CS1013 C# and .Net   Question Bank UNIT 1 & 2 WITH ANSWER

Unit - I

Part A

1. What are the advantages of using .NET?
2. Differentiate value type and reference type.
3. Is it possible to have two Main() in a C# code? If so, how is it resolved?
4. What do you mean by CLR?
5. What is metadata? Mention its uses in .NET.
6. What is Namespace? List out the Namespaces of .NET Framework.
7. What are boxing and unboxing
8. Define inter-operability. How does .net Achieve this?
9. What do you mean by Jagged Arrays?
10. What is the use of Enumeration?What is string? How strings are declared?

PART – B

1. Explain in detail about the .net Framework
2. (i) Explain the execution model of the .NET framework.(6)
   (ii) Describe the components of the .Net framework and explain the features of each component (10)
3. EXPLAIN THE FOLLOWING
   i. For each   ii. Structures   iii. Arrays   iv. Array list
4. Explain in detail about various operators available in C#.
5. (i) Explain about various value and reference types supported by c# (8)
   (ii) What is jagged array? Explain its use with simple example (8)
6. Explain about Enumerators and structures in C#.

Unit - I

Part A

1. What are the advantages of using .NET?

- Maintainability
- Deployment
- Security
- Cross platform: service-oriented Architecture:
- Interoperation with existing applications:
- Integration with legal systems
- Code management
- Robustness
- All libraries in one place
- Localisation and globalisation
- Less complexity and consistent programming model
- Rapid application development
- Scalability

2. Differentiate value type and reference type.

Value Types

Value types include the following:

- All numeric data types
- Boolean, Char, and Date
- All structures, even if their members are reference types
- Enumerations, since their underlying type is always SByte, Short, Integer, Long, Byte, UShort, UInteger, or ULong

Reference Types

Reference types include the following:

- String
- All arrays, even if their elements are value types
- Class types, such as Form
- Delegates

3. Is it possible to have two Main() in a C# code? If so, how is it resolved?

- Yes. This problem can be resolved by specifying which Main is to be use< to the compiler at the time of compilation as shown below: csc filename.cs/main:classname

4. What do you mean by CLR?

- Common Language Runtime (CLR) is a managed execution environment that is part of Microsoft's .NET framework. CLR manages the execution of programs written in different supported languages.
- CLR transforms source code into a form of bytecode known as Common Intermediate Language (CIL). At run time, CLR handles the execution of the CIL code.

5. What is metadata? Mention its uses in .NET.

- "metadata is data that describes the state of the assembly and a detailed description of each type, attribute within the assembly".
- Metadata is often called data about data or information about information. Metadata is structured information that describes, explains, locates, or helps to retrieve, use, or manage an information resource.
- Metadata is used differently in different areas. Metadata schemes have been developed to describe various types of textual and non-textual objects including books, e-documents, art objects, educational and training materials, and scientific datasets.

6. What is Namespace? List out the Namespaces of .NET Framework.

- A namespace is a section of code that is identified with a specific name. The name could be anything such as somebody's name, the name of the company's department, or a city.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.WinForms;
using System.Data;
```

7. What are boxing and 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 w hich it places the value of the value type, 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.

Unboxing

- Unboxing is the process of converting the object type back to the value type.

```
int m = 100;
object om = m; // box m
int n = (int) om; // unbox om back to an int
```

8. Define inter-operability. How does .net Achieve this?

- Ability to work with each other, in the loosely coupled environment of a service-oriented architecture, separate resources don't ne^d to know the details of how they each work, but they need to have enough common ground to reliably exchange messages without error or misunderstanding. Through DLLs

9. What do you mean by Jagged Arrays?

- C# also allows you to create a special type of two-dimensional array called a jagged array. A jagged array is an array of arrays in which the length of each array can differ.
- Thus, a jagged array can be used to create a table in which the lengths of the rows are not the same.
- Jagged arrays are declared by using sets of square brackets to indicate each dimension. For example, to declare a two-dimensional jagged array, you will use this general form:

```
typef ] [] array-name = new type[size][ ];
```

10. What is the use of Enumeration?What is string? How strings are declared?

- An enumeration type (also named an enumeration or an enum) provides an efficient way to define a set of named integral constants that may be assigned to a variable.
- A string is an object of type String whose value is text. Internally, the text is stored as a sequential read- only collection of Char objects. There is no null-terminating character at the end of a C# string

```
string[] arr = new String[]{};
```

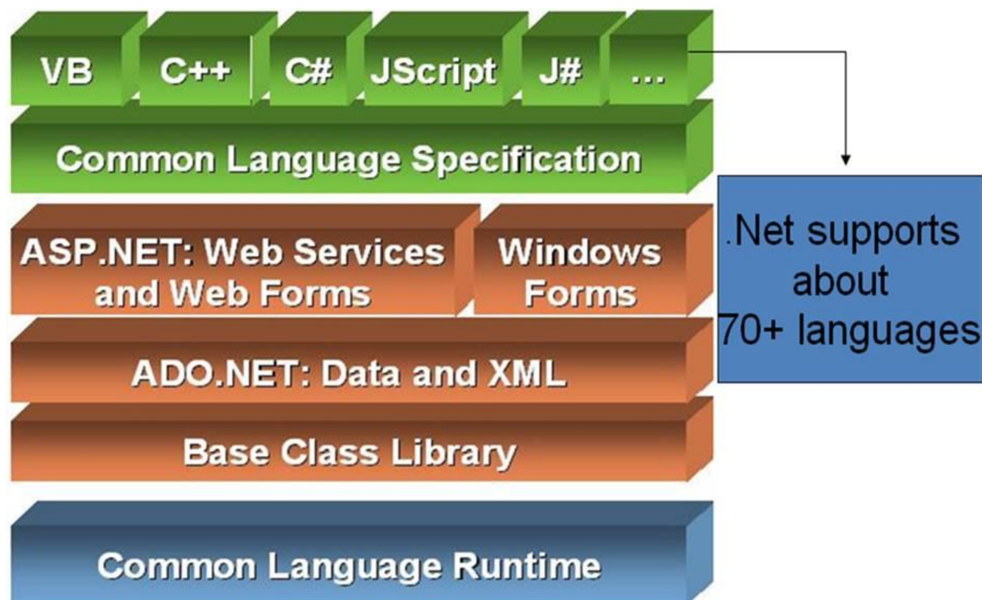
PART – B

1. Explain in detail about the .net Framework

Architecture of .Net Framework

There are many articles are available in the web on this topic; I just want to add one more article over the web by explaining Components of .Net Framework.

Components of .Net Framework



Net Framework is a platform that provides tools and technologies to develop Windows, Web and Enterprise applications. It mainly contains two components,

1. Common Language Runtime (CLR)
2. .Net Framework Class Library.

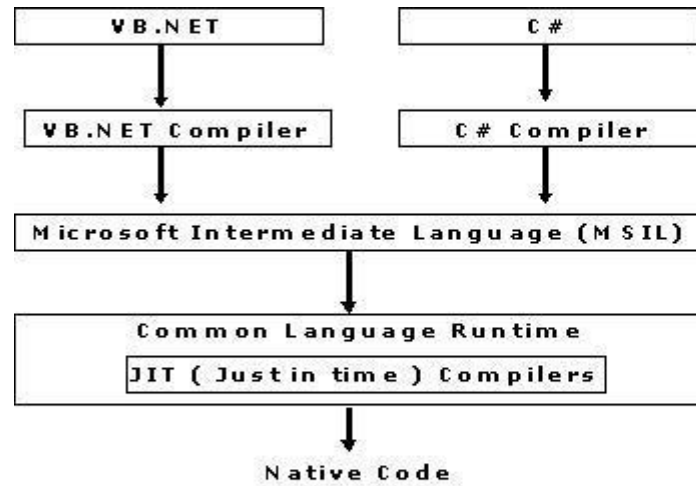
1**. Common Language Runtime** (CLR)

.Net Framework provides runtime environment called **Common Language Runtime** (CLR).It provides an environment to run all the .Net Programs. The code which runs under the CLR is called as **Managed Code**. Programmers need not to worry on managing the memory if the programs are running under the CLR as it provides memory management and thread management.

Programmatically, when our program needs memory, CLR allocates the memory for scope and de-

allocates the memory if the scope is completed.

Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to **Microsoft Intermediate Language** (MSIL) intern this will be converted to **Native Code** by CLR. See the below Fig.



There are currently over 15 language compilers being built by Microsoft and other companies also producing the code that will execute under CLR.

2. **.Net Framework Class Library** (FCL)

This is also called as Base Class Library and it is common for all types of applications i.e. the way you access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET.

The following are different types of applications that can make use of .net class library.

1. Windows Application.
2. Console Application
3. Web Application.
4. XML Web Services.
5. Windows Services.

In short, developers just need to import the BCL in their language code and use its predefined methods and properties to implement common and complex functions like reading and writing to file, graphic rendering, database interaction, and XML document manipulation.

3. **Common Type System** (CTS)

It describes set of data types that can be used in different .Net languages in common. (i.e), CTS ensures that objects written in different .Net languages can interact with each other.

For Communicating between programs written in any .NET complaint language, the types have to be compatible on the basic level.

The common type system supports two general categories of types:

Value types:

Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations.

Reference types:

Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types. The class types are user-defined classes, boxed value types, and delegates.

4**. Common Language Specification** (CLS)

It is a sub set of CTS and it specifies a set of rules that needs to be adhered or satisfied by all language compilers targeting CLR. It helps in cross language inheritance and cross language debugging.

Common language specification Rules:

It describes the minimal and complete set of features to produce code that can be hosted by CLR. It ensures that products of compilers will work properly in .NET environment.

Sample Rules:

1. Representation of text strings
2. Internal representation of enumerations
3. Definition of static members and this is a subset of the CTS which all .NET languages are expected to support.
4. Microsoft has defined CLS which are nothing but guidelines that language to follow so that it can communicate with other .NET languages in a seamless manner.

2. (i) Explain the execution model of the .NET framework.(6)

- The source code is compiled to IL while the metadata engine creates metadata information. IL and metadata are linked with other native code if required and the resultant IL code is saved.
- During execution, the IL code and any requirement from the base class library are brought together by the class loader.
- The combined code is tested for type-safety and then compiled by the JIT compiler to produce native machine code, which is sent to the runtime manager for execution.

(ii) Describe the components of the .Net framework and explain the features of each component (10)

- .NET Framework
- .NFT Building NFT Device
- Block Services Software
- .NET User .NET Infrastructure
- Experience and Tools
- Visual Studio .NET Experience .NET Services
- Windows .NET .NET Framework

Microsoft .NETplatform includes the following components that would help develop a new generation of smart internet services:

♦ .NET infrastructure and tools < .NET user experience
♦ .NET building block
♦ .NET device software

.NET Framework
.NET framework is one of the tools provided by the .NET infrastructure and tools component of the .NET platform. The .NET provides an environment for building, deploying and running web services and other applications. It consists of three distinct technologies.

♦ Common Language Runtime (CLR)
♦ Framework Base Classes
♦ User and program interfaces (ASP.NET and Winforms)

- .NET Framework
- ASP NET Windows Forms
  (Web Services) (User Interface)
  Framework Base Classes
- Common Laneuaee Runtime
  - **Common Language Runtime**
    CLR is the core of the .NET framework and is responsible for loading and running C# and other .NET language programs. It also support cross language interoperability.
    CLR activities
    The source code is compiled to IL while the metadata engine creates metadata information. IL and metadata are linked with other native code if required and the resultant IL code is saved. During execution, the IL code and any requirement from the base class library are brought together by the class loader. The combined code is tested for type-safety and then compiled by

the JIT compiler to produce native machine code, which is sent to the runtime manager for execution.

- **Common Type System (CTS)**
  The .NET framework provides multiple language support using the featu.e known as Common Type System that is built into CLR. The CTS supports variety of types and operations found in most programming languages and therefore calling one language from another does not require type conversions.
- **Common Language Specification (CLS)**
  The CLS defines a set of rules that enables interoperability on the .NET platform. These rules serve as a guide to third party compiler designers and library builders. The CLS is a subset of CTS and therefore the languages supporting the CLS can use each others class libraries as if they are tlHr own. APIs that are designed following the rules of CLS can easily be used by all the .NET languages.

- **Microsoft Intermediate Language (MSIL)**
  MS1L or IL, is an instruction set into which all the .NET programs are compiled. It is an assembly language and contains instructions for loading, storing, initializing and calling methods. When we compile any CLS compliant language program, the source code is compiled i.ito MSIL. Managed Code
  The code that satisfies the CLR at runtime in order to execute is referred to as managed code. For example C# compiler generates IL code (Managed code)
  Framework Base Classes

- ♦ Input/Output Operations
  ♦ String handling
  ♦ Security
  ♦ Windowing
  ♦ Windows messages
  ♦ Database management User ard Program Interfaces
  The .NET framework provides the following tools for managing user and application interfaces:
  Windows forms
  Web forms
  Console applications
  Web services
  These tools enable users to develop user friendly desktop as well as web based applications using a wide variety of languages on the .NET platform. Visual Studio .NET
  Visual studio.NET supports an Integrated Development environment (IDE)

3. EXPLAIN THE FOLLOWING
   i. For each ii. Structures iii. Arrays iv. Array list

(i) for .. each

The foreach statement 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 for which an iteration is currently being performed, in is a keyword.
- The expression must be an array or collection type and an explicit conversipn must exist from the element type of the collection to the type of the iteration variable.

Example:

Program Start

```
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 tor statement as follows:

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

Program End

Program to print the contents of a numerical array using foreach statement using System;

Program Start

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

Program End

The output of the program is 11 22 33 44

The advantage of foreach over the for statement is that it automatically detects the boundaries of the collection being iterated over

(ii) structures

- 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 arc required.
- Because they are value types stored on the stack, they have the following advantages compared to class objccts 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.

Defining a Struct

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

```
struct struct-name
{
data member 1; data member2;
}
```

Example:

```
struct Student v

{
pubile string Name;
public int RollNumber;
public double TotaMarks;
}
```

- 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 vo create variables of type Student.

Example:

Students si; //declarc a student

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

Assigning Values to Members

Member variables can be accessed using the simple dot notation as follows:
si.Name =”John”;
sl.RollNumber = 999;
sl.TotalMarks = 575.50;
We may also use the member variables in expressions on the right hand side.

Fxample: Final Marks = sl.TotalMarks + 5.0 ;

Copying Structs

We can also copy values from one struct to another. Example:
Students s2; // s2 is declared
s2 = si:
This will copy all those values from si to s2.

Note that we can also use the operator new to create struct variables.
Student s3 = new Student ();

- A struct variable is initialized to the default values of its members as soon as it is declared.
- The structs data members are ‘private’ by default and therefore cannot be accessed outside the struct definition.
- The use of access modifier public means that any one can use the member. A member not declared public cannot be accessed using the dot operator.

Example

```
struct ABC
{
int a ; //private by default
public int b;
private int c; //explicitly declared as private
}
```

(iii) arrays

- An array is a group of contiguous or related data items that share a commo name.
- The complete set of values is referred to as an array; the individua values are called elements.
- Arrays can be of any variable type.

```
ABC abc;
abc. a = 10;
abc. b = 20;

//Error, a is private !! QK, b is public
```

One-Dimensional Array

- A list of items can be given one variable name using only one subscript an such a variable is called a single-subscripted variable or a one-dimension; array.

Creating an Array

Like other variables, arrays must be declared and created in the comput- memory before they are used. Creation of an array involves three steps

1.Declaring the array
2.Creating memory locations
3.Putting values into the memory locations.

Declaration of Arrays

Arrays in C# are declared as follows:

type[ ] arrayname;

Examples:

int[ ] counter; //declare int array reference
float[ ] marks; //declare float array reference
int[ ] x, y; //declare two int array reference

- **Creation of Arrays**
  After declaring an array, we need to create it in the memory.
- C# allows us to create arrays using new operator only, as shown below:
  arraynsmc = new type[size];

Statement Result
int[ ] number number
points nowhere
number = new int [5];
points to int object
number [o]
number [1]
number [2]
number [3]
number [4]

Two-Dimensional Arrays

- There will be situations where a table of values will have to be stored Consider the following data table which shows the value of sales of thre< items by four salesgirls:
  int[,] my Array; my Array = new int [3 , 4];

(iv) array list

- System.Collections namespace defines a class known as Arraylist that car store a dynamically sized array of objects.

- The Arraylist class includes i number of methods to support operations such as sorting, removing anc enumerating its contents.
- It also supports a property count that gives tht number of objects in an array list and a property Capacity to modify oi read the capacity.
- An ArrayList is very similar to an array, except that it has the ability tc grow dynamically. We can create an arraylist by indicating the initial capacity we want.

Example:

ArrayList cities = new ArrayList (30);

It creates cities with a capacity to store thirty objects. If we do not specif) the size, it defaults to sixteen. That is

ArrayList citi es = new ArrayLi st ( );

Some important ArrayList Methods and Properties.

Add() —– Adds an object to a list
Clear()—- Removes all the elements from the list
Contains()—- Determines if an element is in the list
CopyTo()—- Copies a list to another
Insert —- Inserts an element into the list
RemoveO —- Removes the first occurrence of an element
RemoveAt() —- Removes the element at the specified place
RemoveRange()—- Removes a range of elements
SortQ —- Sorts the elements
Capacity —- Gets or sets the number of elements in the list
Count —- Gets the number of elements currently in the list

4. Explain in detail about various operators available in C#.

What is an Expression?

- The most basic expression consists of an operator, two operands and an assignment. The following is an example of an expression:

int theResult = 1 + 2;

- In the above example the (+) operator is used to add two operands (1 and 2) together.
- The assignment operator (=) subsequently assigns the result of the addition to an integer variable named theResult.
- The operands could just have easily been variables or constants (or a mixture of each) instead of the actual numerical values used in the example.

In the remainder of this chapter we will look at the various types of operators available in C#.

The Basic Assignment Operator

- We have already looked at the most basic of assignment operators, the = operator.
- This assignment operator simply assigns the result of an expression to a variable.
- In essence the = assignment operator takes two operands.
- The left hand operand is the variable to which a value is to be assigned and the right hand operand is the value to be assigned.
- The right hand operand is, more often than not, an expression which performs some type of arithmetic or logical evaluation. The following examples are all valid uses of the assignment operator:

```
x = 10;  // Assigns the value 10 to a variable named x

x = y + z; // Assigns the result of variable y added to variable z to variable x

x = y;   // Assigns the value of variable y to variable x
```

- Assignment operators may also be chained to assign the same value to multiple variables. For example, the following code example assigns the value 20 to the x, y and z variables:

```
int x, y, z;

x = y = z = 20;
```

C# Arithmetic Operators

- C# provides a range of operators for the purpose of creating mathematical expressions.
- These operators primarily fall into the category of binary operators in that they take two operands.
- The exception is the unary negative operator (-) which serves to indicate that a value is negative rather than positive.
- This contrasts with the subtraction operator (-) which takes two operands (i.e. one value to be subtracted from another).

int x = -10; // Unary - operator used to assign -10 to a variable named x

x = y - z; // Subtraction operator. Subtracts z from y

The following table lists the primary C# arithmetic operators:

Operator	Description
-(unary)	Negates the value of a variable or expression
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulo

Compound Assignment Operators

- C# provides a number of operators designed to combine an assignment with a mathematical or logical operation.
- These are primarily of use when performing an evaluation where the result is to be stored in one of the operands. For example, one might write an expression as follows:
  - x = x + y;
- The above expression adds the value contained in variable x to the value contained in variable y and stores the result in variable x. This can be simplified using the addition compound assignment operator:
  - x += y
- The above expression performs exactly the same task as x = x + y but saves the programmer some typing. This is yet another feature that C# has inherited from the C programming language.

Operator	Description
x += y	Add x to y and place result in x
x -= y	Subtract y from x and place result in x
x *= y	Multiply x by y and place result in x
x /= y	Divide x by y and place result in x
x %= y	Perform Modulo on x and y and place result in x
x &= y	Assign to x the result of logical AND operation on x and y
x |= y	Assign to x the result of logical OR operation on x and y
x ^= y	Assign to x the result of logical Exclusive OR on x and y

Increment and Decrement Operators

- Another useful shortcut can be achieved using the C# increment and decrement operators. As with the compound assignment operators described in the previous section, consider the following C# code fragment:

  x = x + 1; // Increase value of variable x by 1
  x = x - 1; // Decrease value of variable y by 1

- These expressions increment and decrement the value of x by 1. Instead of using this approach it is quicker to use the ++ and -- operators. The following examples perform exactly the same tasks as the examples above:
    - x++; Increment x by 1
      x--; Decrement x by 1

Comparison Operators

- In addition to mathematical and assignment operators, C# also includes set of logical operators useful for performing comparisons.
- These operators all return a Boolean (bool) true or false result depending on the result of the comparison.
- These operators are binary in that they work with two operands.
- Comparison operators are most frequently used in constructing program flow control.
- For example an if statement may be constructed based on whether one value matches another:
    - if (x == y)
    System.Console.WriteLine ("x is equal to y");
- The result of a comparison may also be stored in a bool variable. For example, the following code will result in a true value being stored in the variable result:

```
bool result;
int x = 10;
int y = 20;
result = x < y;
```

Operator	Description
x == y	Returns true if x is equal to y
x > y	Returns true if x is greater than y
x >= y	Returns true if x is greater than or equal to y
x < y	Returns true if x is less than y
x <= y	Returns true if x is less than or equal to y
x != y	Returns true if x is not equal to y

Boolean Logical Operators

- Another set of operators which return boolean true and false values are the C# boolean logical operators. These operators both return boolean results and take boolean values as operands.
- The key operators are NOT (!), AND (&&), OR (||) and XOR (^).
- The NOT (!) operator simply inverts the current value of a boolean variable, or the result of an expression.
- For example, if a variable named flag is currently true, prefixing the variable with a '!' character will invert the value to be false:

```
bool flag = true; //variable is true
bool secondFlag;
secondFlag = !flag; // secondFlag set to false
```

- The OR (||) operator returns true if one of its two operands evaluates to true, otherwise it returns false. For example, the following example evaluates to true because at least one of the expressions either side of the OR operator is true:

- if ((10 < 20) || (20 < 10)
- System.Console.WriteLine("Expression is true");

❖ The AND (&&) operator returns true only if both operands evaluate to be true. The following example will return false because only one of the two operand expressions evaluates to true:
  - if ((10 < 20) && (20 < 10)
  - System.Console.WriteLine("Expression is true");

❖ The XOR (^) operator returns true if one and only one of the two operands evaluates to true. For example, the following example will return true since only one operator evaluates to be true:
  - if ((10 < 20) ^ (20 < 10))
  - System.Console.WriteLine("Expression is true");

The Ternary Operator

❖ C# uses something called a ternary operator to provide a shortcut way of making decisions. The syntax of the ternary operator is as follows:

  [condition] ? [true expression] : [false expression]

  - int x = 10;
  - int y = 20;
  - System.Console.WriteLine(x > y ? x : y);

5. (i) Explain about various value and reference types supported by c# (8)

VALUE TYPES

The value types of C# can be grouped into two categories, namely
♦ User-defined types (or Complex types) and
♦ Predefined types (or simple types)
We can define our own complex types known as user-defined value types which include struct types and enumerations.
Predefined value types which are also known as simple types (or primitive types) are further subdivided into
Numeric Types
Boolean Types
Character Types
Numeric types includes integral types, floating point types and decimal types.

Integer Types

❖ Integer types can hold whole numbers such as 123, -96 and 5639. The size of the values that ran hr stored depends on the integral data types we choose.

❖ C# supports the concept of unsigned types and therefore it supports eight types of integers Signed Integers
Signed integer types can hold both positive and negative numbers.

❖ It should be remembered that wider data types require more time for manipulation and therefore it is advisable to use smaller data types wherever possible.

❖ For example, instead of storing a number like 50 in an int type variable, we must use a sbyte variable to handle this number.

Floating-Point Types

The floating point types are used to hold numbers containing fractional parts such as 27.56 and -1.375.

- There are two kinds of floating point storage in C#
- Double-precision types arc used when we need greater precision in storage of floating-point numbers.
  Floating point data types support a special value known as Not-a- Number(NaN)

Decimal Types

The decimal types is a high precision, 128-bit data types that is designed for use in financial and monetary calculations.

- It can **store values in the range 1.0×10-28 to 7.9 x 10 28 with 28 significant digits.To specify a number to be decimal type, we must append the charactcr M (or m) to the value, like I23.34M.**
- **. Boolean Type**
  Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a Boolean type can take : true or false.
- Both these words have been declared as keywords.
- Boolean type is denoted by the keyword bool and uses only one bit of storage

REFERENCE TYPES

The reference types of C# can be grouped into two categories, namely

♦ User-defined types (or Complex types) and

♦ Predefined types (or simple types)

User-defined reference types refer to thtffce which we define using predefined types. These include.

1. Classes
2. Interfaces
3. Delegates
4. Arrays

Predefined reference types include two data types:

1. Object type
2. String type

Classes

A class is a template that defines the form ol an object. It specifies both the data and the code that will operate on that data. C# uses a class specification to construct objects. Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object.

interfaces

C# provides an alternative 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.

delegates

C# implements the callback technique in a much safer and more object- oriented manner, using a kind oi object called delegate object.

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. Arrays
An array is a group of contiguous or related data items that share a common name. The complete set of values is referred to as an array; the individual values are called elements.

Arrays can be of any variable type.

Object Types
C# uses a class specification to construct objects. Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object.
String Objects
Strings represent a sequence of characters. The easiest way to represent a sequence of characters in C# is by using a character array. C# supports two types of strings, namely, immutable strings and mutable strings.

(ii) What is jagged array? Explain its use with simple example (8)

- C# also allows you to create a special type of two-dimensional array called a jagged array. A jagged array is an array of arrays in which the length of each array can differ.

- Thus, a jagged array can be used to create a table in which the lengths of the rows are not the same.
  Jagged arrays are declared by using sets of square brackets to indicate each dimension.
- For example, to declare a two-dimensional jagged array, you will use this general form:
  typef] [] array-name = new type[size][ ];
- int[][] jagged = new int[3][];
- jagged[0] = new int[4];
- jagged[1] = new int[3];
- jagged[2] = new int[5];

```
// Demonstrate jagged arrays,
using System;
class Jagged
{
public static void MainQ
{
 int[][] jagged = new int[3][]; jagged[0] = new int[4]; jagged[l] = new int[3]; jagged[2] = new int[5]; int i;
// display values in first array for(i=0; i < 4; i++) Console. Write(jagged[0][i] + " ");
Console. WriteLine();
// display values in second array for(i=0; i < 3; i++) Console. WriteGagged[ 1 ][i] + " ");
Console. WriteLine();
H display values in third array for(i=0; i < 5; i++) Console. Write(jagged[2][i] + " ");
Console. WriteLine();
}
}
```

The output is shown here: 0 123 0 1 2 0 1234
jagged arrays are arrays of arrays, there is no restriction that requires that the arrays be single-dimensional. For example, the following creates an array of two-dimensional arrays:
int[][J jagged = new int[3][J;
The next statement assigns jagged[0] a reference to a 4×2 array:

jagged[0] = new int[4,2];
The following statement assigns a value to jagged[0] [1,0]:
jagged[0][l,0] = i;

6. Explain about Enumerators and structures in C#.

Enumerators

An enumeration is a set of named integer constants. The keyword enuni declares an enumerated type. The general form for an enumeration is Here, the type name of the enumeration is specified by name. The enumeration list is a comma-separated list of identifiers, enum name { enumeration list }; Here is an example. The following code fragment defines an enumeration called Apple that enumerates various types of apples:

Program Start

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland, Mcintosh };
 Program to demonstrate the enumeration
using System;
class EnumDemo {
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland, Mcintosh };
public static void Main() { string[] color = { “Red”, “Yellow”,
“Red”, “Red”, “Red”,
“Reddish Green”
};
Apple i; // declare an enum variable
// use i to cycle through the enum
for(i = Apple. Jonathan; i // use an enumeration to index an array
for(i = Apple.Jonathan; i }
}
```

Program End

The output from the program is shown here:
Jonathan has value of 0
GoldenDel has value of 1
RedDel has value of 2
Winesap has value of 3
Cortland has value of 4
Mcintosh has value of 5
Color of Jonathan is Red
Color of GoldenDel is Yellow
Color of Mcintosh is Reddish Green

struct in C# .NET

- Struct is an encapsulated entity. Struct doesn't uses complete oops concept but are used for user defined data type. All the member of the struct has to be initialized, as it is value type.

- A struct is a simple user-defined type, a lightweight alternative to a class. A structure in C# is simply a composite data type consisting of a number elements of other types.

- Similar to classes, structures have behaviors and attributes. As a value type, structures directly contain their value so their object or instance is stored on the stack.

- Struts support access modifiers, constructors, indexers, methods, fields, nested types, operators, and properties.

How do define struct?

```
public struct Student
      {
          int id;
          int zipcode;
          double salary;
}
```

Some points about structs

- Struct is used to improve the performance and clarity of code.
- Struct uses fewer resources in memory than class.
- When we have small and frequent use of some work use structs over classes.
- Performance can suffer when using structures in situations where reference types are expected due to boxing and unboxing.
- You should pass structs to method as ref parameters in order to avoid the performance loss associated with copying data.
- Structs reside on the stack, so we should keep them small.
- Structs can't be inherited and we can say they are sealed.
- Structure implicitly inherits from System.ValueType.
- The default constructor of a structure initializes each field to a default value. You cannot replace the default constructor of a structure.
- You can't define destructor for structs
- Structs can be inherited from an interface?

Structs and inheritance

- Structs don't provide inheritance. It is not possible to inherit from a struct and a struct can't derive from any class.
- Once exception that all type in C# is derive from the class System.Object, so structs also have the access to the methods etc., of System.Object.

Note: Inheritance is indirect: Structs derive from System.ValueType, which in turns derive from System.Object.ValueType adds no new method of its own, but provides overrides of some the Object methods that are more appropriate to value types.

Difference between structs and classes

structs	classes
❖ structs are value type	❖ classes are reference type
❖ structs are stored in stack or a inline	❖ classes are stored on managed heap
❖ structs doesn't support inheritance	❖ classes support inheritance
❖ But handing of constructor is different in structs. The complier supplies a default no-parameter constructor, which your are not permitted to replace	❖ Constructors are fully supported in classes

UNIT – II

PART – A

1. What is the use of static constructor in C#?
2. What is the use of 'new' in inheritance?
3. Distinguish between ref and out parameters
4. What is inclusion polymorphism?
5. What is exception? How it is handled in C#?
6. What is the use of finally block?
7. Define delegate and event handlers?
8. Compare new with override modifier.
9. What are multicast delegates?
10. What is static member & static method?

PART – B

1. Explain clearly the use of interfaces Use a banking system as a programming example.
2. (i)List out the exception handling statements supported in C# and explain with an example. [8]
   (ii) What is the use of 'is' operator in interfaces? Explain. [8]
3. (i) Consider a class distance which stores a distance value using kilometer and meter. Overload the + operator to add two distance objects.
   (ii) Explain in detail about the concept of operator overloading.
4. Explain in detail, using an appropriate programming example, the use of events
5. Discuss about inheritance and polymorphism in detail
6. Explain creating and using delegates with example.

UNIT – II

PART – A

1. What is the use of static constructor in C#?

a. A static constructor is called before any object of the class is created
b. It is usually used to assign initial values to static data members.
c. A static constructor is dcclarcd by prefixing a static keyword to th constructor definition.
Example:
Program Start :
Class string {
Static string() //No parameters should be used {
// class can have only one static constructor.}
}
Program End

2. What is the use of 'new' in inheritance?

- It is used in accessibility of baseclass members. It is used to hide the inherited member. It is used in a non-nested class declarations. It is used on an abstract class.

3. Distinguish between ref and out parameters

- A reference parameter does not create a new storage location. A parameter declared with the ref modifier is a reference parameter. Example:
  void Modify(ref int x)
- The output parameters are used to pass results back to the calling method. This is achieved by declaring parameters with an out keyword. An output parameter does not create a new storage location.

4. What is inclusion 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
- It is also known as dynamic binding because the selection of the appropriate method is done dynamically at runtime

5. What is exception? How it is handled in C#?

- An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally** and **throw**.

6. What is the use of finally block?

finally:

- The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

7. Define delegate and event handlers?

- C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.
- To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you are handling.

```
public delegate void EventHandler(object  Source, EventArgs e);
```

8. Compare new with override modifier.

New	override

❖ The new modifier instructs the compiler to use your child class implementation instead of the parent class implementation.
❖ Any code that is not referencing your class but the parent class will use the parent class implementation.
 	
public class Base
{
  public virtual void DoIt()
  {
  }
}
public class Derived : Base
{
  public new void DoIt()
  {
  }
}
Base b = new Derived();
Derived d = new Derived();
b.DoIt();
d.DoIt();
	❖ The override modifier may be used on virtual methods and must be used on abstract methods.
	❖ This indicates for the compiler to use the last defined implementation of a method. Even if the method is called on a reference to the base class it will use the implementation overriding it.
	public class Base
	{
	  public virtual void DoIt()
	  {
	  }
	}
	public class Derived : Base
	{
	  public override void DoIt()
	  {
	  }
	}
	Base b = new Derived();
	b.DoIt();

9. What are multicast delegates?

Multicast delegate:
- If a delegate is used to refer more than one function is known as multi cast delegate
- They are nothing but a single delegate that can invoke multiple methods of matching signature.
- Derives from **System.MulticastDelegate** class which is a subclass of **System.Delegate**
- In this, we create a single delegate that in turn invokes multiple encapsulated methods.
- We can use Multi-cast Delegates when multiple calls to different methods are required.

10. What is static member & static method?

Static Data Members
- A C# class can contain both static and non-static members. When we declare a member with the help of the keyword static, it becomes a static member.
- A static member belongs to the class rather than to the objects of the class. Hence static data members are also known as class members and non-static members are known as instance members.
- Static members and preloaded in memory while instance members are post loaded in memory.

Static Methods
- When a method is declared as static then that method can access only other static members available in the class and it is not possible to access instance members

PART – B

1. Explain clearly the use of interfaces Use a banking system as a programming example.
- Interface is nothing but an contract of the system which can be implemented on accounts.
- .Net doesn't support the multiple inheritance directly but using interfaces we can achieve multiple inheritance in .net.
- An Interface can only contains abstract members and it is a reference type.
- Interface members can be Methods, Properties, Events and Indexers. But the interfaces only contains declaration for its members.
- Class which inherits interface needs to implement all it's methods.
- All declared interface member are implicitly public.

Declaring Interfaces

- Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

```
public interface ITransactions
{
  // interface members
  void showTransaction();
  double getAmount();
}
```

Banking system as a programming example

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

namespace InterfaceApplication
{

  public interface ITransactions
  {
    // interface members
    void showTransaction();
    double getAmount();
  }
  public class Transaction : ITransactions
  {
    private string tCode;
    private string date;
    private double amount;
    public Transaction()
    {
      tCode = " ";
      date = " ";
      amount = 0.0;
    }
    public Transaction(string c, string d, double a)
    {
      tCode = c;
      date = d;
      amount = a;
    }
    public double getAmount()
    {
      return amount;
    }
    public void showTransaction()
    {
      Console.WriteLine("Transaction: {0}", tCode);
      Console.WriteLine("Date: {0}", date);
      Console.WriteLine("Amount: {0}", getAmount());

    }

  }
  class Tester
  {
    static void Main(string[] args)
    {
      Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
```

```
        Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
        t1.showTransaction();
        t2.showTransaction();
        Console.ReadKey();
      }
    }
}
```

2. (i)List out the exception handling statements supported in C# and explain with an example. [8]

Exception Class Cause of Exception

SystemBxception – Failure to access a type member, such as a method or field

- AccessException – An argument to a method was valid
- ArgumentException – A null argument was passed to a method that does not accept it
- ArgumentNullException – Argument value is out of range
- ArgumentOutofRangeException – Arithmetic over-or underflow has occurred
- ArithmeticException – Attempt to store the wrong type of object in an array
- ArrayTypeMismatchExccption – Image is in the wrong formal
- BadlmageFormatException – Base class for exceptions thrown by the runtime
- CoreException – An attempt was made to cast to an invalid class
- DivideByZerroException – The format of an argument is wrong
- FormatException – An array index is out of bounds
- IndexOutofRangeException – An attempt was made to cast to an invalid class
- InvalidCastException – A method was called at an invalid time
- InvalidOperationException – An invalid version of a DLL was accessed
- MissingMemberException – A number is not valid
- NotFiniteNumberException – Indicates that a method is not implemented by a class
- NotSupportException – Attempt to use an unassigned reference Not enough memory to continue execution A stack has overflowed
- NullReferenceException

OutofMemoryException
StackOverflowException
A failed run-time check; used as a base class forothei Exception

Program Start :
using System;
class Errors
i i

```
public static void Main() {
int [ ] a = 10; intb = 5;
try
{
int x = a [ 2] / (b – a [ 1] ; // Exception here
}
catch (ArithmeticException e)
{
Console.WriteLine(“Division by zero”);
}
catch (IndexOutofRangeException e)
{
Console.WriteLine(“Array index error”);
}
catch (Array TypeMismatohException e)
{
Conso!e.WriteLine(“Wiong data type”);
}
int y = a [ 1] /a[0]; Console. WriteLine(“y = ” + y);
}
}
```

Program End

(ii) What is the use of ‘is’ operator in interfaces? Explain. [8]

- You would like to be able to ask the object if it supports the interface, i then invoke the appropriate methods.
- In C# there arc two ways i accomplish this. The first method is to use the is operator. The form of tl is operator is:
- The is operator evaluates true if the expression (which must be a referent type) can be safely cast to type without throwing an exception.
- The examp illustrates the use of the is operator to test whether a Document implement the IStorable and ICompressible interfaces.

Program Start :

```
#region Using directives
using System;
using System.Collections.Generic; using System.Text; #endregion
namespace IsOperator
{
interface IStorable
{
void Read();
void Write( object obj);
int Status { get; set; }
}
// here’s the new interface
interface ICompressible
{
void Compress(); void Decompress();
/ Document implements IStorable
```

```
public class Document: IStorable
{
private int status = 0;
public Document( string s ) {
Console. WriteLine( “Creating document with: {0}”, s );
}
// IStorable.Read
public void Read() {
Console.WriteLinc( “Reading…”);
}
// IStorable.Write public void Write( object o ) (
Console.WriteLine( “Writing…”);
}
// IStorable.Status
public int Status {
get- {
return status;
set (
status = value;
}
}
// derives from Document and implements ICompressible
public class CompressibleDocument: Document, ICompressible
{
public CompressibleDocument(String s): base(s)
{}
public void Compress() {
Console. WriieLine(“Compressing…”);
}
public void Decompress() {
Console. WriteLine(“Dccompressing…”);
}
}
public class Tester
{
static void Main( ) {
// A collection of Documents Document[] docArray = new Document[2];
// First entry is a Document
docArray[0] = new Document( “Test Document” );
// Second entry is a CompressibleDocument (ok because // CompressibieDocument is a
Document) docArray[ 1 ] =
new CompressibleDocument(“Test compressibleDocument”);
// don’t know what we’ll pull out of this hat foreach (Document doc in docArray)
X
II report your name Console.WriteLine(“Got: {0}”, doc);
// Both pass this test
if (doc is IStorable)
{
IStorable isDoc = (IStorable)doc; isDoc.Read();
}
```

```
// fails for Document
// passes for CompressibleDocument
if (doc is ICompressible)
{
ICompressible icDoc = (ICompressible)doc; icDoc.Compress( );
}
}
```

Program End

3. (i) Consider a class distance which stores a distance value using kilometer and meter. Overload the + operator to add two distance objects.

Program Start:

```
using System;
using System.Collections.Generic;
using System.Linq;
 using System.Text;
namespace ConsoleApplication 12
{
public class Distance {
private int kilometer; private int meter;
public Distance(int km, int m)
{
kilometer = km; meter = m;
}
public static Distance operator ^(Distance lhs, Distance rhs)
{
int km = lhs.kilometer + rhs.kilometer;
 int m = lhs.meter + rhs.meter;
if (m >= 1000)
m -= 1000; kmK;
return new Distance(km, m);
}
public void display() {
Console.WriteLine("{0} km and {1} m", kilometer, meter);
class Program {
static void Main(string[] args)
{
Distance dl = new Distance(4, 500);
Distance d2 = new Distance(5, 750);
Distance d3 = dl + d2;
Console.WriteLine("Distance d 1″); dl.display();
Console.WriteLine("Distance d2″); d2.display();
Console. WriteLine("Distance d3″); d3.display();
}
}
}
```

Program End

(ii) Explain in detail about the concept of operator overloading.

- The mechanism of giving a special meaning to a Standard c# operator with respect to a user defined data type such as classes or structures is known as OPERATOR OVERLOADING.
- All C # binary operators can be over loaded. i.e +, - , *, / , %,&,|, <<,>>.
- All C# unray operators can be over loaded. i.e +,_,!,++,-- .
- All relational operators can be over loaded , but only as pairs. i.e = =, !=, <>, <=, >=

In simple terms let us say if + is overloaded

if you want to find a + b
if a,b are integer result will be sum of a and b and result is int.
if a,b are flot result will be sum of a and b and result is flot.
if a,b are two strings result will be concatenation of String a and String b and final result is string.

In this way + is overloaded with different operations depending up on the data.

4. Explain in detail, using an appropriate programming example, the use of events

- Events are basically a user action like key press, clicks, mouse movements etc., or some occurrence like system generated notifications.
- Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.
- Event: A hook on an object where code outside of the object can say "When that something-something happens, that fires this event, please call my code"
- Event Handler: The code that is called when the event fires
- Firing an event: Basically the same as calling it, it's just a different word for essentially the same thing

Using Delegates with Events

- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.
- The class containing the event is used to publish the event. This is called the **publisher** class.
- Some other class that accepts this event is called the **subscriber** class.
- Events use the **publisher-subscriber** model.
-

Publisher

- A **publisher** is an object that contains the definition of the event and the delegate.The event-delegate association is also defined in this object.
- A publisher class object invokes the event and it is notified to other objects.

Subscriber

- A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events

To declare an event inside a class, first a delegate type for the event must be declared. For example,

```
public delegate void BoilerLogHandler(string status);
```

Next, the event itself is declared, using the **event** keyword:

```
//Defining event based on the above delegate
public event BoilerLogHandler BoilerEventLog;
```

Why use Events or publish / subscribe?

- Any number of classes can be notified when an event is raised.
- The subscribing classes do not need to know how the Metronome (see code below) works, and the Metronome does not need to know what they are going to do in response to the event
- The publisher and the subscribers are decoupled by the delegate.
- This is highly desirable as it makes for more flexible and robust code.
- The metronome can change how it detects time without breaking any of the subscribing classes. The subscribing classes can change how they respond to time changes without breaking the metronome.
- The two classes spin independently of one another, which makes for code that is easier to maintain.

```
using System;

class Test
{
   public event EventHandler MyEvent
   {
      add
      {
         Console.WriteLine ("add operation");
      }

      remove
      {
         Console.WriteLine ("remove operation");
      }
   }
   static void Main()
   {
      Test t = new Test();

      t.MyEvent += new EventHandler (t.DoNothing);
      t.MyEvent -= null;
   }
   void DoNothing (object sender, EventArgs e)
   {
   }
}
```

5. Discuss about inheritance and polymorphism in detail

Inheritance in C#

Creating a new class from existing class is called as inheritance.

```
//All the car properties can be used by the supercar
class SuperCar : car
{
}
```

- When a new class needs same members as an existing class, then instead of creating those members again in new class, the new class can be created from existing class, which is called as inheritance.
- Main advantage of inheritance is reusability of the code.
- During inheritance, the class that is inherited is called as base class and the class that does the inheritance is called as derived class and every non private member in base class will become the member of derived class.

Syntax

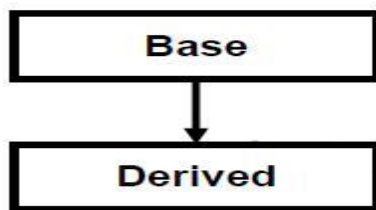
```
[Access Modifier] class ClassName : baseclassname

{

 }
```

Inheritance can be classified to 5 types.

- Single Inheritance
- Hierarchical Inheritance
- Multi Level Inheritance
- Hybrid Inheritance
- Multiple Inheritance

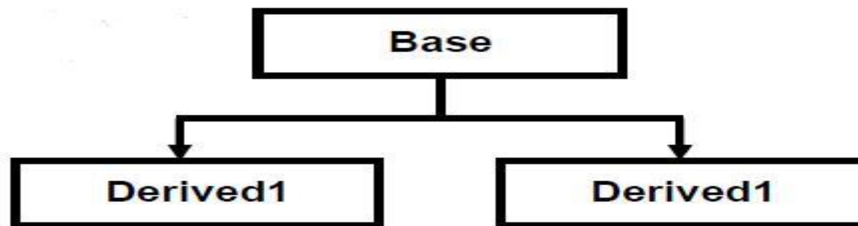
1. **Single Inheritance**

when a single derived class is created from a single base class then the inheritance is called as single inheritance.



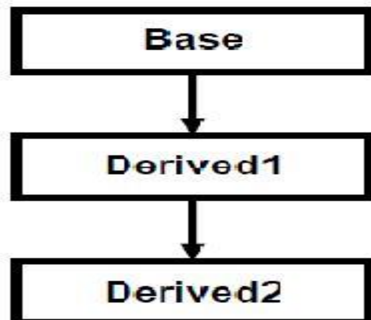
2. Hierarchical Inheritance

when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



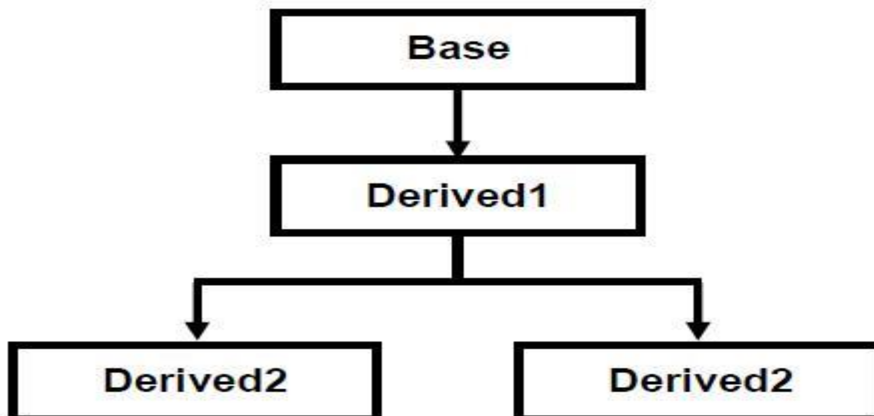
3. Multi Level Inheritance

when a derived class is created from another derived class, then that inheritance is called as multi level inheritance.



4. Hybrid Inheritance

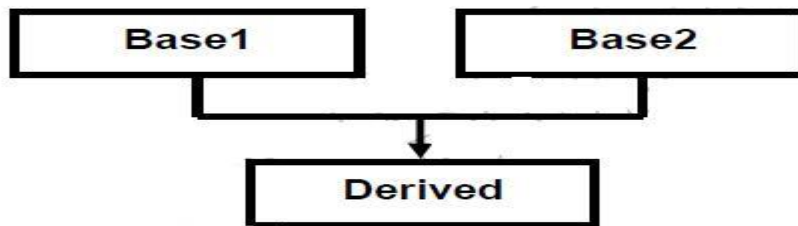
Any combination of single, hierarchical and multi level inheritances is called as hybrid inheritance.



5. Multiple Inheritance

when a derived class is created from more than one base class then that inheritance is called as multiple inheritance.

But multiple inheritance is not supported by .net using classes and can be done using interfaces.



Handling the complexity that causes due to multiple inheritance is very complex. Hence it was not supported in dotnet with class and it can be done with interfaces.

```
using System;
namespace ProgramCall
{
   class BaseClass
   {
      //Method to find sum of give  2 numbers
      public int FindSum(int x, int y)
      {
         return (x + y);
      }
      //method to print given 2 numbers
      //When declared protected , can be accessed only from inside the derived class
      //cannot access  with the instance of  derived class
      protected void Print(int x, int y)
      {
         Console.WriteLine("First Number: " + x);
         Console.WriteLine("Second Number: " + y);
      }
   }

   class Derivedclass : BaseClass
   {
      public void Print3numbers(int x, int y, int z)
      {
         Print(x, y); //We can directly call baseclass members
         Console.WriteLine("Third Number: " + z);
      }

   }
class MainClass
   {
      static void Main(string[] args)
      {
         //Create instance for derived class, so that base class members
         // can also  be accessed
         //This is possible because  derivedclass is inheriting base class
         Derivedclass instance = new Derivedclass();
         instance.Print3numbers(30, 40, 50); //Derived class internally calls base class method.
         int sum = instance.FindSum(30, 40); //calling base class method with derived class instance
```

```
            Console.WriteLine("Sum : " + sum);
            Console.Read();
        }

    }
}
```

Polymorphism

- Polymorphism means having more than one form. **Overloading** and **overriding** are used to implement polymorphism.
- Polymorphism is classified into compile time polymorphism or early binding or static binding and Runtime polymorphism or late binding or dynamic binding.

Method Overloading

- The process of creating more than one method in a class with same name or creating a method in derived class with same name as a method in base class is called as method overloading.
- In VB.net when you are overloading a method of the base class in derived class, then you must use the keyword "Overloads".
- But in C# no need to use any keyword while overloading a method either in same class or in derived class.
- While overloading methods, a rule to follow is the overloaded methods must differ either in number of arguments they take or the data type of at least one argument.

Example for Method Overloading

```
using System;
namespace ProgramCall
{
    class Class1
    {
        public int Sum(int A, int B)
        {
            return A + B;
        }
        public float Sum(int A, float B)
        {
            return A + B;
        }
    }

    class Class2 : Class1
    {
        public int Sum(int A, int B, int C)
        {
            return A + B + C;

        }
    }
    class MainClass
    {
        static void Main()
        {

            Class2 obj = new Class2();
```

```
                Console.WriteLine(obj.Sum(10, 20));

                Console.WriteLine(obj.Sum(10, 15.70f));

                Console.WriteLine(obj.Sum(10, 20, 30));

                Console.Read();
            }
        }
    }
```

Method Overriding

- Creating a method in derived class with same signature as a method in base class is called as method overriding.
- Same signature means methods must have same name, same number of arguments and same type of arguments.
- Method overriding is possible only in derived classes, but not within the same class.
- When derived class needs a method with same signature as in base class, but wants to execute different code than provided by base class then method overriding will be used.
- To allow the derived class to override a method of the base class, C# provides two options, **virtual** methods and **abstract** methods.

```
using System;
namespace methodoverriding
{
  class BaseClass
  {
    public virtual  string YourCity()
    {
      return "New York";
    }
  }
  class DerivedClass : BaseClass
  {
    public override string YourCity()
    {
      return "London";
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      DerivedClass obj = new DerivedClass();
      string city = obj.YourCity();
      Console.WriteLine(city);
      Console.Read();
    }
  }
}
```

Compile time Polymorphism or Early Binding

- The polymorphism in which compiler identifies which polymorphic form it has to execute at compile time it self is called as compile time polymorphism or early binding.

- Advantage of early binding is execution will be fast. Because every thing about the method is known to compiler during compilation it self and disadvantage is lack of flexibility.
- Examples of early binding are overloaded methods, overloaded operators and overridden methods that are called directly by using derived objects.

Runtime Polymorphism or Late Binding

- The polymorphism in which compiler identifies which polymorphic form to execute at runtime but not at compile time is called as runtime polymorphism or late binding.
- Advantage of late binding is flexibility and disadvantage is execution will be slow as compiler has to get the information about the method to execute at runtime.
- Example of late binding is overridden methods that are called using base class object.

6. Explain creating and using delegates with example.

- The dictionary meaning of delegate is "a person acting for another person".
- In C#, ii really means a method acting for another method. 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
  ♦ public
  ♦ protected
  ♦ internal
  ♦ private
- delegate void Simpl cDelcgate();
- delegate int MathOperation(int x, int y);
- public delegate int Compareltems (cbiect ol, obiect o2);
- private delegate string GetAString ();
- delegate double DoubleOperation (doubl e x);

- That is, a delegate may be defined in the following places:
  ♦ Inside a class
  ♦ Outside all classes
  ♦ As the top level object in a namespace

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.
❖ One feature of delegates is that they are type-safe to the extent that they ensure the matching of signatures of the delegate methods.
♦ What type of object the method is being called against, and
♦ Whether the method is a static or an instance method. For instance, the delegate

```
delegate string GetAString()
int N = 100
GctAString si = new GetAString(N.ToString);
 delegate void Delegatel ();
public void Fl() //instance method
 {
Console. WritcLine( ” F1 ” );
}
static public void F2() //static method .
{
Console. WriteLine (“F2″);
}
```

Delegate Instantiation

❖ 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 (express ion)
❖ The delegate-type is the name of the delegate declared earlier whose object is to be created.
❖ The expression must be a method name or a value of a delegate-type .
❖ If it is a method name its signature and return type must be the same as those of the delegate.

```
Program Start :
//delegate declaration
delegate int ProductDelegate (int x, int y);
class Delegate {
static float Product (float a, float b) //signature does
// not match
f – *
{
return ( a * b);
\
static nit Product (int a, int b) // signature matches
/ \
return ( a * b);
}
// delegate instantiation
ProductDelegate p = new ProductDelegate(Product);
}
```

Delegate Invocation

When a delegate is invoked, it in turn invokes the method whose reference has been encapsulated into the delegate, (only if their signatures match).

Delegateobject (parameters list)

♦ 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:

delegatel(x, y); / /void delegate

```
Program Start :
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 operationl = new ArithOp (MathOperation.Add);
ArithOp operation2 = new ArithOp (MathOperation.Sub);
// Invoking delegates
int resultl = operationl(200,100);
int result2 = operation2 (200,100);
Console. WriteLine ("Result 1 = " + result 1);
Console. WriteLine ("Result2 = " + result2); }
Program End
```

If we need to implement more delegate methods, we have to create more delegate objects. In such cases, we may create an array of delegate objects and then use them in a for loop to invoke the methods. Example

```
ArithOp [ ] operation= {new ArithOp(MathOperation.Add), new ArithOp(MathOPeration.Sub) };
```

This creates two delegates, operation[0] to invoke Add method and operation [1] to invoke Sub method. It is also allowed to use delegate objects as method parameters. For instance,

```
ProcessMethod(operation[0], 200, 100);
```

Is valid. This statement invokes the method ProcessMethod using three parameters including the delegate object operationl. The ProcessMethod would be defined as under:

```
static void ProcessMethod (ArithOp operation, int x, int y)
{
int resultl = operation(x,y);
Console. WriteLine ("Resultl = " + result);
}
```