Q.2 a. Briefly explain the advantage of framework base classes in .NET. (5)

Answer:

.NET supplies a library of base classes that we can use to implement applications quickly. We can use them by simply instantiating them and invoking their methods or by inheriting them through derived classes, thus extending their functionality.

Much of the functionality in the base framework classes resides in the vast namespace called system. We can use the base classes in the system namespace for many different tasks including

- Input/output operations
- String handling
- Managing arrays, list etc
- Accessing files and file system
- Security
- Windowing
- Window messages
- Database management
- Evaluation of mathematical functions
- Drawing
- Managing errors and exceptions
- Connecting to internet
- And many more

b. What is the purpose of 'using' directive in C# program? Explain with example. (5)

Answer:

C# supports a feature known as using directive that can be used to import the namespaces in the program. Once the namespace is imported, we can use the elements of that namespace without using the namespace as prefix.

For example:

```
using System; //System is a namespace
Class Myclass
{
        // main method begins
        {
                Console.WriteLine("hello");
        }
        // main method ends
}
```

The first statement in the program is

using system;

This tells the compiler to look in the **System** library for unresolved class names. There is no need to use **System** prefix to the **Console** class in the output line.

When the compiler parses the **Console.WriteLine** method, it will understand that the method is undefined. However, it will then search through the namespaces specified in **using** directives and upon finding the method in **System** namespace, will compile the code without any complaint.

c. With a suitable example explain the concept of boxing and unboxing. (6)

Answer:

In object-oriented programming, methods are invoked using objects. Since value types such as **int** and **long** are not objects, we cannot use them to call methods. C# enables us to achieve this through a technique known as *boxing* . Boxing means the conversion of a value type on the stack to a **object** type on the heap. Conversely, the conversion from an **object** type back to a value type is known as *unboxing.*

4.12.1 Boxing

Any type, value or reference can be assigned to an object without an explicit conversion. When the compiler finds a value type where it needs a reference type, it creates an object 'box' into which it places the value of the value type. The following code illustrates this:

```
int m = 100;
object om = m; // creates a box to hold m
```

When executed, this code creates a temporary reference_type 'box' for the object on heap. We can also use a C-style cast for boxing.

```
int m = 100;
object om = (object)m; //C-style casting
```

Note that the boxing operation creates a copy of the value of the **m** integer to the object **om**. Now both the variables **m** and **om** exist but the value of **om** resides on the heap. This means that the values are independent of each other. Consider the following code:

```
int m = 10:
object om = m;
m = 20;
Console.WriteLine(m);  // m = 20
Console.WriteLine(om); //om = 10
```

When a code changes the value of **m**, the value of **om** is not affected.

4.12.2 Unboxing

Unboxing is the process of converting the object type back to the value type. Remember that we can only unbox a variable that has previously been boxed. In contrast to boxing, unboxing is an explicit operation using C-style casting.

```
int m = 10;
object om = m;    //box m
int n = (int)om;  //unbox om back to an int
```

When performing unboxing, C# checks that the value type we request is actually stored in the object under conversion. Only if it is, the value is unboxed.

When unboxing a value, we have to ensure that the value type is large enough to hold the value of the object. Otherwise, the operation may result in a runtime error. For example, the code

```
int m = 500;
object om = m;
byte n = (byte)om;
```

will produce a runtime error.

Notice that when unboxing, we need to use explicit cast. This is because in the case of unboxing, an object could be cast to any type. Therefore, the cast is necessary for the compiler to verify that it is valid as per the specified value type.

Q.3 a. Explain how a conditional operator evaluates an expression in C# to make decision? Also, compare it with if-else statements. (8)

Answer:

C# has an unusual operator, useful for making two-way decisions; it is a combination of ? and :, and takes three operands. This operator is popularly known *as the conditional operator*. The general form of use of the *conditional operator* is as follows:

```
conditional expression ? expression1 : expression2
```

The *conditional expression* is evaluated first. If the result is true, *expression 1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```
if (x < 0)
    flag = 0;
else
    flag = 1;
```

can be written as

```
flag = (x<0) ? 0 : 1;
```

Consider the evaluation of the following function:

```
y = 1.5x + 3     for x <= 2
y = 2x + 5       for x > 2
```

This can be evaluated using the conditional operator as follows:

```
y = (x > 2) ? (2*x + 5) : (1.5*x + 3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

```
         {    4x + 100     for x < 40
salary = {    300          for x = 40
         {    4.5x + 150   for x > 40
```

This complex equation can be written as

```
salary = (x!=40) ? ((x<40)?(4*x+100):(4.5*x+150)) : 300;  // nesting
```

The same can be evaluated using **if....else** statements as follows:

```
if (x<=40)
    if (x<40)
        salary = 4*x+100;
    else
        salary = 300;
else
        salary = 4.5*x+150;
```

When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use **if** statements when more than a single nesting of the conditional operator is required.

b. How does foreach statement differs from for loop? Using for each statement, write a program in C# to print all values in an integer array. (8)

Answer:

The **foreach** statement is similar to the **for** statement but implemented differently. It enables us to iterate the elements in arrays and collection classes such as **List** and **HashTable**. The general form of the **foreach** statement is:

```
foreach (type variable in expression)
{
    Body of the loop
}
```

The *type* and *variable* declare the *iteration* variable. During execution, the iteration variable represents the array element (or collection element in case of collections) for which an iteration is currently being performed. **in** is a keyword.

The *expression* must be an *array* or *collection* type and an explicit conversion must exist from the element type of the collection to the type of the iteration variable. *Example:*

```
public static void Main (string [ ] args)
{
    foreach ( string s in args)
    {
        Console.WriteLine(s);
    }
}
```

Program 7.6 illustrates the use of **foreach** statement for printing the contents of a numerical array.

Program 7.6 PRINTING ARRAY VALUES USING FOREACH STATEMENT

```
using System;
class ForeachTest
{
        public static void Main ( )
        {
             int[ ] arrayInt = { 11, 22, 33, 44 };
             foreach ( int m in arrayInt)
             {
                  Console.Write(" " + m);
             }
                  Console.WriteLine( );
        }
}
```

Program 7.6 will display the following output:

```
11  22  33  44
```

An important point to note is that we cannot change the value of the iteration variable during execution. For instance, the code

```
int[ ] x = { 1, 2, 3 };
     foreach ( int i in x )
     {
          i++;
          Console.Write(i);
     }
```

will not work. If we need to change the values of an item during the iteration process, we may use the **for** loop construct.

The advantage of **foreach** over the **for** statement is that it automatically detects the boundaries of the collection being iterated over. Further, the syntax includes a built-in iterator for accessing the current element in the collection.

Q.4 a. What is method overloading? Demonstrate it with the help of C# code. (8)

Answer:

C# allows us to create more than one method with the same name, but with the different parameter lists and different definitions. This is called *method overloading*. Method overloading is used when methods are required to perform similar tasks but using different input parameters.

Overloaded methods must differ in number and/or type of parameters they take. This enables the compiler to decide which one of the definitions to execute depending on the type and number of arguments in the method call. Note that the method's return type does not play any role in the overload resolution.

Using the concept of method overloading, we can design a family of methods with one name but different argument lists. For example, an overloaded add( ) method handles different types of data as shown below:

```
// Method definitions
int add ( int a, int b ) { ... }                //Method1
int add ( int a, int b, int c ) { ... }         //Method2
double add ( float x, float y ) { ... }         //Method3
double add ( int p, float q ) { ... }           //Method4
double add (float p, int q ) { ... }            //Method5

//Method calls
int m = add ( 5, 10 ) ;     //calls method1
double x = add ( 15, 5.0F );                    //calls method4
double x = add ( 1.0F, 2.0 F );                 //calls method3
int m = add ( 5, 10, 15 ) ;                     //calls method2
double x = add ( 2.0F, 10 );                    //calls method5
```

The method selection involves the following steps:

1. The compiler tries to find an exact match in which the types of actual parameters are the same and uses that method.
2. If the exact match is not found, then the compiler tries to use the implicit conversions to the actual arguments and then uses the method whose match is unique. If the conversion creates multiple matches, then the compiler will generate an error message.

```
using System;
class Overloading
{
        public static void Main( )
        {
            Console.WriteLine(volume (10));
            Console.WriteLine(volume(2.5 F, 8));
            Console.WriteLine(volume(100L,75,15));
        }
        static int volume ( int x )                         // cube
        {
            return (  x * x * x  ) ;
        }
        static double volume ( float r , int h )            // cylinder
        {
            return ( 3.14519 * r * r * h ) ;
        }
            static long volume ( long l , int b , int h )   // box
        {
            return ( l * b * h ) ;
        }
}
```

The output of Program 8.9 would be:

```
1000
157.2595
112500
```

b. What are jagged arrays? Write a program in C# to sort a list of numbers. **(3+5)**

Answer:

C# treats multidimensional arrays as 'arrays of arrays'. It is possible to declare a two-dimensional array as follows:

```
int[ ][ ] x= new int[3][];      //three rows array
x[0]    = new int[2];           //first row has two elements
x[1]    = new int[4];           //second row has four elements
x[2]    = new int[3];           //third row has three elements
```

These statements create a two-dimensional array having different lengths for each row as shown in Fig. 9.3. Variable-size arrays are called *jagged* arrays.

The elements can be accessed as follows:

```
x [1] [1] = 10;
int y = x [2[[2];
```

Note the difference in the way we access the two types of arrays. With rectangular arrays, all indices are within one set of square brackets, while for jagged arrays each element is within its own square brackets.

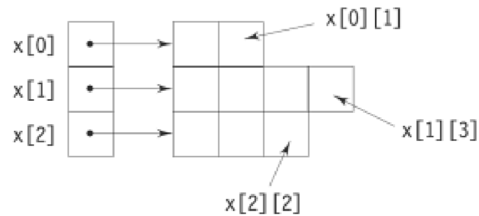

Fig. 9.3 *Variable size arrays*

```
using System;
class NumberSorting
{
    public static void Main( )
    {
        int[ ]number = { 55, 40, 80, 65, 71 };
        int n = number.Length;
        Console.Write("Given list : ");
        for (int i = 0; i < n; i++)
        {
            Console.Write("  " + number[i]);
        }
        Console.WriteLine("\n");
        // Sorting begins
        for (int i = 0; i < n; i++)
        {
            for (int j = i+1; j < n; j++)
            {
                if (number[i] < number[j])
                {
                    // Interchange values
                    int temp = number[i];
                    number[i] = number[j];
                    number[j] = temp;
                }
            }
        }
        Console.Write("Sorted list : ");
        for (int i = 0; i < n; i++)
        {
            Console.Write("  " + number[i]);
        }
        Console.WriteLine("  ");
    }
}
```

Program 9.1 displays the following output:

```
Given list     : 55 40 80 65 71
Sorted list    : 80 71 65 55 40
```

Q.5 a. What are mutable strings? Explain the following methods and properties of StringBuilder class. (8)
(i) Append( )
(ii) Insert( )
(iii) Replace( )
(iv) Remove( )
(v) Capacity
(vi) Length

Answer:

Mutable strings that are modifiable can be created using the **StringBuilder** class. *Examples:*

```
StringBuilder str1 = new StringBuilder("abc");
v   StringBuilder str2 = new StringBuilder ( );
```

The string object **str1** is created with an initial size of three characters and **str2** is created as an empty string. They can grow dynamically as more characters are added to them. They can grow either unbounded or up to a configurable maximum. Mutable strings are also known as *dynamic strings*.

The **StringBuilder** class supports many methods that are useful for manipulating dynamic strings. The **System.Text** namespace contains the **StringBuilder** class and therefore we must include the **using System.Text** directive for creating and manipulating mutable strings.

Append( ) – Appends a string
Insert( ) – Inserts a string at a specified position
Replace( ) – replaces all instances of a character with a specified one
Remove( ) – removes the specified characters
Capacity – To retrieve or set the number of characters the object can hold
Length – To retrieve or set the length

b. What do you mean by a structure? How a structure is declared in C#? Illustrate nesting of structures with an example. (8)

Answer:

Structures (often referred to as *structs*) are similar to classes in C#. Although classes will be used to implement most objects, it is desirable to use structs where simple composite data types are required. Because they are value types stored on the stack, they have the following advantages compared to class objects stored on the heap:

- They are created much more quickly than heap-allocated types.
- They are instantly and automatically deallocated once they go out of scope.
- It is easy to copy value type variables on the stack.

The performance of programs may be enhanced by judicious use of structs.

A struct in C# provides a unique way of packing together data of different types. It is a convenient tool for handling a group of logically related data items. It creates a *template* that may be used to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations.

Structs are declared using the **struct** keyword. The simple form of a struct definition is as follows:

```
struct struct-name
{
    data member1;
    data member2;
```

```
        . . .
        . . .
}
```

Example:

```
struct Student
{
    public string Name;
    public int RollNumber;
    public double TotalMarks;
}
```

The keyword **struct** declares **Student** as a new data type that can hold three variables of different data Types. These variables are known as *members* or *fields* or *elements*. The identifier **Student** can now be used to create variables of type **Student**. *Example:*

```
Student s1 ;    //declare a student
```

s1 is a variable of type **Student** and has three member variables as defined by the template.

C# permits declaration of structs nested inside other structs. The following code is valid:

```
struct Employee
{
    public string name;
    public int code;
    public struct Salary
    {
        public double basic;
        public double allowance;
    }
}
```

We can also use struct variables as members of another struct. This implies nesting of references to structs.

```
struct M
{
    public int x;
}
    struct N
{
    public M m;   // object of M
    public int y;
}
. . .
. . .
N n;
n.m.x = 100;     // x is a member of m, a member of n
n.y = 200;       // y is a member of n
```

Q.6 a. What do you mean by a property? What are various features of property? Why they are referred to as smart fields? (8)

Answer:

One of the design goals of object-oriented systems is not to permit any direct access to data members, because of the implications of integrity. It is normal practice to provide special methods known as *accessor methods* to have access to data members. We must use only these methods to set or retrieve the values of these members. Recall that we have used a method **GetData( )** in Program 12.1 to provide values to the data members **length** and **breadth** of **Rectangle** class. Similarly, we could use another method to read the values of these members. Program 12.5 shows how accessor methods can be used to set and get the value of a private data member.

Program 12.5 | **ACCESSING PRIVATE DATA USING ACCESSOR METHODS**

```
using System;
class Number
{
    private int number;
    public void SetNumber( int x )          //accessor method
    {
        number = x;                         //private number accessible
    }
    public int GetNumber( )                 //accessor method
    {
        return number;
    }
}

class NumberTest
{
    public static void Main ( )
    {
        Number n = new Number ( );
        n.SetNumber (100);      // set value
        Console.WriteLine("Number = " + n.GerNumber( )); // get value
        // n.number; //Error! Cannot access private data
    }
}
```

Output of Program 12.5:

```
Number = 100
```

The **SetNumbers** method is also known as the *mutator* method. Using accessor methods works well and is a technique used by several OOP languages, including C++ and Java. However, it suffers from the following drawbacks:

- We have to code the accessor methods manually.
- Users have to remember that they have to use accessor methods to work with data members.

In order to overcome these problems, C# provides a mechanism known as *properties* that has the same capabilities as accessor methods, but is much more elegant and simple to use. Using a property, a programmer can get access to data members as though they are public fields. (Properties are sometimes referred to as 'smart fields' as they add smartness to data fields.)

Program 12.6 IMPLEMENTING A PROPERTY

```
using System;
class Number
{
    private int number;
    public int Anumber // property
    {
        get
        {
            return number;
        }
        set
        {
            number = value;
        }
    }
}
class    PropertyTest
{

        public void static Main ( )
        {
        Number n = new Number ( );
        n.Anumber = 100;
        int m = n.Anumber;
        Console.WriteLine("Number = " + m);
    }
}
```

The class now declares a property called **Anumber** of type **int** and defines a *get accessor* method (also known as *getter*) and a *set accessor* method (also known as *setter*). The getter method used the keyword **return** to return the field's value to the caller. The setter method uses the keyword **value** to receive the value being passed in from the user. The type of **value** is determined by the type of property.

As the names imply, getter method is used to get (or read) the value and the setter method is used to set (or write) the value. In Program 12.6, the statement

```
n.Anumber = 100;
```

invokes the setter method and places an integer value 100 in a variable named **value** which in turn is assigned to the field **number**.

Similarly, the statement

```
int m = n.Anumber;
```

invokes the getter method and assigns the value of the property to **m**.

A property can omit either a **get** clause or the **set** clause. A property that has only a getter is called a *read-only* property, and a property that has only a setter is called a *write-only* property. A write-only property is very rarely used. There are other powerful features of properties. They include:

- Other than fetching the value of a variable, a **get** clause uses code to calculate the value of the property using other fields and returns the results. This means that properties are not simply tied to data members and they can also represent dynamic data.
- Like methods, properties are inheritable. We can use the modifiers **abstract, virtual, new** and **override** with them appropriately, so that the derived classes can implement their own versions of properties.
- The **static** modifier can be used to declare properties that belong to the whole class rather than to a specific instance of the class. (Like static methods, static properties cannot be declared with the **virtual, abstract** or **override** modifiers.)

An important point to note here is that we can specify any modifier only at the property level and this will affect both the accessors equally. For instance, we cannot override only one, leaving the other unaffected.

© IETE

b. What is polymorphism? Show how polymorphic behaviour can be achieved with the help of virtual methods. (8)

Answer:

polymorphism is the capability of one object to behave in multiple ways. Polymorphism can be achieved in two ways as shown in Fig. 13.8. C# supports both of them.

Operation polymorphism is implemented using overloaded methods and operators. We have already used the concept of overloading while discussing methods and constructors. The overloaded methods are 'selected' for invoking by matching arguments, in terms of number, type and order. This information is known to the compiler at the time of compilation and, therefore, the compiler is able to select and bind the appropriate method to the object for a particular call at *compile time* itself. This process is called *early binding*, or *static binding*, or *static linking*. It is also known as *compile time polymorphism*.

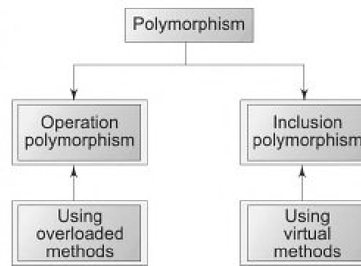

Fig. 13.8 *Achieving polymorphism*

Inclusion polymorphism is achieved through the use of virtual functions. Assume that the class **A** implements a **virtual** method **M** and classes **B** and **C** that are derived from **A** override the virtual method **M.** When **B** is cast to **A**, a call to the method **M** from **A** is dispatched to **B**. Similarly, when **C** is cast to **A**, a call to **M** is dispatched to **C**. The decision on exactly which method to call is delayed until runtime and, therefore, it is also known as *runtime polymorphism.* Since the method is linked with a particular class much later after compilation, this process is termed *late binding*. It is also known as *dynamic binding* because the selection of the appropriate method is done dynamically at runtime. Program 13.7 illustrates the use of virtual methods to implement polymorphic behaviour of objects.

```
using System;
class Maruthi
{
    public virtual void Display ( ) //virtual method
    {
        Console.WritLine("Maruthi car");
    }
}
class Esteem : Maruthi
{
    public override void Display( )
    {
        Console.WriteLine("Maruthi Esteem");
    }
}
class Zen : Maruthi
{
    public override void Display ( )
    {
        Console.WriteLine("Maruthi Zen");
    }
}
class Inclusion
{
    public static void Main( )
    {
        Maruthi m = new Maruthi ( );
        m = new Esteem ( );       //upcasting
        m.Display ( );
        m = new Zen ( );          //upcasting
        m.Display ( )
    }
}
```

Program 13.7 outputs:

```
Maruthi  : Esteem
Maruthi  : Zen
```

In Program 13.7, a particular car object, whether it be **Esteem** or **Zen**, is cast to **m** which is of type **Maruthi**, the base class, using the casting statements.

```
m = new Esteem ( );
m = new Zen ( );
```

When an object of **Esteem** is cast to **m**, then, **m** behaves like an **Esteem** type object. Similarly, when a **Zen** object is cast to **m**, it behaves like a **Zen** type object. Therefore, the two identical calls produce two different outputs:

```
Maruthi Esteem
Maruthi Zen
```

Q.7 a. Demonstrate the use of interface to support the concept of multiple inheritance. (8)

Answer:

we inherit methods and properties from several distinct classes. Since 'C++ - like' implementation of multiple inheritance proves difficult and adds complexity to the language, C# provides an alternate approach known as *interface* to support the concept of multiple inheritance. Although a C# class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

Most often we have situations where the base class of a derived class implements an interface. In such situations, when an object of the derived class is converted to the interface type, the inheritance hierarchy is searched until it finds a class that directly implements the interface. Consider the code in Program 14.3.

Program 14.3 | INHERITING A CLASS THAT IMPLEMENTS AN INTERFACE

```
using System;
interface Display
{
    void Print ( );
}
class B : Display    // implements Display
{
    public void Print ( )
    {
        Console.WriteLine("Base Display");
    }
}
class D : B          // inherits B class
{
    public new void Print ( )
    {
        Console.WriteLine("Derived Display");
    }
}
class InterfaceTest3
{
    public static void Main( )
    {
        D d = new D ( );
        d.Print ( );
        Display dis = (Display) d;
        dis.Print ( );
    }
}
```

Program 14.3 would produce the following output:

```
Derived Display
Base Display
```

Note that the statement

```
dis.Print ( );
```

calls the method **Print ( )** in base class B but not the one available in the derived class itself. This is because the derived class does not implement the interface. That is, the use of modifier **new** in the derived class "hides" the **Print** method implemented in the base class.

b. What is an operator method? Describe its syntax. Show how a binary operator "+" can be overloaded in C#. (8)

Answer:

To define an additional task to an operator, we must specify what it means in relation to the class (or struct) to which the operator is applied. This is done with the help of a special method called *operator method,* which describes the task. The general form of an operator method is:

```
public static retval operator op (arglist)
{
        Method body //task defined
}
```

The operator is defined in much the same way as a method, except that we tell the compiler it is actually an operator we are defining by the **operator** keyword, followed by the operator symbol *op*. The key features of operator methods are:

- They must be defined as **public** and **static.**
- The *retval* (return value) type is the type that we get when we use this operator. But, technically, it can be of any type.
- The *arglist* is the list of arguments passed. The number of arguments will be one for the unary operators and two for the binary operators.
  - In the case of unary operators, the argument must be the same type as that of the enclosing class or struct.
  - In the case of binary operators, the first argument must be of the same type as that of the enclosing class or struct and the second may be of any type.

Program 15.2 | **OVERLOADING + OPERATOR**

```
using System;
class Complex
{
    double x;           //real part
    double y;           //imaginary part
    public Complex ( )
    {
    }
    public Complex(double real, double imag)
    {
        x = real;
        y = imag;
    }
    public static Complex operator + (Complex c1, Complex c2)
    {
        Complex c3 = new Complex ();
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return (c3);
    }
    public void Display( )
    {
        Console.Write(x);
        Console.Write(" + j" + y);
        Console.WriteLine( );
    }
}
class ComplexTest
{
        public static void Main( )
        {
            Complex a, b, c;
            a = new Complex (2.5 , 3.5);
            b = new Complex (1.6, 2.7);
            c = a + b;
            Console.Write(" a = ");
            a.Display( );
            Console.Write("b = ");
            b.Display( );
            Console.Write("c =");
            c.Display( );
        }
    }
```

The output of Program 15.2 would be:

```
a = 2.5 + j3.5
b = 1.6 + j2.7
c = 4.1 + j6.2
```

Q.8 a. What do you mean by multicast delegates? Demonstrate with an example. (8)

Answer:

We have seen so far that a delegate can invoke only one method (whose reference has been encapsulated into the delegate). However, it is possible for certain delegates to hold and invoke multiple methods.

Such delegates are called *multicast delegates.* Multicast delegates, also known as *combinable delegates*, must satisfy the following conditions:

- The return type of the delegate must be **void.**
- None of the parameters of the delegate type can be declared as output parameters, using **out** keyword.

If **D** is a delegate that satisfies the above conditions and **d1, d2, d3** and **d4** are the instances of **D**, then the statements

```
d3 = d1 + d2; //d3 refers to two methods
d4 = d3 - d2; //d4 refers to only d1 method
```

are valid provided that the delegate instances **d1** and **d2** have already been initialized with method references and **d3** and **d4** contain **null** reference.

For a multicast delegate instance that was created by combining two delegates, the invocation list is formed by concatenating the invocation list of the two operands of the addition operation. Delegates are invoked in the order they are added.

Program 16.2 illustrates the application of multicast delegates. The program demonstrates the use of both addition and removal of delegates. Note the order in which the delegates **m3** and **m4** invoke the methods.

Program 16.2 **IMPLEMENTING MULTICAST DELEGATES**

```
using System;
delegate void MDelegate( );
class DM
{
    static public void Display( )
    {
        Console.WriteLine("NEW DELHI");
    }
    static public void Print( )
    {
        Console.WriteLine("NEW YORK");
    }
}
class MTest
{
    public static void Main( )
    {
        MDelegate m1 = new MDelegate(DM.Display);
        MDelegate m2 = new MDelegate (DM.Print);
        MDelegate m3 = m1 + m2;
        MDelegate m4 = m2 + m1;
        MDelegate m5 = m3 - m2;
        //invoking delegates
        m3( );
        m4( );
        M5( );
    }
}
```

The output of Program 16.2 would be:

```
NEW DELHI
NEW YORK
NEW YORK
NEW DELHI
NEW DELHI
```

b. **Briefly describe System. Console class and its various Console input and output methods. (8)**

Answer:

The methods for reading from and writing to the console are provided by the **System.Console** class. This class gives us access to the standard input, standard output and standard error streams as shown in Table 17.1.

Table 17.1 *Input/Output streams*

STREAM OBJECT	REPRESENTS
Console.In	Standard input
Console.Out	Standard output
Console.Error	Standard error

The standard input system **Console.In** gets input by default from the keyboard. We can also redirect it to receive input from a file. The standard output stream **Console.Out** sends output to the screen by default. We can also redirect the output to a file.

The standard error stream **Console.Error** usually sends error messages to the screen. Even when the standard output is sent to a file, error messages, if any, will be displayed on the screen.

The console input stream object supports two methods for obtaining input from the keyboard:

- **Read ( )**Returns a single character as **int**. Returns -1 if no more characters are available.
- **ReadLine ( )**Returns a string containing a line of text. Returns **null** if no more lines are available.

These methods can be invoked using either **Console.In** object or **Console** class itself. The following code snippet reads a character from the keyboard and displays it on the screen.

```
int x = Console.Read ( );
Console.WriteLine ( (char) x );
```

We have used a casting operator (char) to **x** to convert it to a character type. Remember, **x** was read as an **int**.

The following code reads entire line of text as a single string and displays it on the screen.

```
string str = Console.ReadLine( );
Console.WriteLine(str);
```

The console output stream supports two methods for writing to the console:

- **Write ( )** Outputs one or more values to the screen without a newline character
- **WriteLine ( )** Outputs one or more values to the screen (same as Write ( ) method) but adds a newline character at the end of the output

Both the methods have various overloaded forms for handling all the predefined types and therefore we can easily output many different types of data. **WriteLine( )** also allows us to output data in many different formats (in a way, comparable to the **printf** method of C).

The **Write( )** method sends information into a buffer. This buffer is not flushed until a newline (or end-of-line) character is sent. As a result, this method prints output on one line until a newline character is encountered. For example, the statements,

```
Console.Write("Hello ");
Console.Write("C Sharp!");
```

will display the words Hello C Sharp! on one line and wait for displaying of further information on the same line. We may force the display to be brought to the next line by printing a newline character as shown below:

```
Console.Write("\n");
```

For example, the statements

```
Console.Write("Hello");
Console.Write("\n");
Console.Write("C Sharp!");
```

will display the output in two lines as follows:

```
Hello
C Sharp!
```

The **WriteLine( )** method, by contrast, takes the information provided and displays it on a single line followed by a line feed (carriage return). This means that the statements

```
Console.WriteLine("Hello");
Console.WriteLine("C Sharp!");
```

will produce the following output:

```
Hello
C Sharp!
```

Q.9 a. What are run-time errors? Give some examples. (4)

Answer:

Sometimes, a program may compile successfully creating the **.exe** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Passing a parameter that is not in a valid range or value for a method.
- Attempting to use a negative size for an array.
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting an invalid string to a number or vice versa.
- Accessing a character that is out of bounds of a string, and so on.

When such errors are encountered, C# typically generates an error message and aborts the program.

b. Briefly describe the usage and purpose of finally statement while handling exceptions. (6)

Answer:

C# supports another statement known as a **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements. A **finally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:

```
try                         try
{                           {
    ..........                      ..........
    ..........                      ..........
}                               }
finally                         catch (....)
{                               {
    ..........                      ..........
    ..........                      ..........
}                               }
                                catch (....)
                                {
                                    ..........
                                    ..........
                                }
                                .
                                .
                                .
                                .
                                finally
                                {
                                    ..........
                                    ..........
                                }
```

When a **finally** block is defined, the program is guaranteed to execute, regardless of how control leaves the try, whether it is due to normal termination, due to an exception occurring or due to a jump statement. Figure 18.3 illustrates this. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

In Program 18.4, we may include the last two statements inside a finally block as shown below:

```
finally
{
    int y = a[1]/a[0];
    Console.WriteLine("y = " +y);
}
```

This will produce the same output.

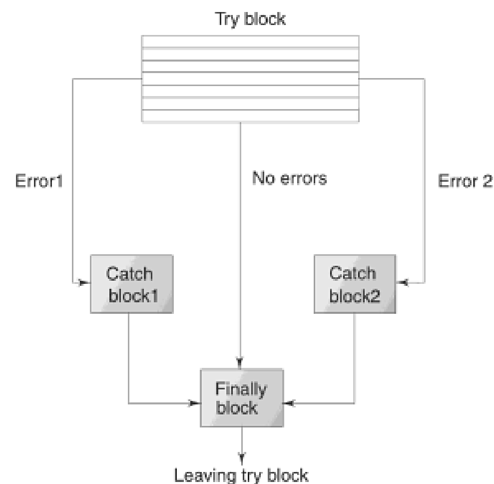

Fig. 18.3 *Execution paths of try-catch-finally blocks*

c. Briefly describe the process of creating and starting up threads with an example. (6)

Answer:

A thread can be created in C# using the constructor of the **Thread** class. We need to pass the **ThreadStart delegate** to the **Thread** class constructor along with the name of the method from which the execution should start. The **ThreadStart delegate** is defined as:

```
public delegate void ThreadStart();
```

To create a new thread in a C# program, we can use the following statement:

```
Thread threadname = new Thread(new ThreadStart(methodname);
```

Here, *threadname* represents the name of the new thread and *methodname* is the name of the method from which the execution starts.

For example, to create a thread **t1**, we can use the following code:

```
Thread t1 = new Thread(new ThreadStart(First);
```

If the method from which execution needs to start is defined in a class other than the class in which the thread is created then we must use an object of that class to access the method. **The Start()** method of the **Thread** class starts a new thread. Program 19.1 illustrates how a thread is created and executed.

Program 19.1 | **CREATING AND STARTING THREADS**

```
using System;
using System.Threading;
public class AOne
{
    public void First()
    {
        Console.WriteLine("First method of AOne class is running on T1 thread.");
        Thread.Sleep(1000);
        Console.WriteLine("The First method called by T1 thread has ended.");
    }
    public static void Second()
    {
        Console.WriteLine("Second method of AOne class is running on T2 thread.");
        Thread.Sleep(2000);
        Console.WriteLine("The Second method called by T2 thread has ended.");
    }
}
```

```
public class ThreadExp
{
    public static int Main(String[] args)
    {
        Console.WriteLine("Example of Threading");
        AOne a = new AOne();
        Thread T1 = new Thread(new ThreadStart(a.First)); // Creating thread T1
        T1.Start(); // Starting thread T1
        Console.WriteLine("T1 thread started.");
        Thread T2 = new Thread(new ThreadStart(AOne.Second)); Creating T2
        T2.Start(); Starting thread T2
        Console.WriteLine("T2 thread started.");
        return 0;
    }
}
```

In the above C# program, two threads **T1** and **T2** have been created and started using the **Start()** method of the **Thread** class. In case of **T1** thread, the execution starts from **First method** defined in the **AOne** class while in case of **T2** thread the execution starts from **Second method**, which is a static method. The output of the above program is:

```
Example of Threading
First method of AOne class is running on T1 thread.
T1 thread started.
Second method of AOne class is running on T2 thread.
T2 thread started.
The First method called by T1 thread has ended.
The Second method called by T2 thread has ended.
Press any key to continue . . .
```

TEXT BOOK

I. Programming in C# - A Primer, E. Balagurusamy, Second Edition, TMH, 2008

© IETE