompareTo (object obj)
    {
        Car temp = (Car) obj;
        if (this.CarID > temp.CarID)
            return 1;
        if (this.CarID < temp.CarID)
            return -1;
        else
            return 0;
    }
}
```

```
static void Main ()
{
    Car [ ] myAutos = new Car[5];
    ...
    // Sort car arrary using IComparable
    Array.Sort (myAutos);
}
```

ICompare interface

- ICompare is typically not implemented on the type you are trying to sort

```
public class PetNameComparer : IComparable
{
    ...
    int IComparable.CompareTo (object o1, object o2)
    {
        Car c1 = (Car) o1;
        Car c2 = (Car) o2;
        return string.Compare(t1.PetName, t2.PetName);
    }
}
```

```
static void Main ()
{
    Car [ ] myAutos = new Car[5];
    ...
    // Sort car arrary using IComparable
    Array.Sort (myAutos, new PetNameCompare ());
}
```

System.Collections namespace

- IDictionary
- IHashtable
- IList
- ArrayList
- Queue
- Stack

Questions