Solutions

Q.2 a. What are the benefits of .Net strategy advanced by Microsoft? (6)

Answer:
Microsoft has advanced the .NET strategy in order to provide a number of benefits to developers and users. Some of the major benefits are:

- Simple and faster systems development
- Rich object model
- Enhanced built-in functionality
- Many different ways to communicate with the outside world
- Integration of different languages into one platform
- Easy deployment and execution
- Wide range of scalability
- Interoperability with existing applications
- Simple and easy-to-build sophisticated development tools
- Fewer bugs
- Potentially better performance

b. What is the Microsoft Intermediate Language? (4)

Answer:
Microsoft intermediate language (MISL), or simple IL, is an instruction set into which all the .NET programs are compiled. It is akin to assembly language and contains instruction for loading, storing, initializing, and calling methods. When we compile a C# program or any program written in a CLS-complaint language, the source code is compiled into MSIL.

c. With the help of example, explain 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*.**

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 codes will illustrates this:

```
int m = 10;
object ob = 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 = 10;
```

```
object ob = (object)m;    // C-style casting
```

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

```
int m = 10;
object ob = m;
m = 20;
Console.WriteLine(m);                // m = 20
Console.WriteLine(op);          // op = 10
```

When a code changes the value of 'm', the value of 'op' is not affected.

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 been previously been boxed. In contrast to boxing, unboxing is an explicit operation using casting.

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

when performing unboxing, C# checks that the value type we request is actually stored in the object under conversion. 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 = 10;
object ob = m;
byte n = (byte) op;
```

will produce a runtime error.

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. With the help of syntax and example, explain "foreach" statement used in C# programming. (5)

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* declares 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 is as follows:

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

This program segment displays the command line arguments. The same may be achieved using the **for** statement as follows:

```
public static void Main (string[] args) {
    for (int i=0; i<args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

The following program illustrates the use of **foreach** statement for printing the contents of a numerical array.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace foreachloopdemo
{
    class ForeachTest {
        public static void Main() {
            int[] arrayInt = { 32, 56, 76, 21 };
            foreach (int m in arrayInt)
            {
                Console.Write(" " + m);
            }
            Console.WriteLine();
        }
    }
}
```

b. What is "fallthrough" in switch statement? How it is achieved in C#? (5)

Answer:

In the absence of the break statement in a case block, if the control moves to the next case block without any problem, it is known as 'fallthrough'. Fallthrough is permitted in C, C++ and Java. But C# does not permit automatic fallthrough, if the case block contains executable code. However, it is allowed if the case block is empty. For instance,

```
switch (m) {
    case 1 :
        x = y;
    case 2 :
        x = y + 1;
    default :
        x = y - m;
}
```

is an error in C#. If we want two consecutive case blocks to be executed continuously, we have

to force the process by using the **goto** statement. For example:

```
switch (m) {
        case 1 :
                x = y;
                goto case 2;
        case 2 :
                x = y + 1;
                goto default;
        default :
                x = y - m;
                break;
}
```

The **goto** mechanism enables us to jump backward and forward between cases therefore arrange labels arbitrarily. Example is as follows:

```
switch (m) {

        default :
                x = y - m;
                break;
        case 2 :
                x = y + 1;
                goto default;
        case 1 :
                x = y;
                goto case 2;
}
```

c. Given a number, write a program using while loop to find the sum of digits of the number. The number should be READ through keyboard. For example if the entered number is 12345, the output should be (1+2+3+4+5). (6)

Answer:

```
class SumOfDigit
    {
        static void Main(string[] args)
        {
            int num;
            int sum = 0;
            Console.WriteLine("Program to Find the SUM of digits of the
                  Entered Number");
            Console.WriteLine("Enter any number : ");
            num = Int32.Parse(Console.ReadLine());

            while (num != 0)
            {
                int dig = num % 10;
                sum = sum + dig;
                num = num / 10;
            }
```

```
            Console.WriteLine("Required Sum of digits of the entered
                  number is : " + sum);
        }
    }
```

Q.4 a. What is method overloading? Explain. Write a program to illustrate the concept of method overloading. (8)

Answer:

C# allows us to create more than one method with the same name, but with different parameters 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. 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 arguments list. For example, an overloaded `add()` method handles different types of data as shown below:

```
//Method Definitions
int add (int a, int b) { ... }           // method 1
int add (int a, int b, int c) { ... }    // method 2
double add (float x, float y) { ... } // method 3
double add (int p, float q) { ... }      // method 4
double add (float x, int y) { ... }      // method 5

// Method Calls
int x = add(10, 20);                     // method 1
double y = add(3.2f, 25);                // method 5
int z = add (2, 3, 4);                // method 2
double x = (2.2f, 3.3f);              // method 3
double z = (10, 2.2f);                // method 4
```

The method selection involves the following steps:

- The compiler tries to find an exact match in which the types of actual parameters are the same and uses that method.
- 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;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MethodOverloading
{
    class MethodMOverloading
```

```
    {
        static void Main(){
            Console.WriteLine(volume(10));
            Console.WriteLine(volume(2.5f, 8));
            Console.WriteLine(volume(10L, 25, 5));
        }
        static int volume(int x)
        {
            return (x * x * x);
        }
        static double volume(float r, int h)
        {
            return (3.14 * r * r * h);
        }
        static long volume(long l, int b, int h)
        {
            return (l * b * h);
        }
    }
}
```

b. Explain the two different ways to initialize an array. Give example. (4)

Answer:

The array can be initialized using the array subscripts as shown below.

```
arrayname[subscript] = value;
```

Example:

```
list[0] = 25;
list[1] = 35;
.........
.........
list[10] = 99;
```

C# creates arrays starting with a subscript of 0 and ends with a value one less than the *size* specified. Unlike C, C# protects arrays from overruns and underruns. Trying to access an array beyond its boundaries will generate an error message. Example

```
list[15] = 10;       // error
```

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

```
type [] arrayname = [list of values];
```

The array initialize is a list of values separated by commas and defined on both ends by curly braces. No size is given. The compiler allocates enough space for all the elements specified in the list.

```
int[] list = {25, 35, 45, 55, 65, 75, 85, 95, 15, 99};
```

The preceding line is equivalent to :

```
int[] list = new list[8]{25, 35, 45, 75, 85, 95, 15, 99};
```

This combines all three steps, namely declaration, creation and initialization.

c. What is a variable size array? How is it different from a rectangular array?

(4)

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. Variable-size arrays are called ***jagged*** arrays. The elements can be accessed as follows:

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

The difference is 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 elements is within its own square brackets.

Q.5 a. Describe any four methods that can be used to create immutable strings using 'string' or 'String' objects. (8)

Answer:

We can create immutable strings using 'string' or 'String' objects in a number of ways as described below:

- **Assigning String Literals**
  The most common way to create a string is to assign a quoted string of characters known as *string literal* to a string object. For example:

```
string str1;          // declaring a string object
str1 = "ABC";         // assigning string literal
```

  Both these statements can be combined into one as follows:

```
string str1 = "ABC";
```

- **Copying Strings**
  We can also create new copies of existing strings. This can be accomplished in two ways:
  i. Using the overloaded = operator
  ii. Using the static **Copy** method

  Example:

```
string str2 =str1;                    // assigning
string str2 = string.Copy(str1);          // copying
```

  Both these statements would accomplish the same thing, namely, copying the contents of str1 into str2.

- **Concatenating Strings**
  We may also create new strings by concatenating existing strings. There are a couple of ways to accomplish this:
  i. Using the overloaded + operator
  ii. Using the static **Concat** method

Examples:

```
string str3 = str1 + str2;    //str1 and str2 exist already
string str3 = str1.Concat(str2);
```

If `str1 = 'ABC'` and `str2 = 'xyz'`, then both the statements will store the string `'ABCxyz'` in `str3`. The content of str2 is simply appended to the content of `str1` and the result is stored in `str3`.

- **Reading from the Keyboard**
  It is possible to read a string value interactively from the keyboard and assign it to a string object.

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

  On reaching this statement, the computer will wait for a string of characters to be entered from the keyboard. When the 'return key' is pressed, the string will be read and assigned to the string object **str.**

- **The ToString Method**
  Another way of creating a string is to call the **ToString** method on an object and assign the result to a string variable.

```
int num = 123;
string numStr = num.ToString();
```

  This statement converts the number 123 to a string '123' and then assigns the string value to the string variable **numStr.**

b. What is structure? How values are assigned to the members of structure? State the important differences between structure and class. (8)

Answer:

Structures, often referred as structs, provide a unique way of packing together data of different. 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.

Structures are declared using the **structb keyword**. Its syntax is defined 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 of structure Student. The identifier **Student** can now be used to create variables of type **Student**. Examples is as follows:

```
Student s1;
```

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

Member variables can be assigned values using the simple dot notation as follows:

```
s1.Name = "Ajay";
s1.RollNumber = 2000;
s1.TotalMarks = 5000.75;
```

Important difference between Structure and Class are listed below:

Category	Classes	Structures
Data type	Reference type and therefore stored on the heap	Value type and therefore stored on the stack. Behave like simple data types
Inheritance	Support inheritance	Do not support inheritance
Default values	Default value of a class type is null	Default value is the value produced by 'zeroing out' the fields of the struct.
Field initialization	Permit initialization of instance fields	Do not permit initialization of instance fields.
Constructors	Permit declaration of parameterless constructors	Do not permit declaration of parameterless constructors
Destructors	Supported	Not supported
Assignment	Assignment copies the reference	Assignment copies the values

Q.6 a. Explain with suitable examples the three pillars of object-oriented programming. (6)

Answer:

All object-oriented languages employ three core principles, namely

- Encapsulation
- Inheritance, and
- Polymorphism

These are often referred as three pillars of object-oriented programming.

- ***Encapsulation*** provides the ability to hide the internal details of an object from its users. The outside user may not be able to change the state of an object directly. However, the state of an object may be altered indirectly using what are known as *accessor* and *mutator* methods. In C#, encapsulation is implemented using access modifier keywords **public, private, and protected**.
  The concept of encapsulation is also known as *data hiding* and *information hiding*. When done properly, we can create software 'black boxes' that can be independently tested and used.
- ***Inheritance*** is the concept we use to build new classes using the existing class definitions. Through inheritance we can modify a class the way we want to create new objects. The original class is known as *base* or *parent* class and the modified one is known as *derived class* or *subclass* or *child class.*
  The concept of inheritance facilitates the reusability of existing code and thus improves the integrity of programs and productivity of programmers.

- ***Polymorphism*** is the third concept of OOP. It is the ability to take more than one form. For example, an operation may exhibit different behaviour in different situations. The behaviour depends upon the types of data used in the operation. For example, an addition operation involving two numeric values will produce a sum and the same addition operation will produce a string if the operands are string values instead of numbers. Similarly, a method when called with one set of parameters may draw a circle but when called with another set of parameters may draw a triangle.
  Polymorphism is extensively used while implementing inheritance.

b. What is an Indexer? How does an indexer differ from a property in terms of implementation? (5)

Answer:
Indexers are location indicators and are used to access class objects, just like accessing elements in an array. They are useful in cases where a class is a container for other objects.
An indexer looks like a property and is written the same way a property is written, but with two differences:

- An indexer takes an index argument and looks like an array.
- The indexer is declared using the name **this.**

The indexer is implemented through **get** and **set** accessors for the [] operator. Example is as follows:

```
public double this[int idx] {
     get {
          // return desired data
     }
     set {
          // set desired data
     }
}
```

The implementation rules for **get** and **set** accessors are the same as for properties. The return type determines what will be returned, in this case a **double**. The parameter inside the square brackets is used as the *index*.

Indexers are sometimes referred to as 'smart arrays'. Indexers and properties are very similar in concept, **but differ in the following ways:**

- A property can be static member, whereas an indexer is always an instance member.
- A **get** accessor of a property corresponds to a method with no parameters, whereas a **get** accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A **set** accessor of a property corresponds to a method with a single parameter named **value**, whereas a **set** accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus the parameter named **value**.
- It is an error for an indexer to declare a local variable with the same name as an indexer parameter.

c. What are the constraints that C# imposes on the accessibility of members and

classes when they are used in the process of inheritance? (5)

Answer:

C# imposes certain constraints on the accessibility of members and classes when they are used in the process of inheritance.

- The direct base class of a derived class must be at least as accessible as the derived class itself.
- Accessibility domain of a member is never larger than that of the class containing it.
- The return type of method must be at least as accessible as the method itself.

For example, the inheritance relationship

```
class A {
      .......
}
public class B : A {
      .......
}
```

is illegal because A is 'internal' by default and B is public. A should be at least as accessible as B. Consider another example:

```
class A {
      private class B {
            public int x;
      }
}
```

Here, the public data x is not accessible outside the class B. It is due to the constraints imposed by the accessibility of class **B**. Because **B** is private, the public on **x** is reduced to private.

In many situations we use methods to handle class objects and therefore they take classes as their return types. In such cases, the return type of a method must be at least as accessible as the method itself. For instance, consider the following code:

```
class A {
      .......
}
public class B {
      A method1() {}            // ok
      internal A Method2() {}   // ok
      public A Method3() {}          // error
}
```

All the three methods declared in class **B** specify **A** as their return type. Since the class **A,** by default, assumes **internal** as the accessibility level, the method

```
      public A Method3() {          //  public  is  higher  than
internal
            .......
      }
```

is an error. The method cannot have an accessibility level higher than that of its return type.

Q.7 a. What is an Interface? What do you mean by "Extending" and interface and "Implementing" interface? Explain with the help of example. (8)

Answer:

An **interface** can contain one or more methods, properties, indexers and events but none of them are implemented in the interface itself. It is responsibility of the class that implements the interface to define the code for implementation of these members.

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 keyword and ***InterfaceName*** is a valid C# identifier. Member declarations will contain only a list of members without implementation code. For example, below is a simple interface that defines a single method:

```
interface Show {
     void Display();
}
```

In addition to methods, interfaces can declare properties, indexers and events. Example is as follows:

```
interface ABC {
     int ABCproperty {
          get;
     }
     event someEvent Changed;
     void Display();
}
```

The accessibility of interface can be controlled by using the modifiers **public, protected, internal, and private.** The use of a particular modifier depends on the contact in which the interface declaration occurs.

Extending An Interface

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new suninterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved as follows:

```
interface I2 : I1 {
     members of I2;
}
```

For example, we can put all members of particular behaviour category in one interface and the members of another category in the other. Consider the code below:

```
interface Addition (
     int Add(int x, int y);
}
```

```
interface Compute : Addition {
     int Sub(int x, int y);
}
```

The interface **Compute** will have both the methods and any class implementing the interface **Compute** should implement both of them; otherwise, it is an error.

We can also combine several interfaces together into a single interface. Following declarations are valid:

```
interface I1 {
     --------
}
interface I2 {
     --------
}

interface I3 : I1, I2 {       // multiple inheritance
     ----------
}
```

While interfaces are allowed to extend other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. It is the responsibility of the class that implements the derived interface to define all the methods.

It is important to remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract members.

Implementing Interface

Interfaces are used as 'superclasses' whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. this is done as follows:

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

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

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

This shows that a class can extend another class while implementing interfaces.

In C#, we can drive from a single class and, in addition, implement as many interfaces as the class needs. When a class inherits from a superclass, the name of each interface to be implemented must appear after the superclass name. Examples:

```
class A : B, I1, I2 {
     // body of class
}
```

where B is the base class and I1, I2 are interfaces. The base class and interfaces are separated by commas.

b. What is operator overloading? Write a program in C# to overload binary plus (+) operator to add two complex number of type x=a+iy. (8)

Answer:
To define an additional task to an operator, we must specify what is 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 overloading** which describe the task. The general form of an operator overloading is:

```
public static retval operator op (arglist) {
        method body // task define
}
```

The operator is defined in much the same way as a method, except that we 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 return value (retval) 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 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.

Examples of overloaded operators are:

public static Vector operator + (Vector a, Vector b)
public static Vector operator – (Vector a)
public static bool operator ==(Vector a, Vector b)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


    class Complex
    {
        double x;
        double y;
        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();
        }
    }
```

Q.8 a. What is delegate? Discuss the syntax of delegate declaration. What are the modifiers that can be applied to a delegate? Give examples. (8)

Answer:

In C#, a delegate means a method acting for another method. A delegate in C# is a class type object and is invoked a method that has been encapsulated into it at the time of its creation. Creating and using delegates involves four steps:

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

A delegate declaration defines a class using the class **System.Delegate** as a base class. Delegate methods are any functions (defined in a class) whose signature matches the delegate signature exactly. The delegate instance holds the reference to delegate methods. The instance is used to invoke the methods indirectly.

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

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 ob1, object ob2);
private delegate string GetString();
delegate double DoubleOperation(double x);
```

Although the syntax of delegate is similar to that of method definition (without method body), the use of keyword **delegate** tells the compiler that it is definition of a new class using the **System.Delegate** as the base class. Since it is a class type, it can be defined in any place where a class definition is permitted. Thus, a delegate may be defined in the following places:

- Inside a class
- Outside all classes
- As the top level object in a namespace

Depending on how visible we want the delegate to be, we can apply any of the visibility modifiers to the delegate definition. Delegate types are implicitly sealed and therefore it is not possible to derive any type from a delegate type. it is also not permissible to derive a non-delegate class type from **System.Delegate.**

b. Explain the general format of the "Standard Numerical format". (8)

Answer:

The **standard numeric format** consists of a numeric format character and optionally a precision specifier. The general format of a marker would appear like

{n:fc[p]}

where **n** is the index number of the argument to be substituted, **fc** is the format character and **p** is the precision specifier

- **Currency Formatting:** The currency formatting converts the numerical value to a string containing a locale-specific currency symbol. By default, the dollar symbol is added. However, it can be changed using a **NumberFormatInfo** object. Examples are as follows:

  ("{0 : C}", 4567.899) ------> $4,567.90

  ("{0 : C}", -4567.899) ------> ($4,567.90)

  When the number is negative, the value is printed as positive inside simple brackets. The decimal part, by default, is rounded to two places.

- **Integer Formatting:** Integer format converts a given numerical value to an integer of

base. The minimum number of digits is determined by the precision specifier. If the precision specifier is less than the actual digits, the specifier is ignored and all the digits are printed. If it is greater, then the result is left-padded with zeros to obtain the specifier number of digits. Examples are as follows:

("{ 0 : D }" , 45678 ----------> 45678
("{ 0 : D8 }" , 45678 ----------> 00045678

- **Exponential Formatting:** Exponential formatting character (E or e) converts a given value to a string in the form of:

  m.dddd E+xxx

  The output will contain only one digit m before the decimal point. The number of decimal places (dddd) is decided by the precision specifier. By default, six places are used. The format character E (or e) will appear in the output. Examples are as follows:

  ("{ 0 : E }" , 34567.899) ------------> 3.456790E+004
  ("{ 0 : E9 }" , 34567.899) ------------> 3.456789900E+004
  ("{ 0 : e5 }" , 34567.899) ------------> 3.45679e+004

  The decimal part is rounded (not truncated) to the specifier places. If the specified places are more, then the decimal part is padded at the right with zeros.

- **Fixed-Point Format:** The fixed-point format converts the given value to a string containing decimal places as decided by the precision specifier. By default, two decimal places are assumed. Zeros as a precision specifier is allowed. Examples are as follows:

  ("{ 0 : F }" , 3456.7899) ------------> 3456.79
  ("{ 0 : F6 }" , 3456.7899) ------------> 3456.789900
  ("{ 0 : F0 }" , 3456.7899) ------------> 3456

  The decimal part is always rounded (or padded with zeros) to the specified precision places. A zero precision converts the value to the nearest integer.

- **Number Format:** This format produces the output with the embedded commas, like 5,678.45. Examples are as follows:

  ("{ 0 : N }" , 34567.899) ------------> 34,567.90
  ("{ 0 : N }" , 34567) ------------> 34,567.00
  ("{ 0 : N4 }" , 34567.899) ------------> 34,567.8990

  The precision specifier decides the number of decimal places. It is two by default. The decimal part is rounded to the specified precision places. If the specified places are more, the decimal part is padded with zeros.

Q.9 a. Consider the following code for nested-try block:

```
try {
    ..... // Point P1
    .....
    try {
        .....  // Point P2
        .....
    }
    catch {
```

```
        .....  // Point P3
        .....
    }
    finally {
        .....
        .....
    }
    .....  // Point P4
    .....
    }
    catch {
    .....
    .....
    }
    finally {
    .....
    .....
    }
```

Explain how exceptions that are thrown as at points P1, P2, P3, and P4 are handled?(2×4)

Answer:

When nested try blocks are executed, the exceptions that are thrown at various points are handled as follows:

- The points P1 and P4 are outside the inner try block and therefore any exceptions thrown at these points will be handled by the **catch** in the outer block. The inner try block is simply ignored.
- Any exception thrown at point P2 will be handled by the inner **catch** handler and the inner **finally** will be executed. The execution will continue at point P4 in the program.
- If there is no suitable **catch** handler to catch an exception thrown at P2, the control will leave the inner block (after executing the inner **finally**) and look for a suitable **catch** handler in the outer block. If a suitable one is found, then that handler is executed followed by the outer **finally** code. The code at point P4 will be skipped.
- If an exception is thrown at point P3, it is treated as if it had been thrown by the outer try block and, therefore, the control will immediately leave the inner block, after executing the inner **finally**, and search for a suitable **catch** handler in the outer block.
- In case, a suitable **catch** handler is not found, then the system will terminate program execution with an appropriate message.

b. What is thread pooling? Write a program in C# to illustrate the use of thread pool. (8)

Answer:

In thread pooling, a thread pool is created to perform multiple tasks simultaneously. A thread pool is basically a group of threads that can be run simultaneously to perform a number of tasks in the background. This feature of C# is mainly used in server applications. In server applications, a main thread receives the requests from the client computers and passes it to a

thread in the thread pool for processing of the request. In this manner, the main thread functions asynchronously and is free to receive the requests from client computers. The main thread does not have to process the request. Instead, the request is transferred to a thread in the thread pool for processing. A delay in receiving the requests from the client computer does not occur because of implementation of thread pooling in server applications. After the thread in the thread pool completes the task of processing the client request, it waits in a queue for performing another task. In this way, the thread in the thread pool can be reused for performing different tasks. The reusability of the thread because of thread pooling enables a server application to avoid creating a new thread for every task.

A thread pool may contain a number of threads, each performing a specific task. If all the threads in a thread pool are occupied in the performing their tasks then a new task, which needs to be processed, waits in a queue until a thread becomes free. The **.NET** frame work provides a thread pool through the **ThreadPool** class. We can either implement a custom thread pool in a C# program or use the thread pool provided through the **ThreadPool** class. It is easy to implement a thread pool through the **ThreadPool** class.

Program to illustrate the concept of Thread Pooling

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Text;

namespace ThreadPooling
{
    class ThreadPoolTest
    {
        static object showThread = new object();
        static int runThreads = 20;

        public static void Main() {
            for (int i = 0; i < runThreads; i++) {
                ThreadPool.QueueUserWorkItem(Display, i);
            }
            Console.WriteLine("Running 20 threads\n");
            lock (showThread) {
                while (runThreads > 0) Monitor.Wait(showThread);
            }
            Console.WriteLine("All Threads stopped successfully");
            Console.ReadLine();
        }
        public static void Display(object threadObj) {
            Console.WriteLine("Started Thread : " + threadObj);
            Thread.Sleep(3000);
            Console.WriteLine("Ended Thread :" + threadObj);
            lock(showThread){
                runThreads--;
                Monitor.Pulse(showThread);
            }

        }
    }
}
```

TEXT BOOK

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