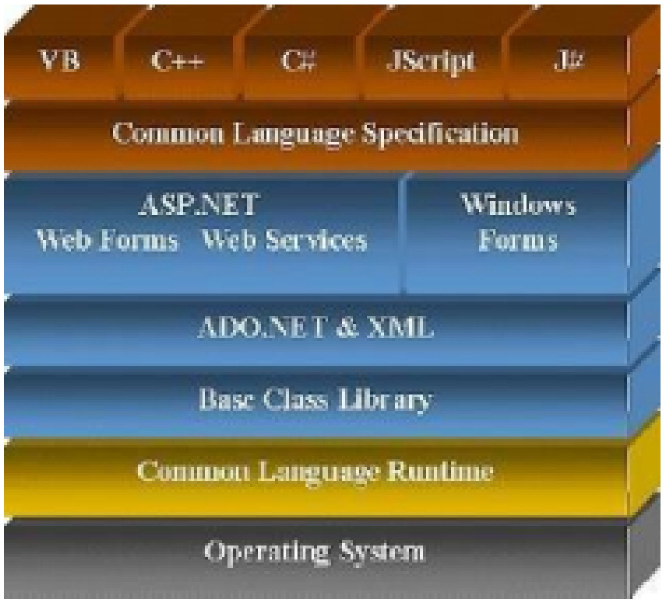
What is .NET Framework? Explain its architecture with help of diagram.



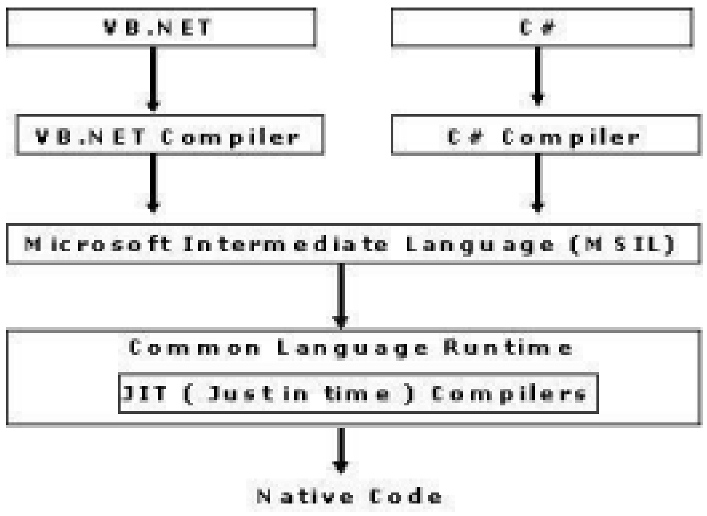
.NET Framework is a platform created by Microsoft for building, deploying, and running applications andservicesthatuse .NETtechnologies,such as desktop applications andWeb services.

▪ It is a platform for application developers.

▪ It is a Framework thatsupportsMultipleLanguage and Crosslanguage integration. ▪
It hasIDE (Integrated Development Environment).
▪ Framework is a set of utilities or can say building blocks of our application system. ▪
.NETFrameworkprovidesinteroperabilitybetweenlanguagesi.e.CommonTypeSystem(CTS). ▪ .NET Framework also includes the .NET Common Language Runtime (CLR), which responsible for maintaining the execution of all applications developed using the .NETlibrary.
▪ The .NET Framework consists primarily of a gigantic library of code.

Components of the .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.
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.


VB.NET
C#
VB.NET Compiler
C# Compiler
Microsoft Intermediate Language (MSIL)
Common Language Runtime
JIT (Just in time) Compilers
Native Code

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)

The .NET Framework includes a set of standard class libraries. A class library is a collection of methods and functions that can be used for the core purpose.

For example, there is a class library with methods to handle all file-level operations. So there is a method which can be used to read the text from a file. Similarly, there is a method to write text to a file.

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. 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. Languages –

The types of applications that can be built in the .Net framework is classified broadly into the following categories.

· WinForms – This is used for developing Forms-based applications, which would run on an end user machine. Notepad is an example of a client-based application. · Web Application – This is used for developing web-based applications, which are made to run on any browser such as Internet Explorer, Chrome or Firefox. o The Web application would be processed on a server, which would have Internet Information Services Installed.
Internet Information Services or IIS is a Microsoft component which is used to execute an Asp.Net application.
The result of the execution is then sent to the client machines, and the output is shown in the browser.

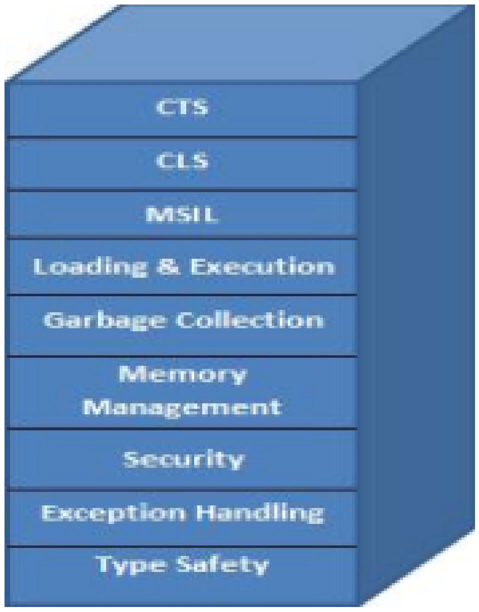
ADO.Net – This technology is used to develop applications to interact withDatabases such as Oracle or Microsoft SQL Server.

Type of .NET Languages

- To help create languages for the .NET Framework, Microsoft created the Common Language Infrastructure specification (CLI). The CLI describes the features that each language must provide in order to use the .NET Framework and common language runtime and to interoperate with components written in other languages. If a language implements the necessary functionality, it is said to be .NET-compliant.
- Every .NET-compliant language supports the same data types, uses the same .NET Framework classes, compiles to the same MSIL, and uses a single common language runtime to manage execution. Because of In addition, components written in one language can easily interoperate with components written in another language. For example, you can write a class in C# that inherits from a base class written in Visual Basic. The .NET Framework was developed so that it could support a theoretically infinite number of development languages. Currently, more than 20 development languages work with the .NET Framework. C# is the programming language specifically designed for the. NET platform, but C++ and Visual Basic have also been upgraded to fully support the .NET framework. The following are the commonly used languages provided by the Microsoft:
  • VC++
  • VB.NET
  .C#
  • J#
  • JScript .NET
  VC++
  VB.NET 4

CLR Common Language Runtime ->it is the Virtual Machine component that manages the execution of programs written in any language that uses the .NET Framework, for example C#, VB.Net, F# and so on. We can say that it is heart and soul of .Net Framework or backbone.

- Programmers write code in any language, including VB.Net, C# and F# then they compile their programs into an intermediate form of code called CLI in a portable execution file (PE) that can be managed and used by the CLR and then the CLR converts it into machine code to be will executed by the processor.
- The information about the environment, programming language, its version and what class libraries will be used for this code are stored in the form of metadata with the compiler that tells the CLR how to handle this code.
- The CLR allows an instance of a class written in one language to call a method of the class written in another language.



Components of the CLR:

CTS

Common Type System(CTS) describes a set of types that can be used in different. Net languages in common. That is, the Common Type System (CTS) ensure that objects written in different .Net languagescaninteractwitheachother.ForCommunicatingbetweenprogramswritteninany.NET complaint language, the types must be compatible on the basic level.
These types can be Value Types or Reference Types. The Value Types are passed by values and stored in the stack. The Reference Types are passed by references and stored in the heap.

CLS

CLS stands for Common Language Specification and it is a subset of CTS. It defines a set of rules and restrictions that every language must follow which runs under .NET framework. The languages which follow these set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration or Interoperability.

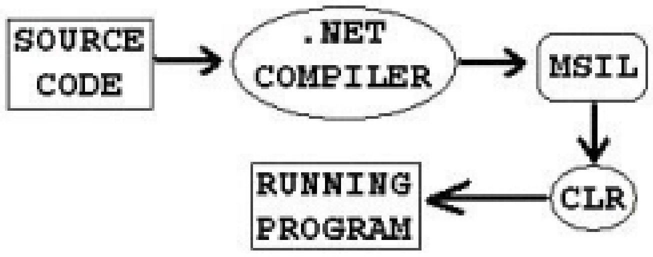
For Example

if we take C-Sharp and VB.net in C# each statement must have to end with a semicolon it is also called a statement Terminator but in VB.NET each statement should not end with a semicolon(;). So, these syntax rules which we have to follow from language to language differ but CLR can understand all the language Syntax because in.NET each language is converted into MSIL code after compilation and the MSIL code is language specification of CLR.

Garbage Collector: It is used to provide the Automatic Memory Management feature. Suppose if there is no garbage collector then programmers have to write the memory management codes which will be a kind of overhead on programmers.

JIT(Just In Time Compiler): It is responsible for converting the CIL(Common Intermediate Language)into machine code or native code using the Common Language Runtime environment.

MSIL->It is language independent code. When you compile code that uses the. NET Famework library, we don't immediately create operating system -specific native code. Instead, we compile our code into Microsoft. Intermediate Language (MSIL)code. The MSIL code is not specific to any operating system or to any language.



Benefits of CLR:

- It improves the performance by providing a richly interact between programs at the run time.
- Enhance portability by removing the need of recompiling a program on any operating system that supports it.·
- Security also increases as it analyzes the MSIL instructions whether they are safe or unsafe. Also, the use of delegates in place of function pointers enhance the type safety and security.
- Support automatic memory management with the help of Garbage Collector. · Provides cross-language integration because CTS inside CLR provides a common standard that activates the different languages to extend and share each other's libraries.
- Provides support to use the components that developed in other .NET programming languages. · Provide language, platform, and architecture independency.
- It allows the creation of the scalable and multithreaded applications in an easier way as a developer has no need to think about the memory management and security issues.

A C# program consists of the following parts −
· Namespace declaration
· A class
· Class methods
· Class attributes
· A Main method
· Statements and Expressions
· Comments

Let us look at a simple code that prints the words "Hello World" −

```
using System;
namespace HelloWorldApplication {
class HelloWorld {
static void Main(string[] args) {
/* my first program in C# */
Console.WriteLine("Hello World");
Console.ReadKey();
}
}
}
```

Let us look at the various parts of the given program −

The first line of the program using System; - the using keyword is used to include the System namespace in the program. A program generally has multiple using statements.

The next line has the namespace declaration. A namespace is a collection of classes. The Hello World Application namespace contains the class HelloWorld.

The next line has a class declaration, the class HelloWorld contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the HelloWorld class has only one method Main.

The next line defines the Main method, which is the entry point for all C# programs. The Main method states what the class does when executed.

The next line /*...*/ is ignored by the compiler and it is put to add comments in the program.

The Main method specifies its behavior with the statement Console.WriteLine("HelloWorld");

WriteLine is a method of the Console class defined in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

The last line Console.ReadKey(); is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

Note the following points −
C# is case sensitive.
All statements and expression must end with a semicolon (;).
The program execution starts at the Main method.
Unlike Java, program file name could be different from the class name.

Compiling and Executing the Program
If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps−

1. Start Visual Studio.
2. On the menu bar, choose File -> New -> Project.
3. Choose Visual C# from templates, and then choose Windows.
4. **Choose Console Application.**
5. Specify a name for your project and click OK button.
6. This creates a new project in Solution Explorer.
7. Write code in the Code Editor.
8. Click the Run button or press F5 key to execute the project.
9. A Command Prompt window appears that contains the line Hello World.

Accepting Values from User –
The Console class in the System namespace provides a function ReadLine()for accepting input from the user and store it into a variable.
For example,
int num;
num = Convert.ToInt32(Console.ReadLine());
Or
Num=int.Parse(Console.ReadLine());
The function Convert.ToInt32() converts the data entered by the user to int data type, because Console.ReadLine() accepts the data in string format.

C# Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.
Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring variable is given below:

```
int i, j;
double d;
```

Here, i, j, d, f, ch are variables and int, double, float, char are data types.
We can also provide values while declaring the variables as given below:

```
int i=2; //declaring 2 variable of integer type
float f=40.2;
char ch='B';
```

Rules for defining variables

1. A variable can have alphabets, digits and underscore.
2. A variable name can start with alphabet and underscore only. It can't start with digit.
3. No white space is allowed within variable name.
4. A variable name must not be any reserved word or keyword e.g. char, float

etc. Valid variable names:
int x; int _x; int k=20;

Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is − const <data_type> <constant_name> = value;
The following program demonstrates defining and using a constant in your program −

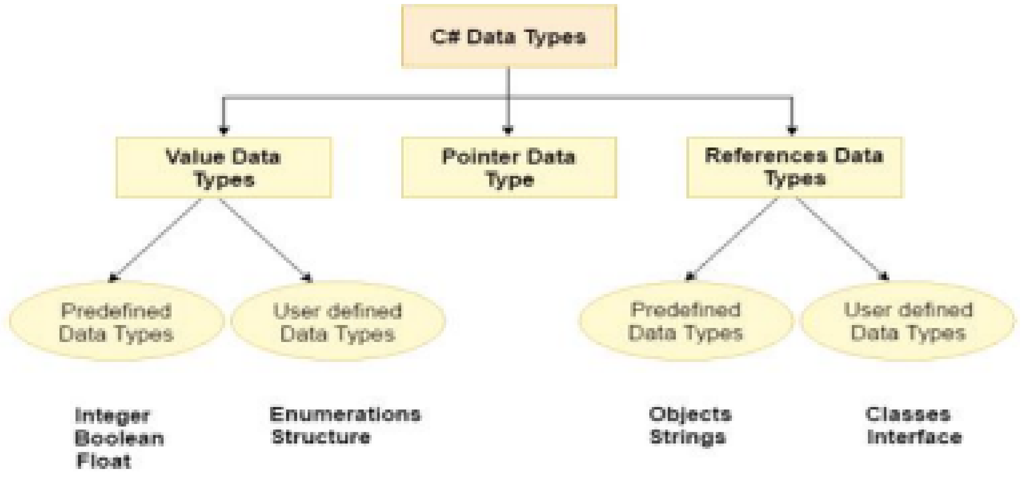
```
using System;
namespace DeclaringConstants {
class Program {
static void Main(string[] args) {
const double pi = 3.14159;

// constant declaration
double r;
Console.WriteLine("Enter Radius: ");
r = Convert.ToDouble(Console.ReadLine());

double areaCircle = pi * r * r;
Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
Console.ReadLine();
}
}
}
```

C# Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character
There are 3 types of data types in C# language.

Value data type	short, int, char, float, double etc

Reference Data Type	String, Class, Object and Interface
Pointer Data Type	Pointers



Value Data Type – A value type variable directly contains data in the memory.hey are derived from the class **System.ValueType**. The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals. To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine: using System;

```
namespace DataTypeApplication
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine("Size of int: {0}", sizeof(int));
Console.ReadLine();
}
}
}
```

When the above code is compiled and executed, it produces the following result:

```
Size of int: 4
```

There are 2 types of value data type in C# language.

1) Predefined Data Types - such as Integer, Boolean, Float, etc.

2) User defined Data Types - such as Structure, Enumerations, etc.

Reference Data Type

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables. A Reference type variable contains memory address of value. If the data is changed by one of the variables, the other variable automatically reflects this change in value.

There are 2 types of reference data type in C# language.

1) Predefined Types - such as Objects, String.

2) User defined Types - such as Classes, Interface.

Object Type
The **Object Type** is the ultimate base class for all data types in C# Common Type

System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user defined types.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

1. int Val = 1;
2: Object Obj = Val; //Boxing

The first line we created a Value Type Val and assigned a value to Val. The second line , we created an instance of Object Obj and assign the value of Val to Obj. From the above operation (Object Obj = i) we saw converting a value of a Value.

Type into a value of a corresponding Reference Type . These types of operation are called Boxing.

and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

1: int Val = 1;
2: Object Obj = Val; //Boxing
3: int i = (int)Obj; //Unboxing

The first two line shows how to Box a Value Type . The next line int i = (int) Obj shows extracts the ValueType from the Object . That is converting a value of a Reference Type into a value of a Value Type. This operation is called UnBoxing.

Type Casting and Conversion
Type conversion is converting one type of data to another type. It is also known as Type Casting.

· **Implicit Casting** (automatically) - converting a smaller type to a larger type size
char -> int -> long -> float -> double

· **Explicit Casting** (manually) - converting a larger type to a smaller size type
double -> float -> long -> int -> char

In C#, type casting has two forms −

Implicit type conversion − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.

Explicit type conversion − These conversions are done explicitly by users using the predefined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion −

```
using System;
namespace TypeConversionApplication {
     class ExplicitConversion {
          static void Main(string[] args) {
               double d = 5673.74;
               int i;

               // cast double to int.
```

```
                i = (int)d;
                Console.WriteLine(i);
                int p=10;
                long q=p;//implicit conversion
                Console.ReadKey();
            }
        }
  }
```

C# Function
Function is a block of code that has a signature. Function is used to execute statements specified in the code block. A function consists of the following components:
Function name: It is a unique name that is used to make Function call.
Return type: It is used to specify the data type of function return value.
Body: It is a block that contains executable statements.
Access specifier: It is used to specify function accessibility in the application.
Parameters: It is a list of arguments that we can pass to the function during call.

C# Function Syntax
```
<access-specifier><return-type>FunctionName(<parameters>)
{
// function body
// return statement
}
```
Access-specifier, parameters and return statement are optional.

Example->
```
Class test
{ public void show()
{ Console.WriteLine("welcome to c#");
}
Public static void Main(String[]args])
{ test t=new test();
t.show();
} }
```

C# Call By Value
In C#, value-type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value. In the following example, we have pass value during function call.

```
using System;
namespace CallByValue
{
class Program
{
public void Sum(int x)
{
x=x+10;
}
static void Main(string[] args)
{
```

```
int val = 50;
Program p = new Program(); // Creating Object
Console.WriteLine("Value before calling the function "+val);
p.Sum(val); // Calling Function by passing value
Console.WriteLine("Value after calling the function " + val);
} } }
```

C# Call By Reference
C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather
than copy of original value. The changes in passed values are permanent and **modify** the original variable value.

```
class Program
{ public void Sum(ref int x)
{
x=x+10;
}
static void Main(string[] args)
{
int val = 50;
Program p = new Program(); // Creating Object
Console.WriteLine("Value before calling the function "+val);
p.Sum ( ref val); // Calling Function by passing value
Console.WriteLine("Value after calling the function " + val);
} }
```

Method Overloading
Method Overloading is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name.
C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.

```
// C# program to demonstrate the function
// overloading by changing the Number
// of parameters
using System;
class GFG {

  // adding two integer values.
  public int Add(int a, int b)
  {
    int sum = a + b;
    return sum;
  }

  // adding three integer values.
  public int Add(int a, int b, int c)
  {
    int sum = a + b + c;
    return sum;
    }

    // Main Method
    public static void Main(String[] args)
    {

        // Creating Object
        GFG ob = new GFG();

        int sum1 = ob.Add(1, 2);
        Console.WriteLine("sum of the two "  + "integer value : " + sum1);

        int sum2 = ob.Add(1, 2, 3);
       Console.WriteLine("sum of the three "   + "integer value : " + sum2);
    }
}
```

C# | Constructor Overloading –
It is quite similar to the Method Overloading. It is the ability to redefine a Constructor in more than one form. A user can implement constructor overloading by defining two or more constructors in a class sharing the same name. C# can distinguish the constructors with different signatures. i.e. the constructor must have the same name but with different parameters list.
We can overload constructors in different ways as follows:

- By using different **type of arguments**
- By using different **number of arguments**
- By using different **order of arguments**

Example:
public ADD (int a, float b);
public ADD (string a, int b);
Here the name of the class is **ADD**. In first constructor there are two parameters, first one is **int** and another one is **float** and in second constructor, also there is two parameters, first one is **string** type and another one is **int** type.
Here the constructors have the same name but the types of the parameters are different, similar to the concept of method overloading.

```
// C# program to Demonstrate the overloading of
// constructor when the types of arguments
// are different
using System;

class ADD {

    int x, y;
    double f;
    string s;

    // 1st constructor
    public ADD(int a, double b)
    {
        x = a;
        f = b;
    }

    // 2nd constructor
    public ADD(int a, string b)
    {
        y = a;
        s = b;
    }

    // showing 1st constructor's result
    public void show()
    {
        Console.WriteLine("1st constructor (int + float): {0} ",
                                                (x + f));
    }

    // shows 2nd constructor's result
    public void show1()
    {
        Console.WriteLine("2nd constructor (int + string): {0}",
                                                (s + y));
    }
}

// Driver Class
class GFG {

    // Main Method
    static void Main()
    {

        // Creating instance and
        // passing arguments
        // It will call the first constructor
        ADD g = new ADD(10, 20.2);

        // calling the method
        g.show();


        // Creating instance and
        // passing arguments
        // It will call the second constructor
        ADD q = new ADD(10, "Roll No. is ");

        // calling the method
        q.show1();
    }
}
```

Output:

```
1st constructor (int + float): 30.2
2nd constructor (int + string): Roll No. is 10
```

By changing the number of the parameters
In this case, we will use two or more constructors having the different number of parameters. The data types of arguments can be the same but the number of parameters will be different.

Example:

```
public ADD (int a, int b);
public ADD (int a, int b, int c);
```

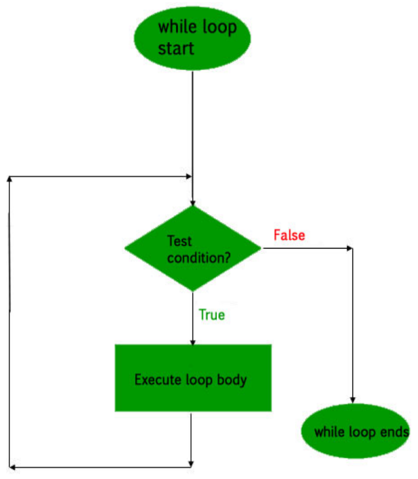
Here, the class name is **ADD**. In the first constructor the number of parameter is **two** and the types of the parameters is **int**. In second constructor the number of parameter is **three** and the types of the parameters are also **int**, their is no problem with the data types.

Loops –

Looping in a programming language is a way to execute a statement or a set of statements multiple times depending on the result of the condition to be evaluated to execute statements. The result condition should be true to execute statements within loops.
Loops are mainly divided into two categories:
Entry Controlled Loops: The loops in which condition to be tested is present in beginning of loop body are known as **Entry Controlled Loops**. **while loop** and **for loop** are entry controlled loops.
1. while loop The test condition is given in the beginning of the loop and all statements are executed till the given boolean condition satisfies when the condition becomes false, the control will be out from the while loop.
Syntax:

```
while (boolean condition)
{
  loop statements...
}
```

Flowchart:



Example:

```
// C# program to illustrate while loop
using System;
```

```
class whileLoopDemo
{
   public static void Main()
   {
      int x = 1;

      // Exit when x becomes greater than 4
      while (x <= 4)
      {
         Console.WriteLine("HelloWorld");

         // Increment the value of x for
         // next iteration
         x++;
      }
   }
}
```

Output:

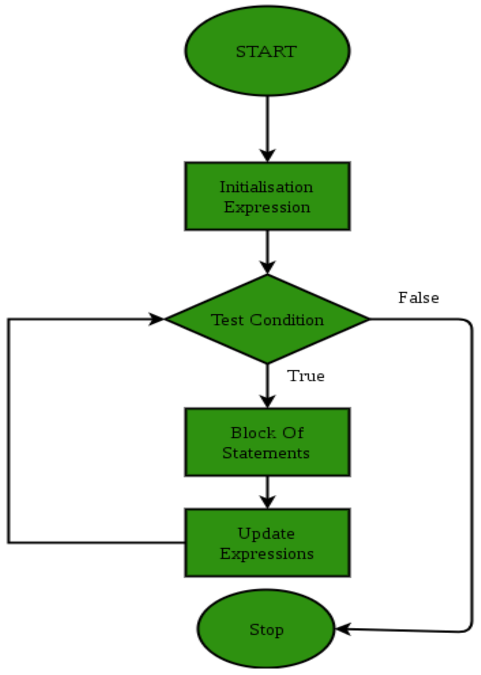
```
HelloWorld
HelloWorld
HelloWorld
HelloWorld
```

2. for loop

for loop has similar functionality as while loop but with different syntax. for loops are preferred when the number of times loop statements are to be executed is known beforehand. The loop variable initialization, condition to be tested, and increment/decrement of the loop variable is done in one line in for loop thereby providing a shorter, easy to debug structure of looping.

```
for (loop variable initialization ; testing condition;
                              increment / decrement)
{
   // statements to be executed
}
```

Flowchart:



1. Initialization of loop variable: Th expression / variable controlling the loop is initialized here. It is the starting point of for loop. An already declared variable can be used or a variable can be declared, local to loop only.
2. Testing Condition: The testing condition to execute statements of loop. It is used for testing the exit condition for a loop. It must return a boolean value **true or false**. When the condition became false the control will be out from the loop and for loop ends.
3. Increment / Decrement: The loop variable is incremented/decremented according to the requirement and the control then shifts to the testing condition again.
Note: Initialization part is evaluated only once when the for loop starts.
Example:

```
// C# program to illustrate for loop.
using System;

class forLoopDemo
{
   public static void Main()
   {
      // for loop begins when x=1
      // and runs till x <=4
      for (int x = 1; x <= 4; x++)
         Console.WriteLine("HelloWorld");
   }
}
```

Output:

```
HelloWorld
HelloWorld
HelloWorld
HelloWorld
```

Exit Controlled Loops: The loops in which the testing condition is present at the end of loop body are termed as **Exit Controlled Loops**. **do-while** is an exit controlled loop.

Note: In Exit Controlled Loops, loop body will be evaluated for at-least one time as the testing condition is present at the end of loop body.

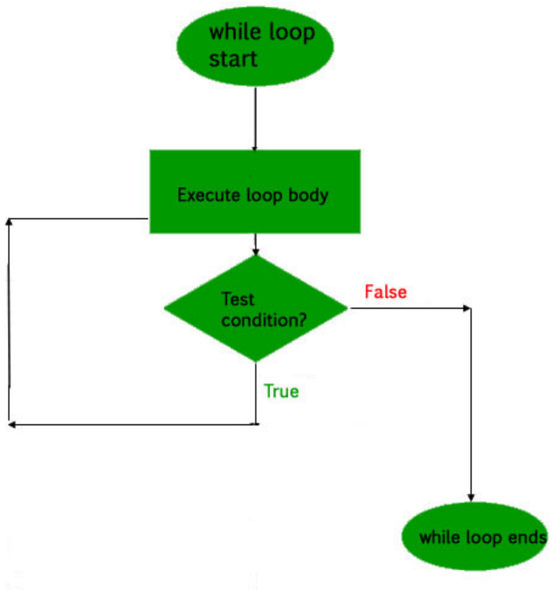
1. do-while loop

do while loop is similar to while loop with the only difference that it checks the condition after executing the statements, i.e it will execute the loop body one time for sure because it checks the condition after executing the statements.

Syntax :

```
do

{

   statements..

}while (condition);
```

Flowchart:



Example:

```
// C# program to illustrate do-while loop
using System;

class dowhileloopDemo
{
   public static void Main()
   {
      int x = 21;
      do
      {
         // The line will be printed even
         // if the condition is false
         Console.WriteLine("HelloWorld");
         x++;
```

```
        }
        while (x < 20);
    }
}
```

Output:

```
HelloWorld
```

Infinite Loops:
The loops in which the test condition does not evaluate false ever tend to execute statements forever until an external force is used to end it and thus they are known as infinite loops.
Example:

```
// C# program to demonstrate infinite loop
using System;
class infiniteLoop
{
    public static void Main()
    {
        // The statement will be printed
        // infinite times
        for(;;)
        Console.WriteLine("This is printed infinite times");
    }
}
```

Output:

```
This is printed infinite times
This is printed infinite times
This is printed infinite times
This is printed infinite times
This is printed infinite times
This is printed infinite times
This is printed infinite times
..........
```

Nested Loops:
When loops are present inside the other loops, it is known as nested loops.
Example:

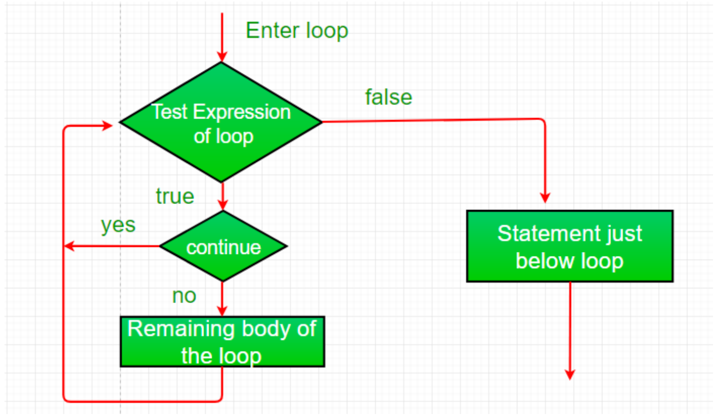
```
// C# program to demonstrate nested loops
using System;

class nestedLoops
{
    public static void Main()
    {
        // loop within loop printing HelloWorld
       for(int i = 2; i < 3; i++)
             for(int j = 1; j < i; j++)
                 Console.WriteLine("HelloWorld");
    }
}
```

Output:

```
HelloWorld
```

continue statement:
continue statement is used to skip over the execution part of loop on a certain condition and move the flow to next updation part.

Flowchart:



Example:

```
// C# program to demonstrate continue
statement
using System;
class demoContinue
{
    public static void Main()
    {
        // HelloWorld is printed only 2 times
        // because of continue statement
        for(int i = 1; i < 3; i++)
        {
            if(i == 2)
              continue;
            Console.WriteLine("HelloWorld");
        }
    }
}
```

Output:

```
HelloWorld
```

Difference between while and do-while

Basis for Comparison	While	Do-while

General Form	while (condition)
	{
	statements; //body of loop
	}
		do{
		.statements; // body of loop.
		} while(Condition);.

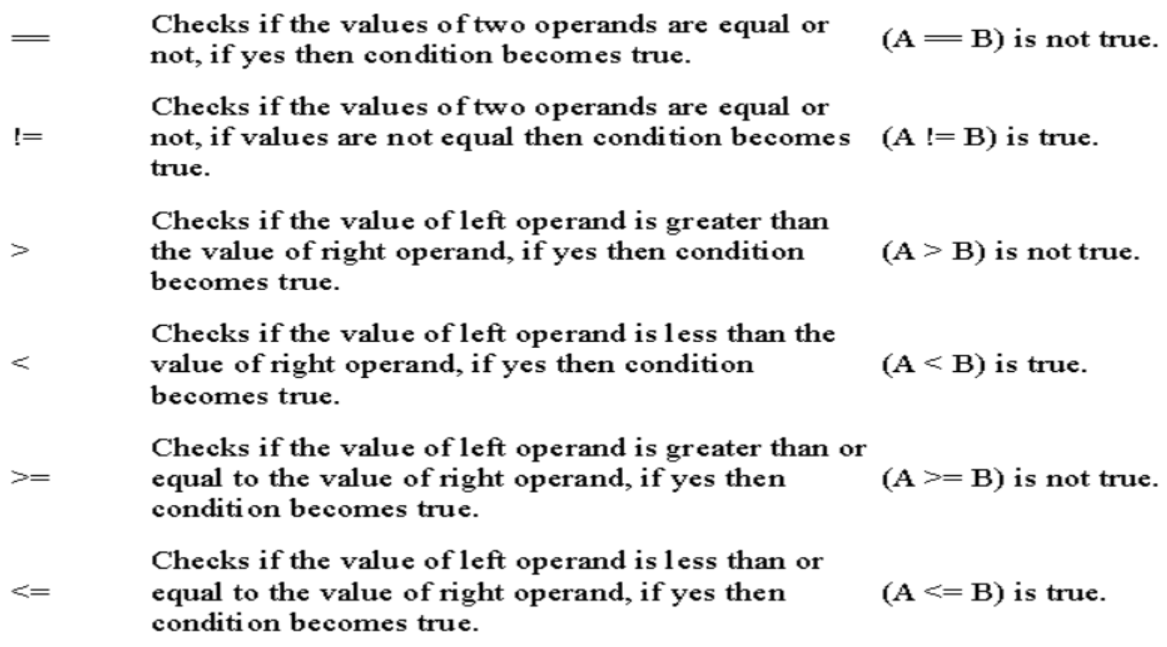
Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.
	It is called entry loop	It is called exit loop
Example		

Operators
An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators −
Arithmetic Operators
Relational Operators
Logical Operators
Bitwise Operators
Assignment Operators
Misc Operators

Arithmetic Operators
Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B**
holds 20 then –

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

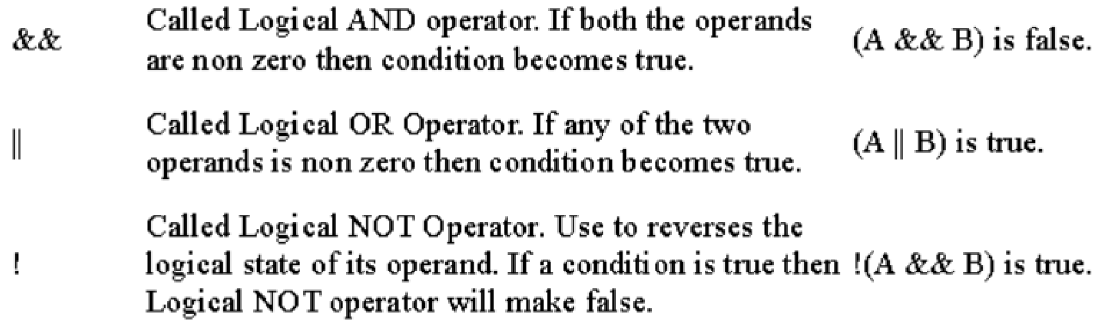
Relational Operators
Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then –



==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then −



&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows –

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; then in the binary format they are as follows −

```
A = 0011 1100
B = 0000 1101
-------------------
A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A  = 1100 0011
```

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, which is 0000 1111

Assignment Operators

There are following assignment operators supported by C# −

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators
There are few other important operators including **sizeof, typeof** and **? :** supported by C#.

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; returns actual address of the variable.
*	Pointer to a variable.	*a; creates pointer named 'a' to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y