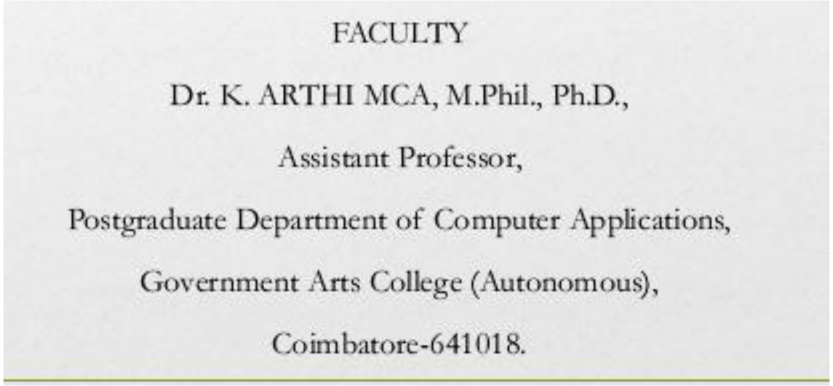
18MCA42C

.NET PROGRAMMING (C#)

UNIT II: Object Oriented Programming in C#



FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

Methods in C#

DECLARING METHODS

form of a method declaration is

```
modifiers type methodname (formal-parameter-list)
{
     method _ body
}
```

Method declaration has five parts:

- Name of the method (*methodname*)
- Type of value the method returns *(type*)
- List of parameters (*formal-parameter-list*)
- Body of the method
- Method modifiers (*modifier*)

```
int Product ( int x, int y )
{
     int m = x * y;       //operation, m is a local variable
     return(m);          // returns the result (int type)
}
```

Table 8.1 *List of method modifiers*

MODIFIER	DESCRIPTION
new	The method hides an inherited method with the same signature
public	The method can be accessed from anywhere, including outside the class
protected	The method can be accessed from within the class to which it belongs, or a type derived from that class
internal	The method can be accessed from within the same program
private	The method can only be accessed from inside the class to which it belongs
static	The method does not operate on a specific instance of the class
virtual	The method can be overridden by a derived class
abstract	A virtual method which defines the signature of the method, but doesn't provide an implementation
override	The method overrides an inherited virtual or abstract method
sealed	The method overrides an inherited virtual method, but cannot be overridden by any classes which inherit from this class. Must be used in conjunction with override
extern	The method is implemented externally, in a different language

THE MAIN METHOD -

```
public static int Main( )
Or
public static void Main( )
```

INVOKING METHODS

objectname.methodname(actual-parameter-list);

Program 8.1 | DEFINING AND INVOKING A METHOD

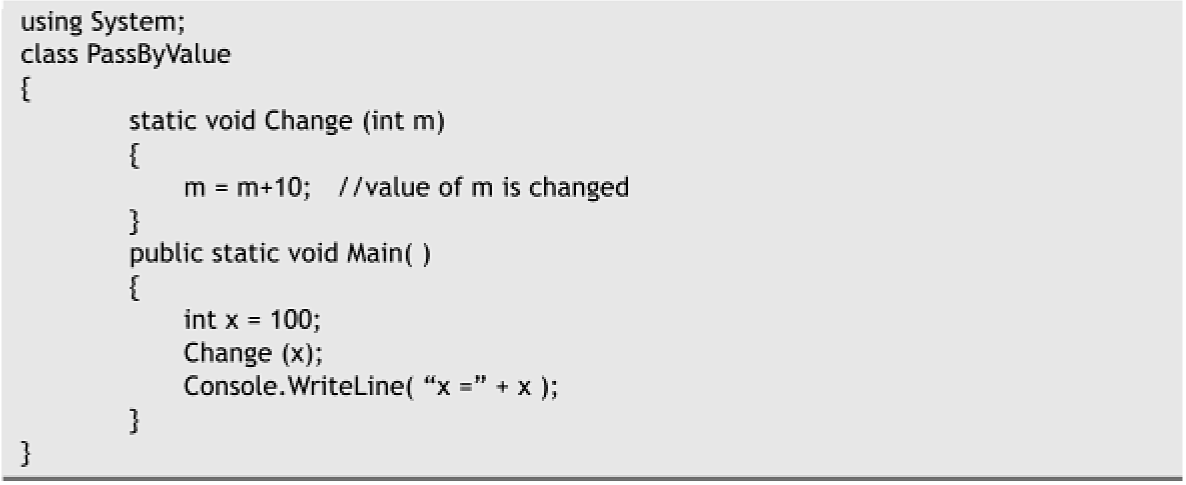
```
using System;
class Method   //   class containing the method
{
     // Define the Cube method
     int Cube ( int x )
     {
          return ( x * x * x );
     }
}
// Client class to invoke the cube method
class MethodTest
{
          public static void Main( )
          {
              // Create object for invoking cube
              Method M = new Method ( );
              // Invoke the cube method
              int y = M.Cube (5);   //Method call
              // Write the result
              Console.WriteLine( y );
        }
}
```

METHOD PARAMETERS

four kinds of parameters.

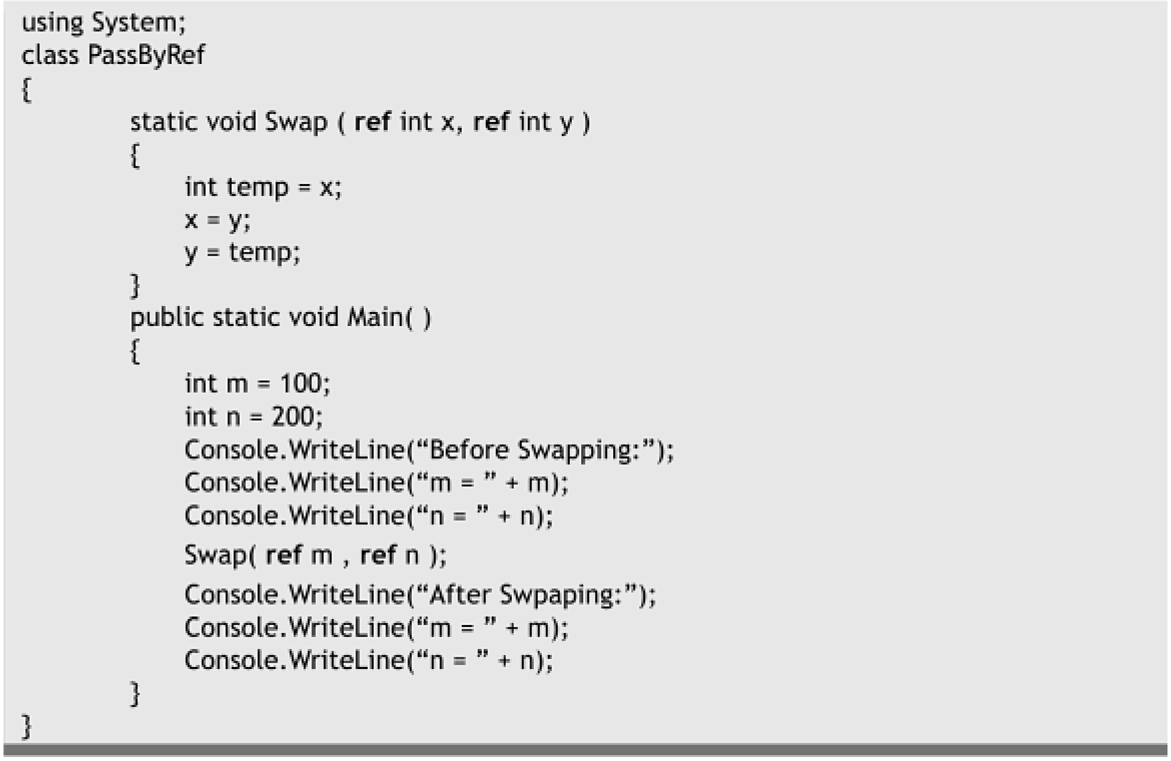
- Value parameters
- Reference parameters
- Output parameters
- Parameter arrays

Program 8.4 | ILLUSTRATION OF PASSING BY VALUE



```
using System;
class PassByValue
{
        static void Change (int m)
        {
            m = m+10;   //value of m is changed
        }
        public static void Main( )
        {
            int x = 100;
            Change (x);
            Console.WriteLine( “x =” + x );
        }
}
```

Program 8.5 | SWAPPING VALUES USING REF PARAMETERS



```
using System;
class PassByRef
{
        static void Swap ( ref int x, ref int y )
        {
            int temp = x;
            x = y;
            y = temp;
        }
        public static void Main( )
        {
            int m = 100;
            int n = 200;
            Console.WriteLine(“Before Swapping:”);
            Console.WriteLine(“m = ” + m);
            Console.WriteLine(“n = ” + n);
            Swap( ref m , ref n );
            Console.WriteLine(“After Swpaping:”);
            Console.WriteLine(“m = ” + m);
            Console.WriteLine(“n = ” + n);
        }
}
```

Classes and Objects

CLASS

```
class classname
{
          [ variables declaration; ]
          [ methods declaration; ]
}
```

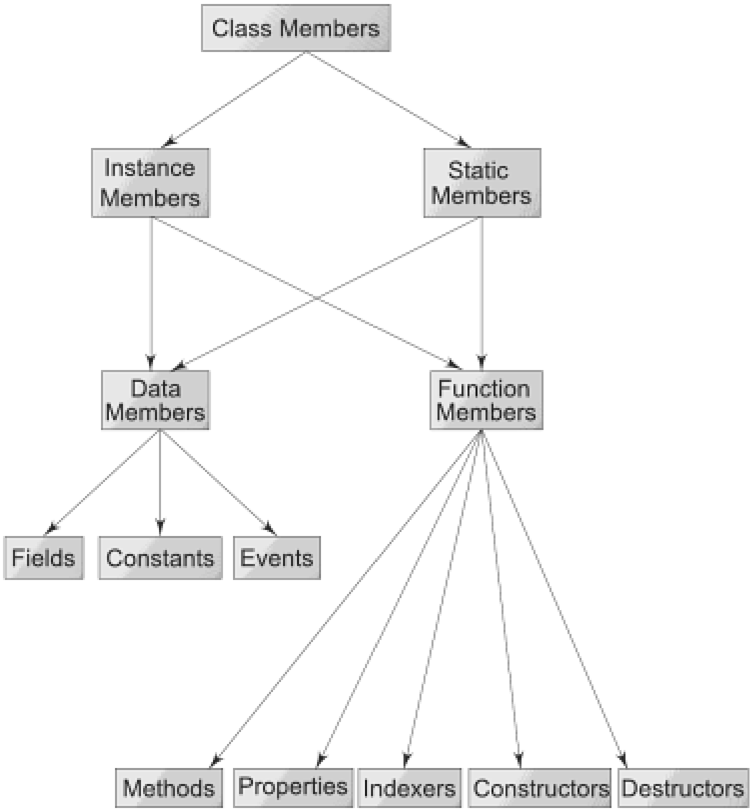


Fig. 12.2 *Categories of class members*

ADDING METHODS

```
class Rectangle
{
     int length;
     int width;
     public void GetData(int x , int y)//mutator method
     {
          length = x ;
          width = y ;
     }
}
```

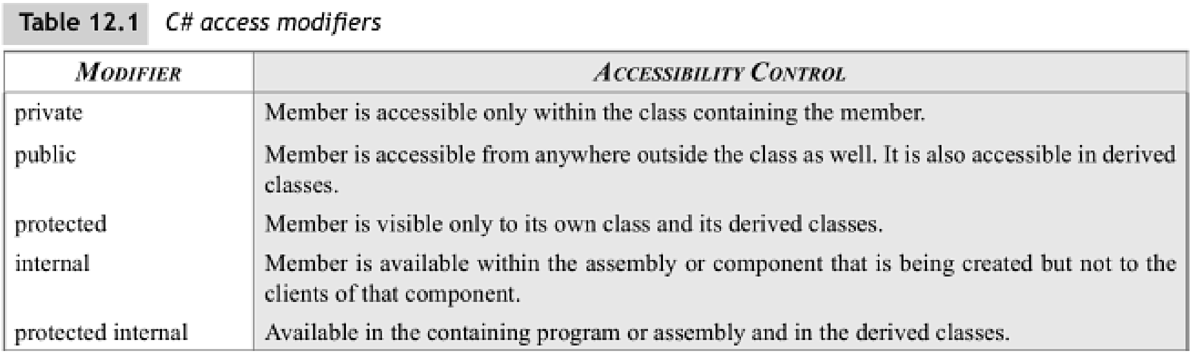
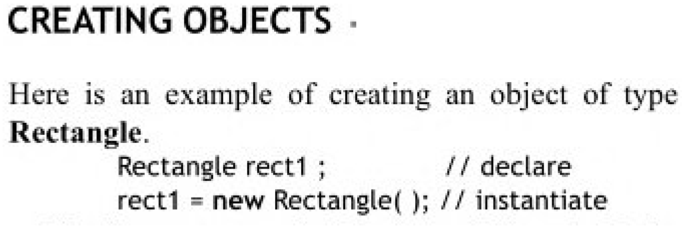


Table 12.1 *C# access modifiers*

MODIFIER	ACCESSIBILITY CONTROL
private	Member is accessible only within the class containing the member.
public	Member is accessible from anywhere outside the class as well. It is also accessible in derived classes.
protected	Member is visible only to its own class and its derived classes.
internal	Member is available within the assembly or component that is being created but not to the clients of that component.
protected internal	Available in the containing program or assembly and in the derived classes.

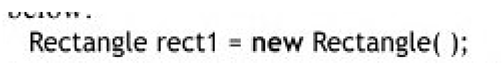


CREATING OBJECTS

Here is an example of creating an object of type **Rectangle**.

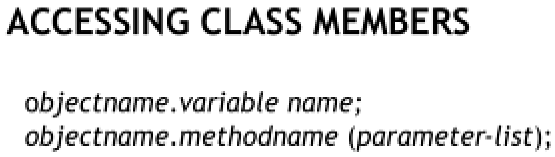
```
        Rectangle rect1 ;           // declare
        rect1 = new Rectangle( ); // instantiate
```

OR



below:

```
  Rectangle rect1 = new Rectangle( );
```



ACCESSING CLASS MEMBERS

```
  objectname.variable name;
  objectname.methodname (parameter-list);
```

Program 12.1 APPLICATION OF CLASSES AND OBJECTS

```
using System;
class Rectangle
{
    public int length, width;                //   Declaration of variables

    public void GetData(int x, int y)        //   Definition of method
    {
        length = x;
        width  = y;
    }

    public int RectArea()                    //       Definition of another method
    {
        int area = length * width;
        return (area);
    }
}

class RectArea                               //   class with main method
{
    public static void Main( )
    {
        int area1,area2;                     //       Local variables
        Rectangle rect1 = new Rectangle();      // Creating objects
        Rectangle rect2 = new Rectangle();

        rect1.length = 15;                   //   Accessing variables
        rect1.width  = 10;
        area1 = rect1.length * rect1.width;

        rect2.GetData(20,12);                //   Accessing methods
        area2 = rect2.RectArea();

        Console.WriteLine("Area1 = " + area1);
        Console.WriteLine("Area2 = " + area2);
    }
}
```

Inheritance

C# classes can be reused in several ways. Reusability is achieved by designing new classes, reusing all or some of the properties of existing ones. The mechanism of designing or constructing one class from another is called *inheritance*. This may be achieved in two different forms.

- Classical form
- Containment form

CLASSICAL INHERITANCE

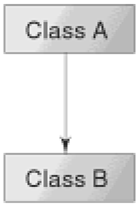


Fig.13.1 *Simple inheritance*

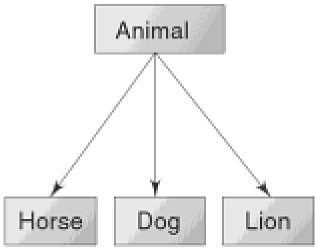
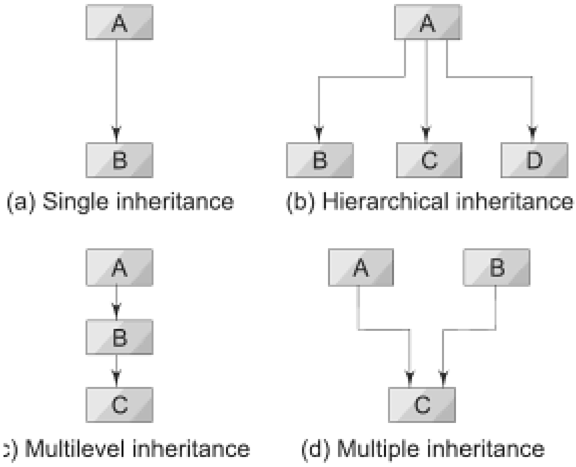


Fig.13.2 *The is-a resistance*



(a) Single inheritance

(b) Hierarchical inheritance

c) Multilevel inheritance

(d) Multiple inheritance

Fig.13.3 *Implementation of inheritance*

- Single inheritance (only one base class)
- Multiple inheritance (several base classes)
- Hierarchical inheritance (one base class, many subclasses)
- Multilevel inheritance (derived from a derived class)

CONTAINMENT INHERITANCE

```
class A
{
      . . . .
}
class B
{
      . . . .
      A a; // a is contained in b
}
B b;
. . . .
```

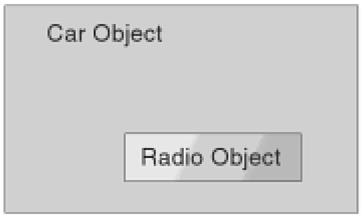


Fig.13.4 *The has-a relationship*

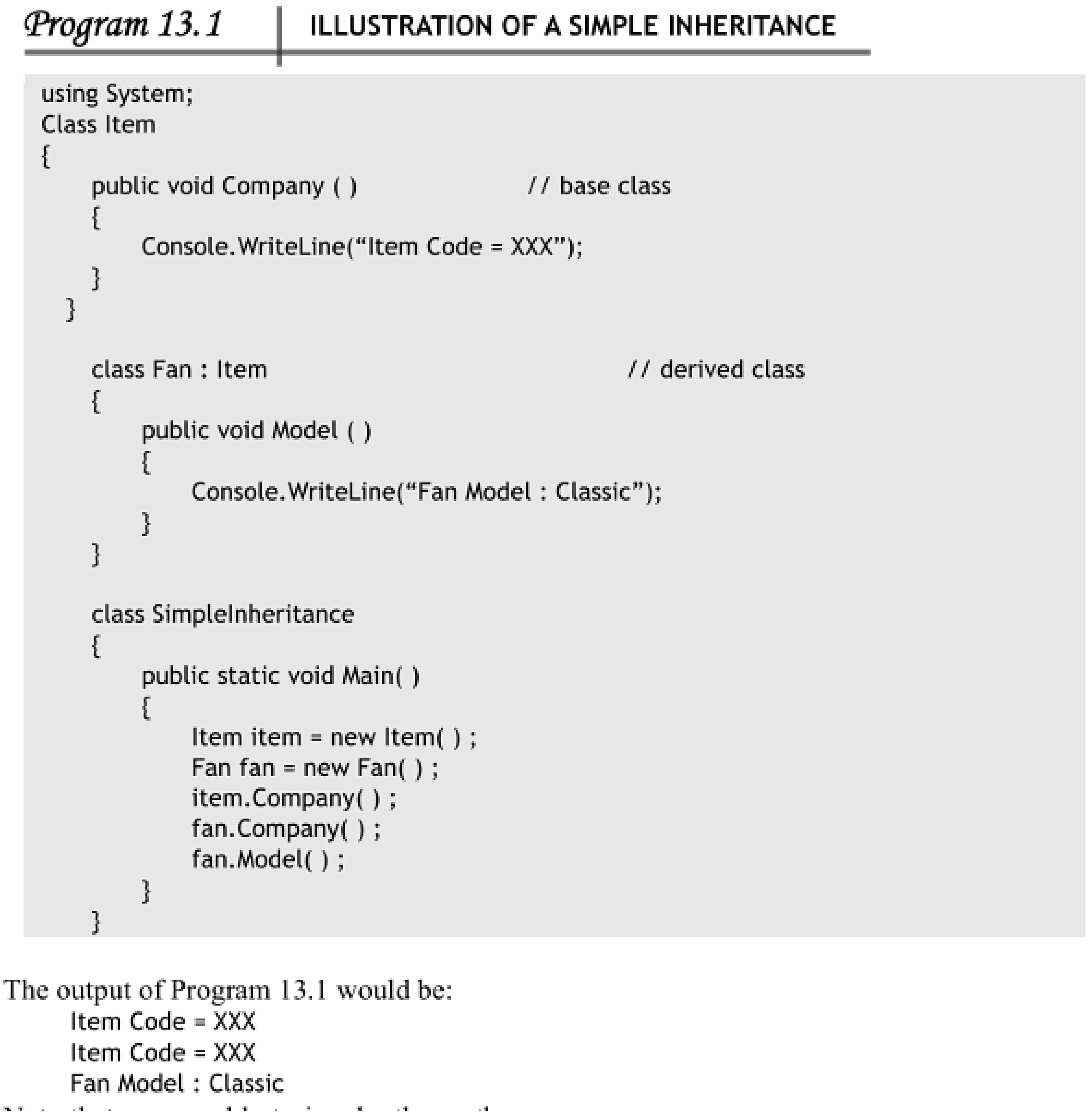
DEFINING A SUBCLASS

A subclass is defined as follows:

```
          Class subclass-name : baseclass-name

          {
                variables declaration ;


      methods declaration ;
 }
```



Program 13.1 ILLUSTRATION OF A SIMPLE INHERITANCE

```
using System;
Class Item
{
     public void Company ( )                  // base class
     {
          Console.WriteLine(“Item Code = XXX”);
     }
  }

     class Fan : Item                                   // derived class
     {
          public void Model ( )
          {
               Console.WriteLine(“Fan Model : Classic”);
          }
     }

     class SimpleInheritance
     {
          public static void Main( )
          {
               Item item = new Item( ) ;
               Fan fan = new Fan( ) ;
               item.Company( ) ;
               fan.Company( ) ;
               fan.Model( ) ;
          }
     }
```

The output of Program 13.1 would be:

```
Item Code = XXX
Item Code = XXX
Fan Model : Classic
```

Some important characteristics of inheritance are:

- A derived class extends its direct base class. It can add new members to those it inherits. However, it cannot change or remove the definition of an inherited member.
- Constructors and destructors are not inherited. All other members, regardless of their declared accessibility in base class, are inherited. However, their accessibility in the derived class depends on their declared accessibility in the base class.
- An instance of a class contains a copy of all instance fields declared in the class and its base classes.
- A derived class can hide an inherited member
- A derived class can override an inherited member

Table 13.1 *Visibility of class members*

Keyword	*Visibility*			
	Containing classes	*Derived classes*	*Containing program*	*Anywhere outside the containing program*
Private	✓			
protected	✓	✓		
Internal	✓		✓	
protected internal	✓	✓	✓	
Public	✓	✓	✓	✓

Program 13.2 APPLICATION OF SINGLE INHERITANCE

```
using System;
class Room                    // base class
{
    public int length;
    public int breadth;
    public Room (int x , int y)      // base constructor
    {
        length          =       x;
        breadth         =       y;
    }
    public int Area ( )
    {
        return (length * breadth );
    }
}
class BedRoom : Room    //Inheriting Room
{
    int height;
                        //subclass constructor
    public Bedroom (int x, int y, int z):base (x,y)
    {

        height = z;
    }
    public int Volume ( )
    {
        return (length * breadth * height);
    }
}
class InherTest
{
    public static void Main( )
    {
        BedRoom room1 = new BedRoom (14, 12, 10);
        int areal = room1.Area ( );         // superclass method
    int volume1 = room1.Volume ( );         // subclass method
        Console.WriteLine("Area1 =   " + areal1);
        Console.WriteLine("Volume1 = " + volume1);
    }
}
```

MULTILEVEL INHERITANCE

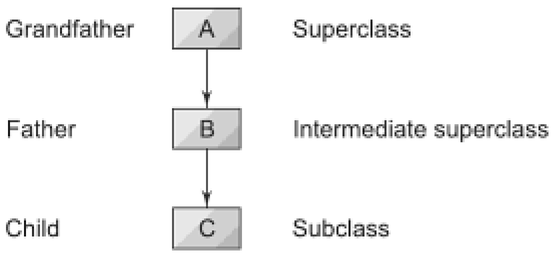


Fig. 13.5 *Multilevel inheritance*

A derived class with multilevel base classes is declared as follows:

```
class A
{
     . . . .
     . . . .
}
class B : A // First level derivation
{
     . . . .
     . . . .
}
class C : B // Second level derivation
{


     . . . .
}
```

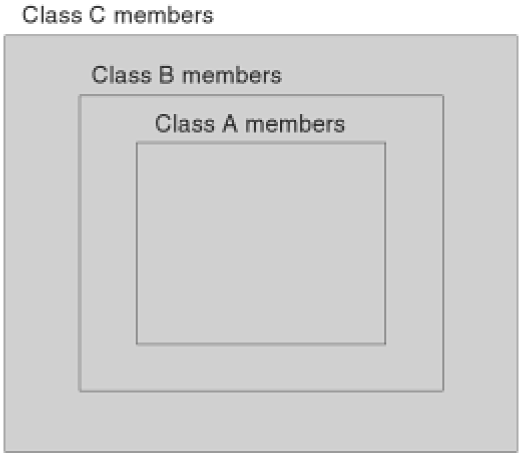
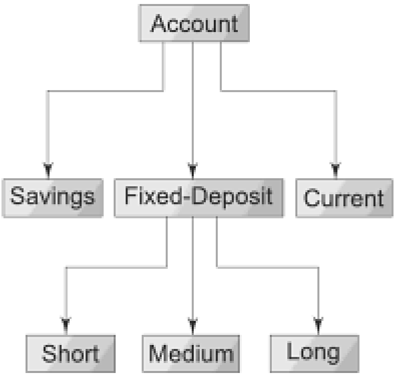


Fig. 13.6 *C contains B which contains A*



13.7 *Hierarchical classification of bank accounts*

ABSTRACT CLASSES

The **abstract** is a modifier and when used to declare a class indicates that the class cannot be instantiated. Only its derived classes (that are not marked abstract) can be instantiated. *Example:*

```
abstract class Base
{
    . . . .
}

class Derived : Base
{
    . . . .
}
. . . .
. . . .
. . . .
Base b1;          //Error
Derived d1;       //OK
```

We cannot create objects of **Base** type but we can derive its subclasses which can be instantiated. Some characteristics of an abstract class are:

- It cannot be instantiated directly
- It can have abstract members
- We cannot apply a **sealed** modifier to it

13.13 SEALED CLASSES: PREVENTING INHERITANCE

Sometimes, we may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called a *sealed class*. This is achieved in C# using the modifier **sealed** as follows:

```
sealed class Aclass
{
    . . . .
    . . . .
}
sealed class Bclass: Someclass
{
    . . . .
    . . . .
}
```

Any attempt to inherit these classes will cause an error and the compiler will not allow it.

Declaring a class **sealed** prevents any unwanted extensions to the class. It also allows the compiler to perform some optimizations when a method of a sealed class is invoked. Usually standalone utility classes are created as sealed classes.

A **sealed** class cannot also be an **abstract** class.

Interface: Multiple Inheritance

An interface in C# is a reference type. It is basically a kind of class with some differences. Major differences include:

- All the members of an interface are implicitly **public** and **abstract.**
- An interface cannot contain constant fields, constructors and destructors.
- Its members cannot be declared **static.**
- Since the methods in an interface are abstract, they do not include implementation code.
- An interface can inherit multiple interfaces.

DEFINING AN INTERFACE -

The syntax for defining an interface is very similar to that used for defining a class. The general form of an interface definition is:

```
interface          InterfaceName
{

        Member declarations;
  }
```

Here**, interface** is the keyword and *InterfaceName* is a valid C# identifier (just like class names).

EXTENDING AN INTERFACE

```
interface      name2 : name1
{

      Members of name2
}
```

IMPLEMENTING INTERFACES -

```
class classname : interfacename
{
      body of classname
}
```

Here the class **classname '**implements' the interface **interfacename.** A more general form of implementation may look like this:

```
class classname : superclass, interface1, interface2. . . .
{
      body of classname
}
```

Program 14.1 | IMPLEMENTATION OF MULTIPLE INTERFACES

```
using System;
interface Addition
{
      int Add ( );
}
interface Multiplication
{
      int Mul ( );
}
class Computation  : Addition, Multiplication
{
      int x, y;
      public Computation (int x, int y )                    //Constructor
      {
                  this.x = x;
                  this.y = y;
      }
      public int Add ( )                                    //Implement Add ( )
      {
                  return ( x + y );
      }
      public int Mul ( )                                    //Implement Mul ( )
      {
                  return ( x * y );
      }
}
class InterfaceTest1
{
      public static void Main( )
      {
                  Computation com = new Computation (10,20);
                  Addition add = (Addition ) com;                        // casting
                  Console.WriteLine (“Sum = ” + add.Add ( ));
                  Multiplication mul = (Multiplication) com;             // casting
                  Console.WriteLine(“Product = ” + mul.Mul ( ) );
      }
}
```

Delegates

A delegate object is a special type of object that contains the details of a method rather than data. Delegates in C# are used for two purposes:

- Callback
- Event handling

The dictionary meaning of **delegate** is “a person acting for another person”. In C#, it really means a method acting for another method. As pointed out earlier, a delegate in C# is a class type object and is used to invoke a method that has been encapsulated into it at the time of its creation. Creating and using delegates involve four steps. They include:

- Delegate declaration
- Delegate methods definition
- Delegate instantiation
- Delegate invocation

DELEGATE DECLARATION

A delegate declaration is a type declaration and takes the following general form:

```
        modifier delegate return-type delegate-name ( parameters);
```

delegate is the keyword that signifies that the declaration represents a class type derived from **System. Delegate**. The *return-type* indicates the return type of the delegate. *Parameters* identifies the signature of the delegate. The *delegate-name* is any valid C# identifier and is the name of the delegate that will be used to instantiate delegate objects.

The *modifier* controls the accessibility of the delegate. It is optional. Depending upon the context in which they are declared, delegates may take any of the following modifiers:

- **new**
- **protected**
- **private**
- **public**
- **internal**

The **new** modifier is only permitted on delegates declared within another type. It signifies that the delegate hides an inherited member by the same name.

Some examples of delegates are:

```
        delegate void SimpleDelegate( );
        delegate int MathOperation(int x, int y);
        public delegate int CompareItems(object o1, object o2);
        private delegate string GetAString( );
        delegate double DoubleOperation(double x);
```

DELEGATE METHODS

The methods whose references are encapsulated into a delegate instance are known as *delegate methods* or *callable entities*. The signature and return type of delegate methods must exactly match the signature and return type of the delegate.

DELEGATE INSTANTIATION

Although delegates are of class types and behave like classes, C# provides a special syntax for instantiating their instances. A *delegate-creation-expression* is used to create a new instance of a delegate.

```
new delegate-type (expression)
```

DELEGATE INVOCATION -

C# uses a special syntax for invoking a delegate. When a delegate is invoked, it in turn invokes the method whose reference has been encapsulated into the delegate, (only if their signatures match). Invocation takes the following form:

```
delegate_object (parameters list )
```

The optional *parameters list* provides values for the parameters of the method to be used.

- If the invocation invokes a method that returns **void,** the result is nothing and therefore it cannot be used as an operand of any operator. It can be simply a statement_expression. *Example:*

```
delegate1(x, y); //void delegate
```

This delegate invokes a method that does not return any value.

- If the method returns a value, then it can be used as an operand of any operator. Usually, we assign the return value to an appropriate variable for further processing. *Example:*

```
double result = delegate2(2.56, 45.73);
```

This statement invokes a method (that takes two **double** values as parameters and returns **double** type value) and then assigns the returned value to the variable **result**.

Program 16.1 | CREATING AND IMPLEMENTING A DELEGATE

```
using System;
//delegate declaration
delegate int ArithOp(int x, int y);
class MathOperation
{
    //delegate methods definition
    public static int Add(int a, int b)
    {
        return (a + b);
    }
```

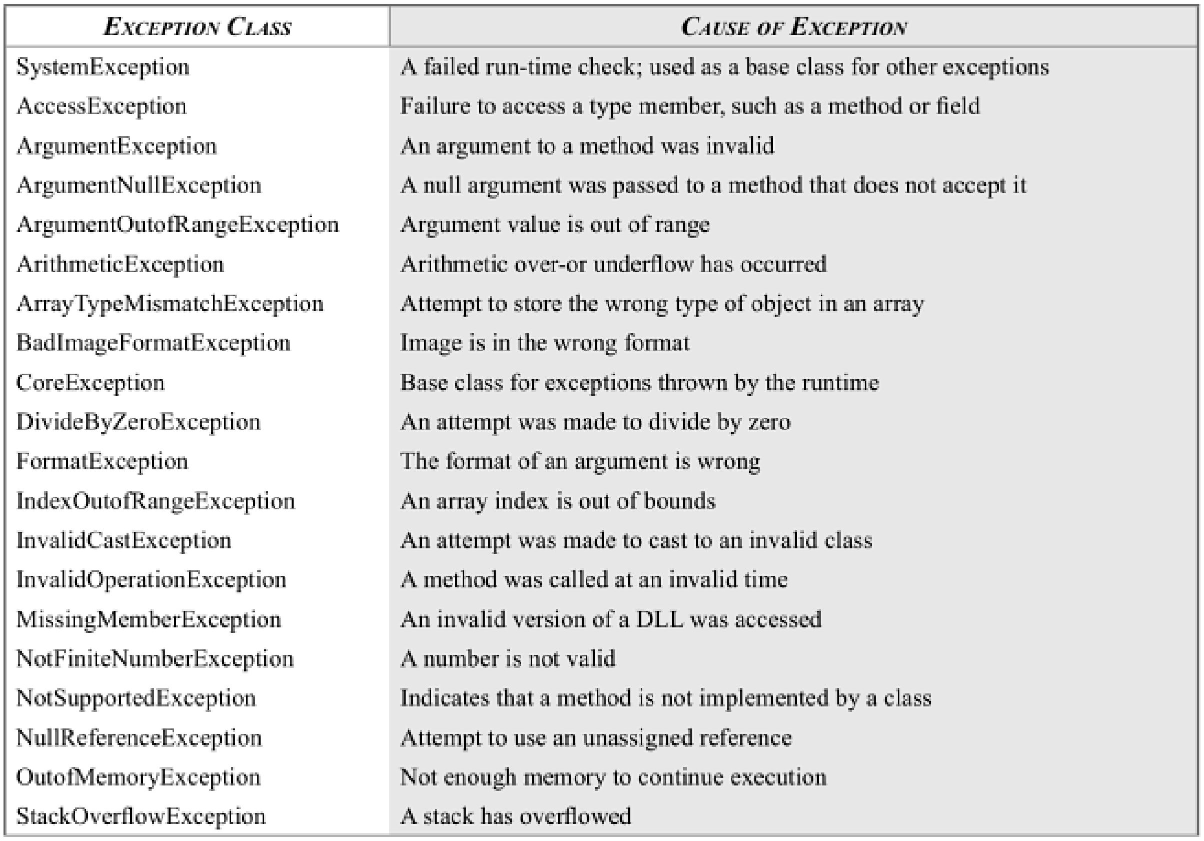
```
        public static int Sub(int a, int b)
    {
        return (a - b);
    }
    }
        class DelegateTest
    {
        public static void Main( )
    {
        //delegate instances
        ArithOp operation1 = new ArithOp (MathOperation.Add);
        ArithOp operation2 = new ArithOp(MathOperation.Sub);
        //invoking delegates
        int result1 = operation1(200, 100);
        int result2 = operation2(200,100);
        Console.WriteLine("Result1 = " + result1);
        Console.WriteLine("Result2 = " + result2);
    }
}
```

EXCEPTIONS

An *exception* is a condition that is caused by a run-time error in the program. When the C# compiler encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred).

- Find the problem (***Hit*** the exception)
- Inform that an error has occurred (***Throw*** the exception)
- Receive the error information (***Catch*** the exception)
- Take corrective actions (***Handle*** the exception)

Table 18.1 *Common C# exceptions*



Exception Class	*Cause of Exception*
SystemException	A failed run-time check; used as a base class for other exceptions
AccessException	Failure to access a type member, such as a method or field
ArgumentException	An argument to a method was invalid
ArgumentNullException	A null argument was passed to a method that does not accept it
ArgumentOutofRangeException	Argument value is out of range
ArithmeticException	Arithmetic over-or underflow has occurred
ArrayTypeMismatchException	Attempt to store the wrong type of object in an array
BadImageFormatException	Image is in the wrong format
CoreException	Base class for exceptions thrown by the runtime
DivideByZeroException	An attempt was made to divide by zero
FormatException	The format of an argument is wrong
IndexOutofRangeException	An array index is out of bounds
InvalidCastException	An attempt was made to cast to an invalid class
InvalidOperationException	A method was called at an invalid time
MissingMemberException	An invalid version of a DLL was accessed
NotFiniteNumberException	A number is not valid
NotSupportedException	Indicates that a method is not implemented by a class
NullReferenceException	Attempt to use an unassigned reference
OutofMemoryException	Not enough memory to continue execution
StackOverflowException	A stack has overflowed

SYNTAX OF EXCEPTION HANDLING CODE

```
..........
..........
try
{
     statement;              // generates an exception
}
catch (Exception e)
{
     statement;              // processes the exception
}
..........
..........
```

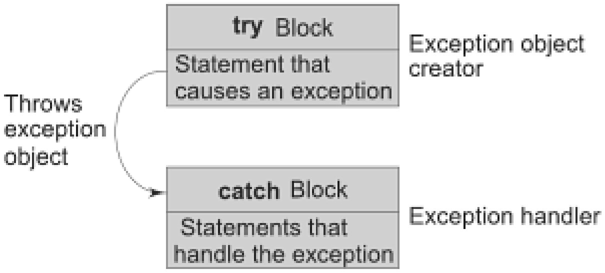


Fig. 18.1 *Exception handling mechanism*

Program 18.3 | USING TRY AND CATCH FOR EXCEPTION HANDLING

```
using System;
class Error3
{
     public static void Main( )
     {
          int a = 10;
          int b = 5;
          int c = 5;
          int x, y ;
          try
          {
               x = a / (b-c);          / / Exception here
          }
               catch (Exception e)
          {
               Console.WriteLine("Division by zero");
          }
          y = a / (b+c);
          Console.WriteLine("y = " + y);
     }
 }
```

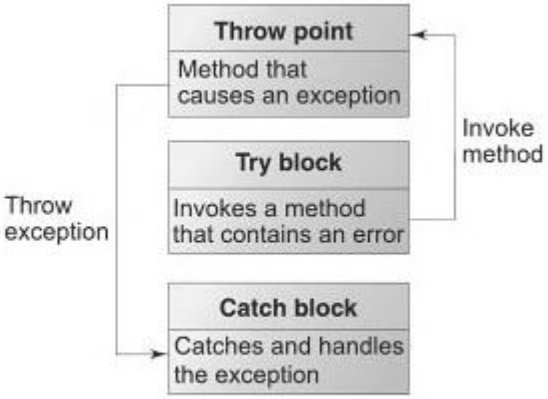

Throw point
Method that causes an exception
Try block
Invokes a method that contains an error
Catch block
Catches and handles the exception
Invoke method
Throw exception

Fig. 18.2 *Invoking a method that contain exceptions*

MULTIPLE CATCH STATEMENTS

```
...........
...........
try
{
     statement ;              / / generates an exception
}
catch (Exception-Type-1 e)
{
     statement;               / / processes exception type 1
}
catch (Exception-Type-2 e)
{
     statement;               / / processes exception type 2
}
.
.
.
catch (Exception-type-N e)
{
     statement ;              / / processes exception type N
}
```

Program 18.4 USING MULTIPLE CATCH BLOCKS

```
using System;
class Error4
{
    public static void Main( )
    {
        int [ ] a = {5,10};
        int b = 5;
        try
        {
            int x = a[2] / b - a[1];
        }
        catch(ArithmeticException e)
        {
            Console.WriteLine(“Division by zero”);
        }
        catch(IndexOutOfRangeException e)
        {
            Console.WriteLine(“Array index error”);
        }
        catch(ArrayTypeMismatchException e)
        {
            Console.WriteLine(“Wrong data type”);
        }
        int y = a[1] / a[0];
        Console.WriteLine(“y = ” + y);
    }
 }
```

THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.