C#_4.1 Inheritance Besics

In the language of C#, a class that is ***inherited*** is called a ***base*** class. The class that does the ***inheriting*** is called a ***derived*** class. (Recall C/C++ 11.1). Derived class declaration is same as C/C++ by using " **:** ". The ***base class name*** follows the ***name of the derived class***, and they are separated by a ***colon*** " **:** ". The general form of a derived class

`class derived_class_name : base_class_name {` */* body of class */* `}`

☠ ***Restrictions (Same as Java part 4.1):*** You can specify only ***one base*** class for any ***derived*** class that you create. C# does not support the inheritance of ***multiple base*** classes into a ***single derived*** class. (This differs from C++, in which you can inherit multiple base classes. Be aware of this when converting C++ code to C#.)

❒ ***Controlling access of a derived class:*** A ***derived*** class cannot access those members of the ***base*** class that are ***private***. A ***private*** class ***member*** will remain ***private*** to its class. It is not accessible by any code ***outside*** its class, including ***derived*** classes.

☞ ***Accessing private members using public properties:*** Since a ***property*** allows you to manage access to an ***instance variable***. Use ***public properties*** to provide access to ***private data***. By making a ***property public*** but declaring its *underlying variable* ***private***, a ***derived*** class can still use the ***property***, but it cannot ***directly access*** the underlying ***private*** variable. (In Java ***accessor methods*** used).

☞ ***Protected Access (Recall C/C++ 11.16):*** A ***protected member*** is created using the ***protected*** access modifier. A ***protected member*** of the ***base*** class becomes a ***protected member*** of the ***derived*** class and is, therefore, accessible by the ***derived*** class. Therefore, by using ***protected***, you can create class members that are ***private*** to their ***class*** but that can still be ***inherited*** and ***accessed*** by a ***derived*** class.

➢ ***Protected is not for general use:*** Use ***protected*** when you want to create a member that is ***private*** throughout a ***class hierarchy***, but is otherwise unrestricted. To manage ***access to a value***, use a ***property***.

C#_4.2 Constructors and Inheritance (Introduction is similar to Java part 4.2)

❒ ***When only the derived class defines a constructor:*** Same as Java part 4.2.

❒ Both ***derived*** and ***base*** defines constructors***:*** Similar as ***Java part 4.2***. In this case base keyword is used. base keyword has two uses:

⇨ Use ***base*** to call a ***base class constructor*** ⇨ Use ***base*** to access a ***member of the base class*** that has been ***hidden by a member of a derived class***.

❒ Use ***base*** to call a base class constructor: The general form of expanded declaration of the derived class' constructor declaration and the base keyword is:

`derived-constructor(parameter-list) : base(arg-list) {` */* body of constructor */* `}`

➢ Here, ***arg-list*** specifies any arguments needed by the constructor in the base class. Notice the placement of the colon.

```
class TwoDShape { /* variables etc. */
        public TwoDShape(double w, double h) { Width = w; Height = h; }
        public double Width { /*…*/}
        public double Height { /*…*/}
        public void ShowDim() { /*…*/}                    }
```

```
// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape { /* variables etc. */

            // Call the base class constructor.
    public Triangle(string s, double w, double h) : base(w, h) { Style = s; }
    /*…*/                 }
```

☞ Any ***form*** of ***constructor*** defined by the ***base*** class can be called by ***base***. The constructor executed will be the one that ***matches*** the ***arguments***.

☞ When a ***derived*** class specifies a ***base*** clause, it is calling the constructor of its ***immediate base*** class (even in a multileveled hierarchy). You pass arguments to the ***base constructor*** by specifying them as ***arguments*** to ***base***. If no ***base clause*** is ***present***, then the base class' ***default constructor*** is called ***automatically***.

❒ Use ***base*** to access a member of the base class that has been ***hidden*** by a member of a ***derived*** class: (Name hiding: Recall Java part 1.13 - C/C++- no restriction. Also C# follows Java. It is the name hiding in ***scope*** – i.e. ***same class***. Now we discussing ***name hiding*** between ***two classes***. Idea is same as Java's ***super.member***).

☞ It is possible for a ***derived*** class to define a member that has the ***same name as a member*** in its ***base*** class. When this happens, the ***member*** in the ***base*** class is ***hidden within the derived*** class. It is not technically an ***error*** in C#, the compiler will issue a ***warning message*** - alerts you to the fact that a name is being hidden.

☝ If your ***intent*** is to ***hide*** a ***base*** class ***member***, then to prevent this warning, the ***derived class member*** must be ***preceded*** by the ***new*** keyword. Understand that this use of ***new*** is ***separate*** and ***distinct*** from its use when creating an ***object instance***. Here is an example of name hiding:

```
class A { public int i = 0; }

    /* Create a derived class. The i in A is hidden by the I in B. Notice the use of new. */
class B : A { new int i;      // this i hides the i in A
          public B(int b) { i = b; /* i in B */ }     }
```

⇨ First, ***notice*** the use of ***new***. In essence, it ***tells the compiler*** that you know a ***new variable*** called ***i*** is being ***created*** that ***hides*** the ***i*** in the ***base*** class ***A***. If you leave ***new*** out, a ***warning*** is ***generated***.

☞ There is a second form of base that acts like this, except that it always refers to the base class of the derived class in which it is used. This usage has the general form: `base.member` It is similar to Java's `super.member` . Here, ***member*** can be either a ***method*** or an ***instance variable***. This form of ***base*** is most applicable to situations in which ***member names*** of a ***derived*** class ***hide*** members by the ***same name*** in the ***base*** class.

The instance variable ***i*** in ***B*** hides the ***i*** in ***A***, ***base*** allows access to the ***i*** defined in the ***base*** class. Hidden method ***show( )*** is called through the use of ***base***.

☞ ***new*** is used in this program to tell the ***compiler*** that you know that a new method called ***Show( )*** is being created that hides the ***Show( )*** in ***A***.

```
class A {   public int i = 0;
            public void Show() { Console.WriteLine("i in base: " + i); }   /* Show() in A.*/      }
// Create a derived class.
class B : A { new int i;              // this i hides the i in A
              public B(int a, int b) {          base.i = a;            // this uncovers the i in A
                                                i = b; /*  i in B */ }

               new public void Show() {    // This Show( ) hides the one in A.
                                  base.Show();           // this calls the hidden Show() in A
                                  Console.WriteLine("i in derived class: " + i); /* This displays the i in B.*/    } }
```

C#_4.3 Multilevel Hierarchy (Similar as Java part 4.3)

❒ ***CONSTRUCTORS EXECUTION IN CLASS HIERARCHY:*** Similar as Java part 4.3.

C#_4.4 Base Class References and Derived Objects (Similar as Java part 4.4)

C#_4.5 Method Overriding & Virtual Method (Recall C/C++ virtual function 13.2 and Java part 4.5)

A ***virtual method*** is a method that is declared as ***virtual*** in a ***base*** class. It can be ***redefined*** in one or more ***derived*** classes. Thus, each ***derived*** class can have its own version of a ***virtual*** method. C# determines which version of the method to call ***based upon*** the ***type*** of the ***object*** referred to by the ***reference***—and this determination is made at ***runtime***.

[When different . . . ***same as Java part 4.5: Dynamic method dispatch, inside third braces*** *. . . method are executed]*

❒ ***Declaring virtual methods and method overriding:*** You declare a method as ***virtual*** inside a ***base*** class by preceding its declaration with the *keyword* `virtual`. When a ***virtual method*** is ***redefined*** by a derived class, the `override` modifier is used.

☞ The process of ***redefining a virtual method*** inside a derived class is called ***method overriding***. When overriding a method, the ***name***, ***return type***, and ***signature*** of the overriding method ***must be the same*** as the ***virtual method*** that is being ***overridden***. Also, a ***virtual method*** cannot be specified as ***static*** or ***abstract***.

❒ ***Dynamic method dispatch (runtime polymorphism):*** Same as Java part 4.5.

☞ It is not necessary to ***override a virtual method***. If a derived class does not provide its own version of a ***virtual method***, then the one in the ***base*** class is used.

❐ ***Properties/indexers can be virtual:*** ***Properties*** can be modified by the ***virtual*** keyword and overridden using ***override***. The same is true for ***indexers***.

❐ ***Example (Use of array declaration for overriding methods):*** Similar as Java part.

C#_4.6 Abstract Methods and Abstract Classes (Similar Java part 4.6)

Abstract method: To declare an abstract method, use this general form: `abstract type name(parameter-list);`

☑ Notice, ***no method body*** is present. The ***abstract*** modifier can be used only on ***instance methods***. It cannot be applied to ***static methods***.

☑ An ***abstract method*** is automatically ***virtual***, and there is ***no need*** to use the ***virtual modifier***. In fact, it is an ***error*** to use ***virtual*** and ***abstract*** together.

❐ A ***class*** that contains one or more ***abstract methods*** must also be declared as ***abstract*** by preceding its ***class declaration*** with ***abstract***. Similar to Java part 4.6.

❐ When a ***derived*** class inherits an ***abstract class***, it must ***implement*** all of the ***abstract methods*** in the ***base*** class. Similar to Java part 4.6.

C#_4.7 sealed (Similar to Java's "final" , See Java part 4.7)

To prevent a class from being inherited, precede its declaration with sealed. It is illegal to declare a class as both abstract and sealed, since an abstract class is incomplete by itself and relies upon its derived classes to provide complete implementations.

```
sealed class A { /* . . . */ }
class B : A { /* ERROR! Can't derive class A */ }
```

💣 It is ***illegal*** for ***B*** to inherit ***A*** since ***A*** is declared as ***sealed***.

🛠 ***Sealed*** can also be used on ***virtual methods*** to *prevent further overrides*. Eg: assume a base class called B and a derived class called D. A method declared virtual in B can be declared sealed by D. This would prevent any class that inherits D from overriding the method. This situation is illustrated by the following:

```
class B { public virtual void MyMethod() { /*… */} }
class D : B { sealed public override void MyMethod() { /*… */} }  // This seals MyMethod() and prevents further overrides.
class X : D { public override void MyMethod() { /*… */} }          // Error! MyMethod() is sealed! can't be overridden.
```

☛ Because ***MyMethod()*** is sealed by ***D***, it can't be overridden by ***X***.

C#_4.8 C# object Class (Similar Java part 4.8)

The ***object*** class is an implicit ***base class*** of all other classes for all other types (including the value types). i.e. All C# types are derived from ***object***. Technically, the C# name ***object*** is just another name for ***System.Object***, which is part of the ***.NET Framework*** class library. ***object class*** defines the following ***methods*** which available in every object.

Method	***Purpose***
public virtual bool Equals(object ob)	Determines whether the ***invoking object*** is the ***same*** as the one referred to by ***ob***.
public static bool Equals(object ob1, object ob2)	Determines whether ***ob1*** is the same as ***ob2***.
protected virtual Finalize()	Performs ***shutdown actions*** prior to ***garbage collection***. In C#, ***Finalize*** is accessed through a ***destructor***.
public virtual int GetHashCode()	Returns the ***hash code*** associated with the ***invoking object***.
public Type GetType()	Obtains the ***type*** of an object ***at runtime***.
protected object MemberwiseClone()	Makes a "***shallow copy***" of the ***object***. This is one in which the members are ***copied***, but objects ***referred*** to by members ***are not***.
public static bool ReferenceEquals(object ob1, object ob2)	Determines whether ***ob1*** and ***ob2*** refer to the ***same object***.
public virtual string ToString()	Returns a ***string*** that describes the ***object***.

🛠 By default, the ***Equals(object)*** method determines if the ***invoking*** object ***refers*** to the ***same*** object as the one referred to by the ***argument***. (That is, it determines if the ***two references are the same***.) It returns ***true*** if the objects are the ***same*** and ***false otherwise***.

☞ You can ***override*** this method in ***classes*** that you ***create***. Doing so allows you to define ***what equality means*** relative to a class. For example, you could define ***Equals(object)*** so that it ***compares*** the contents of two ***objects*** for ***equality***.

🛠 The ***Equals(object, object)*** method invokes ***Equals(object)*** to compute its result.

🛠 The ***GetHashCode()*** method returns a ***hash code*** associated with the ***invoking object***. This ***hash code*** can be used with any ***algorithm*** that employs hashing as a means of accessing stored objects.

🛠 If you overload the ***== operator***, then you will usually need to override ***Equals(object)*** and ***GetHashCode()***, because most of the time, you will want the ***== operator*** and the ***Equals(object)*** method to function the same. When ***Equals()*** is overridden, you should also override ***GetHashCode()*** so that the two ***methods*** are ***compatible***.

🛠 The ***ToString() method*** returns a ***string*** that contains a description of the object on which it is called. Also, this ***method*** is ***automatically*** called when an object is output using ***WriteLine()***. Many classes ***override*** this ***method***. Doing so allows them to ***tailor a description*** specifically for the ***types*** of ***objects*** that they create. Eg:

```
using System;
class MyClass {          static int count = 0;      int id;      public MyClass() { id = count; count++; }
                         public override string ToString() { return "MyClass object #" + id; }      /* Override ToString( ) */      }
class Test { static void Main() { MyClass ob1 = new MyClass();           MyClass ob2 = new MyClass();        MyClass ob3 = new MyClass();
                         Console.WriteLine(ob1);    Console.WriteLine(ob2);    Console.WriteLine(ob3); }}    /* ToString( ) called automatically */
```

```
OUTPUT:
MyClass object #0
MyClass object #1
MyClass object #2
```

C#_4.9 Boxing and Unboxing

Boxing: A ***reference*** of type ***object*** can be used to refer to any ***other type***, including ***value types***. When an ***object reference*** refers to a ***value type***, a process known as ***boxing*** occurs. ***Boxing*** causes the value of a ***value type*** to be stored in an ***object instance***, which can be used like any other ***object***. In all cases, ***boxing*** occurs ***automatically***. You simply assign a value to an ***object reference***. C# handles the ***rest***.

Unboxing: ***Unboxing*** is the process of retrieving a value from an ***object***. This action is performed using a ***cast*** from the ***object reference*** to the desired ***value type***.

☠ Attempting to ***unbox*** an object into an ***incompatible type*** will result in a ***runtime error***. Following is a simple boxing/unboxing example.

```
using System;
class BoxingDemo { static void Main() {  int x;        object obj;
                                         x = 10;
obj = x;                /* box x into an object */
int y = (int)obj;       /* unbox obj into an int */
Console.WriteLine(y); }}
```

☛ Notice that the value in ***x*** is ***boxed*** simply by ***assigning*** it to ***obj***, which is an ***object reference***. The ***integer*** value in ***obj*** is retrieved by ***casting obj*** to ***int***.

❐ ***Boxing*** and ***unboxing*** allow C#'s ***type system*** to be fully ***unified***. All types derive from ***object class***. A ***reference*** to any type can be assigned to a variable of ***type object*** (class). ***Boxing/unboxing*** *automatically handles* the details for the value types. Furthermore, because all ***types*** are derived from ***object class***, they all have ***access*** to ***object***'s methods. For example, ***Boxing*** makes it possible to ***call methods on a value***, consider the following rather surprising program:

```
using System;
class MethOnValue { static void Main() { Console.WriteLine(186.ToString()); } }
```

☛ It displays ***186***. Since ***ToString()*** returns a ***string representation*** of the ***object*** on which it is called. In this case, the string representation of ***186*** is ***186***!