MODULE-1

ASP.NET

ASP.NET is a web application framework designed and developed by Microsoft. ASP.NET is open source and a subset of the .NET Framework(is a software) and successor of the classic ASP (**A**ctive **S**erver **P**ages). The ASP.NET application codes can be written in any of the following languages:

- C#
- Visual Basic.Net
- Jscript
- J#

WHAT IS .NET?

NET stands for Network Enabled Technology. In .NET, dot (.) refers to object-oriented and NET refers to the internet. So, the complete .NET means through object-oriented we can implement internet-based applications.

.NET FRAMEWORK

A framework is a software. Or you can say a framework is a collection of many small technologies integrated together to develop applications that can be executed anywhere.

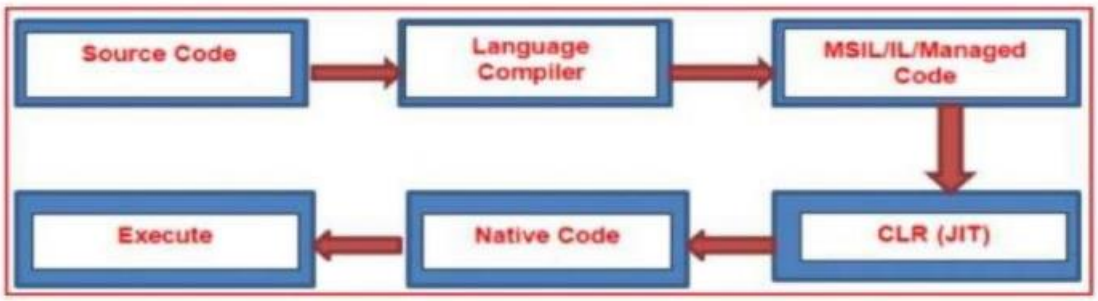
The DOTNET Framework provides two things are as follows

1) **BCL (Base Class Libraries)**

   Base Class Libraries (BCL) is designed by Microsoft. Without BCL we can't write any code in .NET. So, BCL is also known as the building block of .NET Programs. These are installed into the machine when we installed the .NET framework. BCL contains pre-defined classes and these classes are used for the purpose of application development.

2) **CLR (Common Language Runtime)**

   CLR stands for Common Language Runtime and it is the core component under the .NET framework which is responsible for converting the MSIL (Microsoft Intermediate Language) code into native code.



In the .NET framework, the code is compiled twice.

1) In the 1st compilation, the source code is compiled by the respective language compiler and generates the intermediate code which is known as MSIL (Microsoft Intermediate Language) or IL (Intermediate language code) Or Managed Code.
2) In the 2nd compilation, MSIL is converted into Native code (native code means code specific to the Operating system so that the code is executed by the Operating System) and this is done by CLR.
3) Always 1st compilation is slow and 2nd compilation is fast.

TYPES OF .NET FRAMEWORK

The .net framework is available in three different flavours

1) **DOTNET Framework:** This is the general version required to run .NET applications on Windows OS only.
2) **.NET mono Framework:** This is required if we want to run DOT NET applications on other OS like Unix, Linux, MAC OS, etc.
3) **DOT NET Compact Framework:** This is required to run .NET applications on other devices like mobile phones and smartphones.

COMPONENTS OF CLR:

1) **JIT COMPILER:**

   JIT stands for the Just-in-Time compiler. It is the component of CLR that is responsible for converting MSIL code into Native Code. Native code is the code that is directly understandable by the operating system.
2) **MEMORY MANAGER:**

   Allocates memory for variables and objects and it is Used by the applications.

3) GARBAGE COLLECTOR:

Garbage Collector in .NET Framework is nothing but is a Small Routine or you can say it's a Background Process Thread that runs periodically and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

Generations of Garbage Collector:

They are Generation 0, Generation 1, and Generation 2.

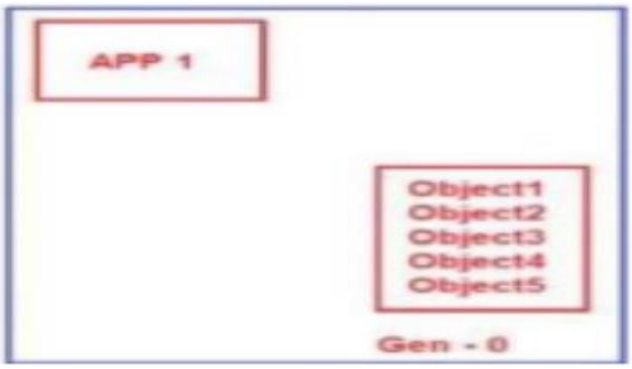
Understanding Generation 0, 1, and 2:

• Let say you have a simple application called App1.

• As soon as the application started it creates 5 managed objects.

• Whenever any new objects (fresh objects) are created, they are moved into a bucket called Generation 0.

For better understanding please have a look at the following image.



We know our hero Mr. Garbage Collector runs continuously as a background process thread to check whether there are any unused managed objects so that it reclaims the memory by cleaning those objects.

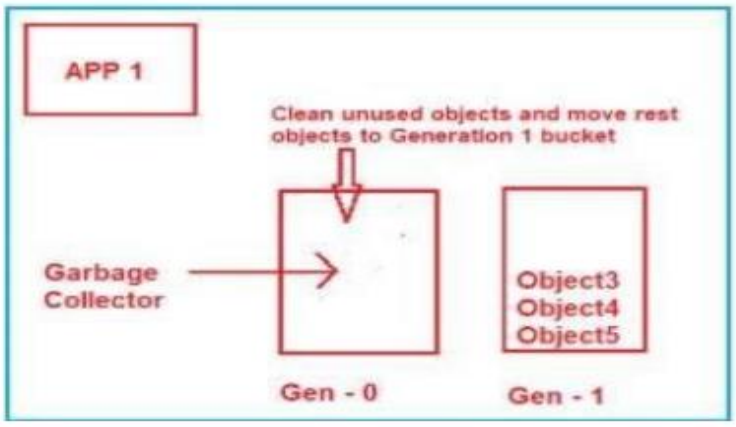
• Now, let say two objects (Object1 and Object2) are not needed by the application.

• So, Garbage Collector will destroy these two objects (Object1 and Object2) and reclaims the memory from Generation 0 bucket.

• But the remaining three objects (Object3, Object4, and Object5) are still needed by the application.

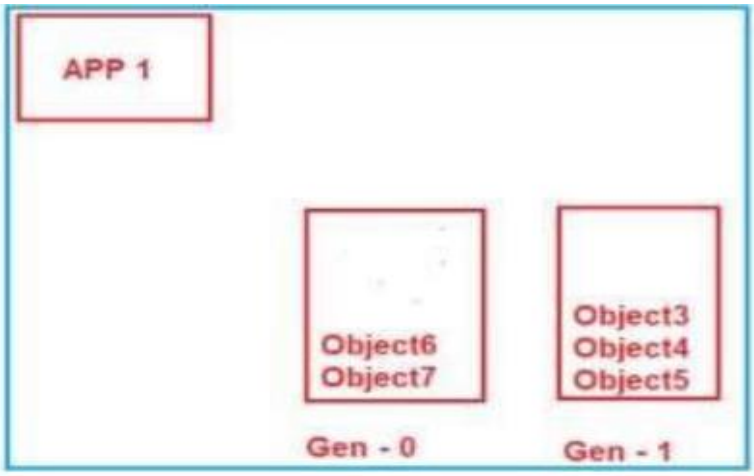
• So, the Garbage collector will not clean those three objects.

What Garbage Collector will do is, he will move those three managed objects (Object3, Object4, and Object5) to Generation 1 bucket as shown in the below image.



Now, let say your application creates two more fresh objects (Object6 and Object7).

As fresh objects, they should be created in Generation 0 bucket as shown in the below image.

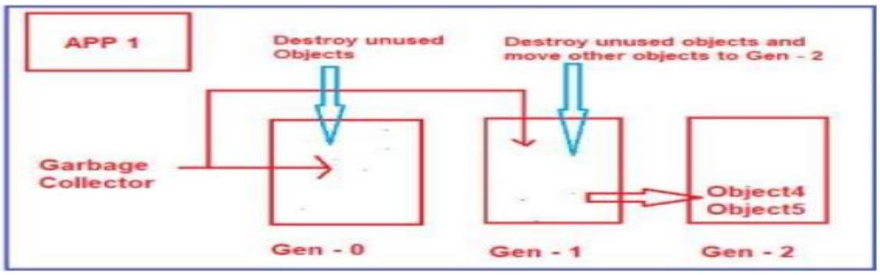


Now, again Garbage Collector runs and it comes to **Generation 0 bucket** and checks which objects are used.

Let say both objects (Object6 and Object7) are unused by the application, so it will remove both the objects and reclaims the memory.

Now, it goes to the **Generation 1 bucket,** and checks which object are unused.

Let say Object4 and Object5 are still needed by the application while object3 is not needed. So, what Garbage Collector will do is, it will destroy Object3 and reclaims the memory as well as it will move Objec4 and Object5 to Generation 3 bucket which is shown in the below image.



4) **Exception Manager:**

The Exception Manager component of CLR in the .NET Framework redirects the control to execute the catch or finally blocks whenever an exception has occurred at runtime.

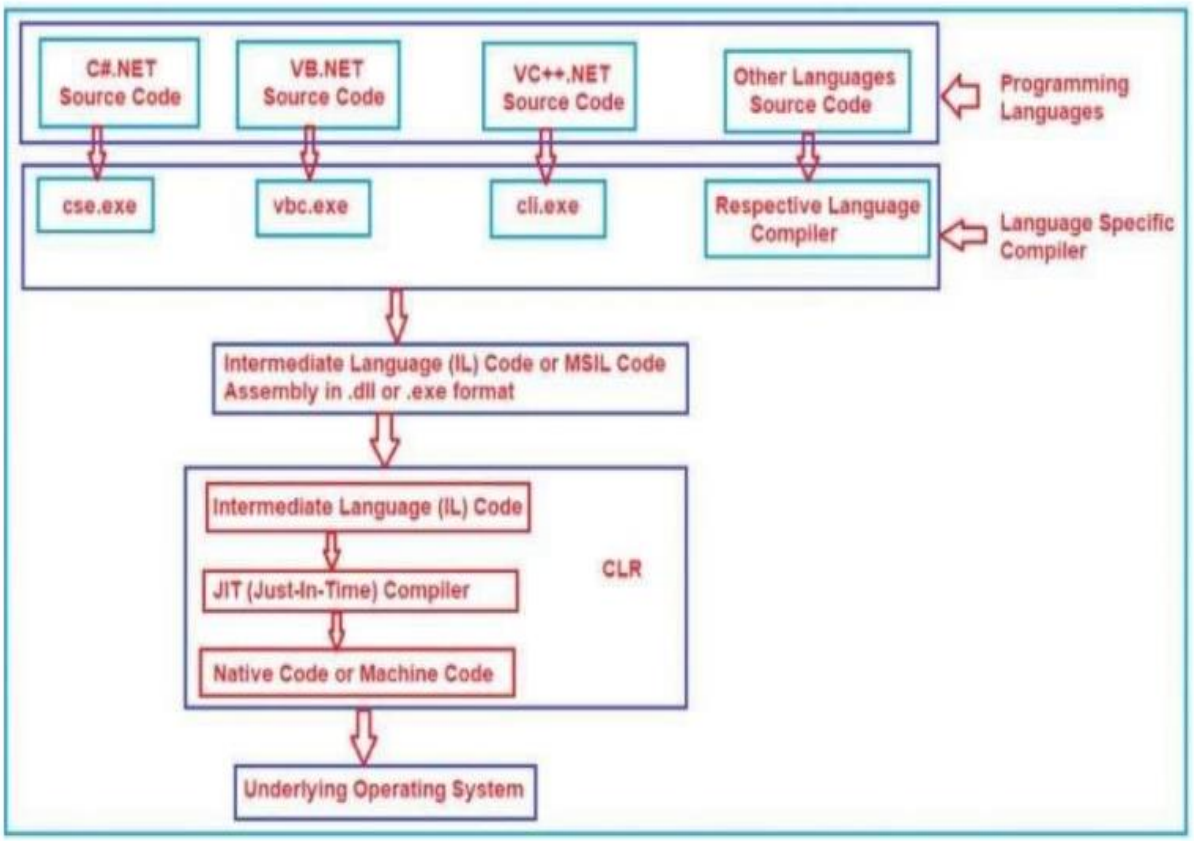
5) **CLS (Common Language Specification)**

It is a part of CLR.The .NET supports many programming languages such as C#, VB.Net, J#, etc.,. Every programming language has its own rules for writing the code that is known as common language specification. One programming language cannot be understood by the other programming language. In order to ensure smooth communication between different programming languages the most important thing is that they should have a common language specification. It is a subset of common type S/M. It defines set of rules and restrictions that every language must follow which runs under net framework.

6) **CTS (Common Type System)**

Common Type System (CTS) describes the data types that can be used by managed code. CTS defines how these types are declared, used and managed in the runtime. It facilitates cross-language integration, type safety, and high-performance code execution. The rules defined in CTS can be used to define your own classes and values.

.NET PROGRAM EXECUTION PROCESS



In .NET, the application execution consists of 2 steps. They are as follows:

1) In step1 the respective language compiler compiles the Source Code into Intermediate Language (IL).
2) In the 2nd step, the JIT compiler of CLR will convert the Intermediate Language (IL) code into native code (Machine Code or Binary Code) which can then be executed by the underlying operating system.

As the .NET assembly is in Intermediate Language (IL) format and not in native code or machine code, the .NET assemblies are portable to any platform as long as the target platform has the Common Language Runtime (CLR). The target platform's CLR converts the Intermediate Language code into native code or machine code that the underlying operating system can understand.

MODULE-2

C#:

C# (C-Sharp) is a programming language developed by Microsoft that runs on the .NET Framework. C# is used to develop web apps, desktop apps, mobile apps, games and much more using System. C# was developed by Anders Hejlsberg and his team during the development of .NET Framework. C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment.

Features of C#:

1) **Simple**

   C# is a user-friendly language that offers a structured approach to problem-solving. it provides a wide range of library functions and data types to work.

2) **Modern Programming Language**

   C# programming is a popular and powerful language that is for creating scalable, interoperable, and robust applications.

3) **Object Oriented**

   C# is an object-oriented programming language, which makes development and maintenance easier. In contrast, with procedure-oriented programming languages, managing code becomes difficult as project size grows.

4) **Type Safe**

   The code is type safe can only access memory locations that it has permission to execute. This feature significantly enhances program security.

5) **Interoperability**

   The interoperability process allows C# programs to perform all the tasks that a native C++ application.

6) **Scalable and Updateable**

   C# is a programming language that is scalable and can be updated automatically. To update our application, we remove the old files and replace them with new ones.

7) **Component Oriented**

It is widely used as a software development methodology to create applications that are more strong and can easily scale.

8) **Structured Programming Language**

C# is a structured programming language that allows us to divide programs into parts using functions, making it easy to understand and modify.

9) **Fast Speed**

The compilation and execution time of C# language is fast.

C# programming basically consist of following parts:

1) Namespace:

A namespace is a collection of classes and using is keyword to import namespace (Using is just like #include in C-Language), there are various inbuilt namespaces available in C#.net.

Syntax:

```
using System; // root namespace
namespace group1 // user-defined namespace
{
        {
                public class bank
                {
                        // function and variables
                }
        }
}
```

2) Class:

A class as a blueprint of a specific object.

Syntax:

```
Class classname
{
        //statement
}
```

3) Methods:

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main. To use a method, you need to:

a) Define the method

b) Call the method

4) Attributes:

Class attributes are variables defined in the class. These attributes define the state of the object at a particular time. Attributes can be either primitive type like int, byte, char, long, double etc.

4) Main method:

The Main() method is an entry point of console and windows applications on the .NET or .NET Core platform. It is also an entry of ASP.NET Core web applications. When you run an application, it starts the execution from the Main() method. So, a program can have only one Main() method as an entry point. However, a class can have multiple Main() methods, but any one of them can be an entry point of an application.

5) Statements and expressions:

A statement is a basic unit of execution of a program. A program consists of multiple statements. An expression in C# is a combination of operands (variables, literals, method calls) and operators that can be evaluated to a single value.

6) Comments:

Comments are used in a program to help us understand a piece of code. They are human readable words intended to make the code readable. Comments are completely ignored by the compiler.

In C#, there are 2 types of comments:

1. Single Line Comments (//)
2. Multi Line Comments (/* */)

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

1. A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore.
2. The first character in an identifier cannot be a digit.
3. It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore (_) can be used.
4. It should not be a C# keyword.

Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers; however, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

auto	double	int	typedef
break	else	long	union
case	enum	return	unsigned
do	if	Struct	switch

Data types:

In C#, variables are categorized into the following types:

1. Value types
2. Reference types
3. Pointer types

Value Types:

Value type variables can be assigned a value directly. They are derived from the class System.ValueType. The value types directly contain data. Some examples are int, char, float, which stores numbers, alphabets, floating point numbers, respectively.When you declare an int type, the system allocates memory to store the value.

Example:

```
static void changeValue( int X)
{
    X=200;
    Console.WriteLine(X);
}
static void main(string[] args)
{
    int i=100;
    console.WriteLine(i);
    changeValue(i);
    console. WriteLine(i);
}
```

Reference Type :

A reference type doesn't store its value directly. Instead, it stores the address where the value is being stored.

Example:

```
Static void changeReferenceType(Student std2)
{
    Std2.StudentName=”Steve”;
}
static void main(String[] args)
{
    Student std1=new student;
    Std1.StudentName=”Bill”;
```

```
    changeReferenceType(std1);
    console. WriteLine (std1.StudentName);
}
```

Pointer Types:

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as in C or C++.

Syntax for declaring a pointer type is:

type* identifier;

For example,

```
char* cptr;
```

Type Conversion:

Type conversion is basically type casting, or converting one type of data to another type.

In C#, type casting has two forms:

1) Implicit type conversion:

   These conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types and conversions from derived classes to base classes.

2) Explicit type conversion:

   These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

Variables:

A variable is nothing but a name given to a storage area that our programs can manipulate.

Syntax for variable definition in C# is:

<data_type> <variable_list>;

Example:

```
int i, j, k;

char c, ch;
```

Constants and Literals:

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.

Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# is rich in built-in operators and provides the following type of operators:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators

1) Arithmetic operators:

The Arithmetic Operators in C# are used to perform arithmetic/mathematical operations like addition, subtraction, multiplication, division, etc. on operands. The following Operators are falling into this category.

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

Example:

3)Relational operators:

The Relational Operators in C# are also known as Comparison Operators. It determines the relationship between two operands and returns the Boolean results, i.e. true or false after the comparison. The Different Types of Relational Operators supported by C# are as follows.

Operator	Description	Example
==	Checks if the value s 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	(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 g re ate r 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.

4)Logical operators:

The Logical Operators are mainly used in conditional statements and loops for evaluating a condition. These operators are going to work with boolean expressions. The different types of Logical Operators supported in C# are as follows:

Operator	Description	Example
&&	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.

5)Bitwise operators:

The Bitwise Operators in C# perform bit-by-bit processing. They can be used with any of the integer (short, int, long, ushort, uint, ulong, byte) types. The different types of Bitwise Operators supported in C# are as follows.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form due to a sig ne d 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 will give 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 will give 15, which is 0000 1111

6)Assignment operators:

Assignment operators are used to assign values to variables.

Symbol	Operation	Example
+=	Plus Equal To	x+=15 is x=x+15
-=	Minus Equal To	x- =15 is x=x-15
*=	Multiply Equal To	x*=16 is x=x+16
%=	Modulus Equal To	x%=15 is x=x%15
/=	Divide Equal To	x/=16 is x=x/16

Decision making:

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed. If the condition is true then it executes the following statements. If the condition is false the control will comes out of the loop. C# provides following types of decision-making statements.

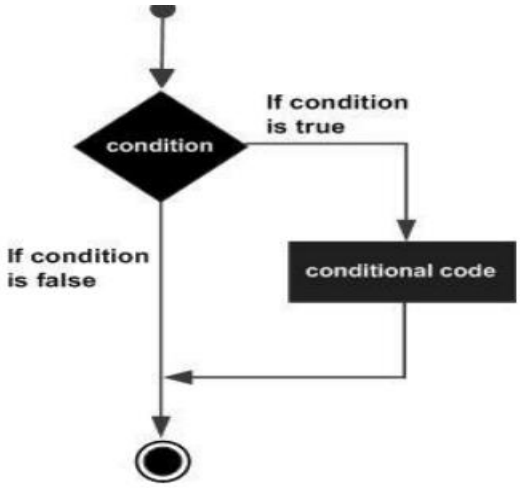
1) if statement
2) if…. else statement
3) nested if statement
4) switch statement

if statement:

An if statement consists of a boolean expression followed by one or more statements. The syntax of an if statement in C# is:

```
if(boolean_expression)
{
        /* statement(s) will execute if the boolean expression is true */
}
```

If the boolean expression evaluates to true, then the block of code inside the if statement will be executed. If boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.



Example:

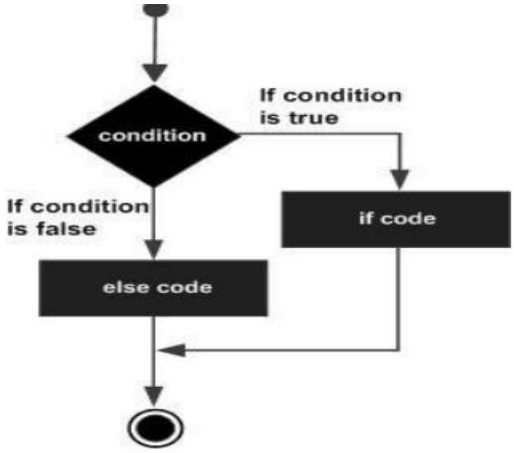
```
using System;
namespace DecisionMaking
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10;
            if (a < 20)
            {
                Console.WriteLine("a is less than 20");
            }
            Console.WriteLine("value of a is : ", a);
        }
    }
}
```

if…. else statement

An if statement can be followed by an optional else statement, which executes when the boolean expression is false. The syntax of an if...else statement in C# is:

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.



Example:

```
using System;
namespace DecisionMaking
{
 class Program
 {
        static void Main(string[] args)
        {
                int a = 100;
                if (a < 20)
                {
                        Console.WriteLine("a is less than 20");
                }
                else
                {
                        Console.WriteLine("a is not less than 20");
                }
                 Console.WriteLine("value of a is :", a);
        }
 }
}
```

Nested if statement

It is always legal in C# to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s). The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
    if(boolean_expression 2)
    {
        /* Executes when the boolean expression 2 is true */
    }
}
```

Example:

```
using System;
namespace DecisionMaking
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 100;
            int b = 200;
            if (a == 100)
            {
                if (b == 200)
                {
                    Console.WriteLine("Value of a is 100 and b is 200");
```

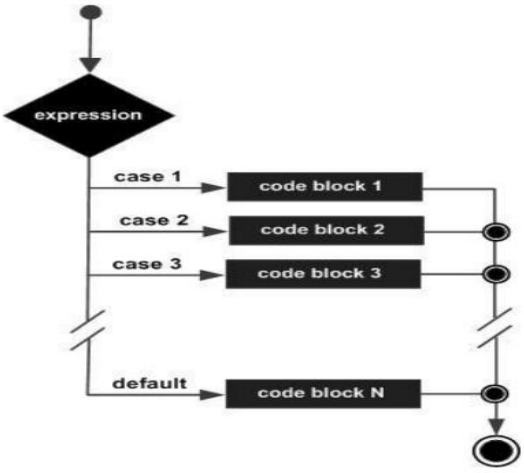
```
                }
            }
            Console.WriteLine("Exact value of a is:", a);
            Console.WriteLine("Exact value of b is:", b);
        }
    }
}
```

Switch statement:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case. The syntax for a switch statement in C# is as follows:

```
switch(expression)
{
    case constant-expression: statement(s);
                              break;
    case constant-expression: statement(s);
                              break;
    default: statement(s);
}
```

Switch statement consist of n number of cases. It checks the expression with each case if the expression matches with any no of cases, it will print the statement belongs to that case. If the expression doesn't match with any number of cases, it will print default statements.



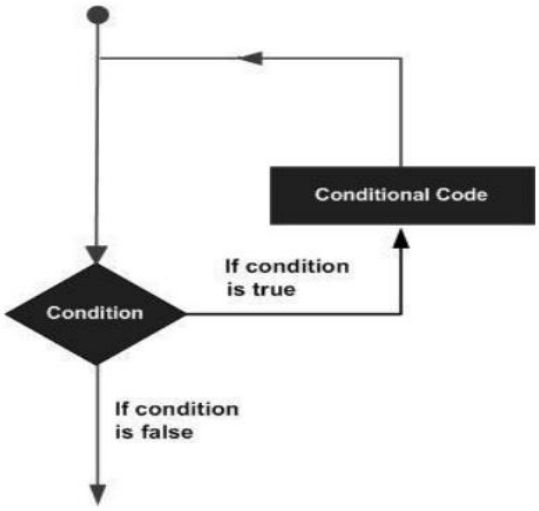
Example:

```
using System;
namespace DecisionMaking
{
        class Program
        {
                static void Main(string[] args)
                {
                        char grade = 'B';
                        switch (grade)
                        {
                                case 'A': Console.WriteLine("Excellent!");
                                                break;
                                case 'B':
                                case 'C': Console.WriteLine("Well done");
                                                break;
                                 case 'D':  Console.WriteLine("You passed");
                                                break;
                                case 'F': Console.WriteLine("Better try again");
                                                break;
                                 default:  Console.WriteLine("Invalid grade");
                                                 break;
```

```
                }
                Console.WriteLine("Your grade is {0}", grade);
            }
        }
}
```

Loops

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C# provides following types of loops to handle looping requirements.

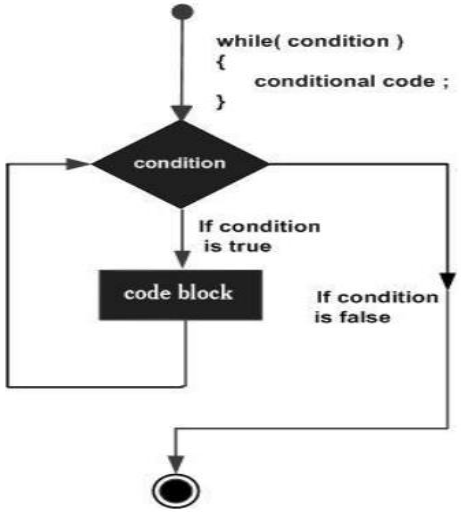
1) While loop
2) Do while loop
3) For loop
4) Nested loops

While loop:

A while loop statement in C# repeatedly executes a target statement as long as a given condition is true. The syntax of a while loop in C# is:

```
while(condition)
{
    statement(s);
}
```

While statement is also called as pre-tested loop. If the condition is true then it executes the statement. If the condition is false the control will come out of the loop.



Example:

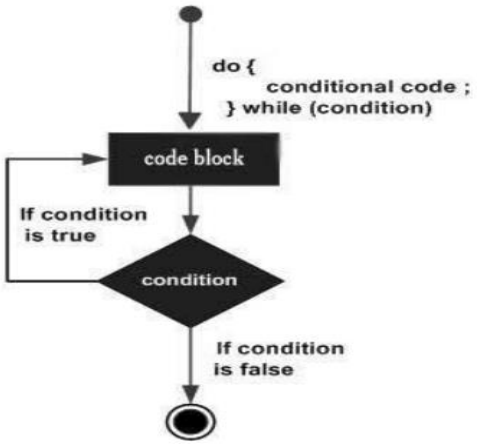
```
using System;
namespace Loops
{
        class Program
        {
                static void Main(string[] args)
                {
                        int a = 10;
                        while (a < 20)
                        {
                                Console. WriteLine("value of a: {0}", a);
                                 a++;

                         }
                }
        }
}
```

Do while loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time. The syntax of a do...while loop in C# is:

```
do
{
        statement(s);
} while (condition);
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.



Example:

```
using System;
namespace Loops
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10;
            do
```

```
            {
                Console. WriteLine("value of a: {0}", a);
                a = a + 1;
                } while (a < 20);
            }
        }
}
```

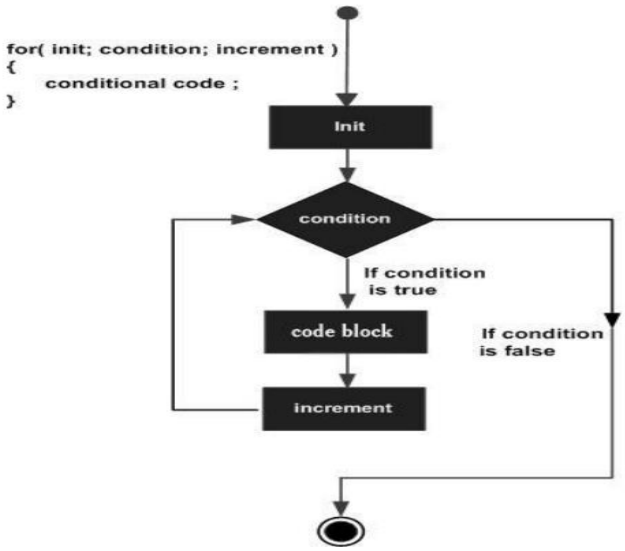
For loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The syntax of a for loop in C# is:

```
for ( init; condition; increment )
{
    statement(s);
}
```

Here is the flow of control in a for loop:

1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.



Example:

```
using System;
namespace Loops
{
        class Program
        {
                static void Main(string[] args)
                {
                        for (int a = 10; a < 20; a = a + 1)
                        {
                                Console. WriteLine("value of a: {0}", a);
                        }
                }
        }
}
```

Access specifiers:

An access specifier defines the scope and visibility of a class member. C# supports the following access specifiers:

1) **Public Access Specifier**

   Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

Example:

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        public double length;
        public double width;
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
```

```
            r.width = 3.5;
            r.Display();
        }
    }
}
```

2) Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

Example

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        private double length;
        private double width;
        public void Acceptdetails()
        {
            Console.WriteLine("Enter Length: ");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter Width: ");
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: ", length);
```

```
                Console.WriteLine("Width: ", width);
                Console.WriteLine("Area: ", GetArea());
            }
        }
        class ExecuteRectangle
        {
            static void Main(string[] args)
            {
                Rectangle r = new Rectangle();
                 r.Acceptdetails();
                r.Display();
                Console.ReadLine();
            }
        }
    }
```

3) **Protected Access Specifier**

Protected access specifier allows a child class to access the member variables and member functions of its base class.

Methods:

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main. To use a method, you need to:

a) Define the method

b) Call the method

Defining Methods in C#:

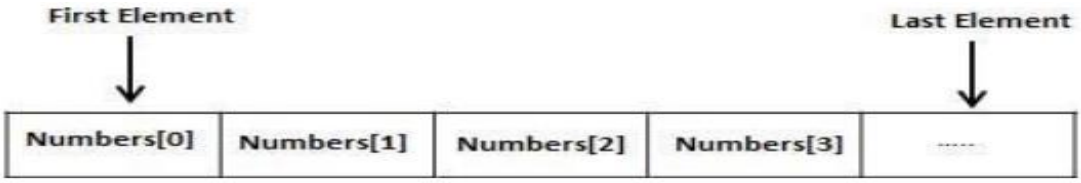
```
<Access specifier><Return Type><Method Name>(Parameter list)
{
    Method body
}
```

Calling Methods in C#:

```
namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            int result;
            if (num1 > num2)
                result = num1;
            else result = num2;
                return result;
        }
        static void Main(string[] args)
        {
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();
            ret = n.FindMax(a, b);
            Console.WriteLine("Max value is :", ret );
        }
    }
}
```

Arrays:

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays:

datatype [] arrayName;

where,

datatype is used to specify the type of elements to be stored in the array.

[ ] specifies the rank of the array. The rank specifies the size of the array.

arrayName specifies the name of the array.

For example,

double [] balance;

Initializing an Array :

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array.

For example,

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

Types of Arrays:

1. One-dimensional array
2. Multi-dimensional array
3. Jagged array

1) One-dimensional array:

One dimensional array is like a sequence of an elements. You access an element via its index 0. You create a single dimensional array using a new operator specifying array element types and no of elements.

Example:

int[] array = new int[5];

string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
Console.WriteLine(weekDays[0]);

Console.WriteLine(weekDays[1]);

Console.WriteLine(weekDays[2]);

Console.WriteLine(weekDays[3]);

Console.WriteLine(weekDays[4]);

2) two-dimensional array :

A two-dimensional array is an array in which each element is referred to by two indexes. It is stored in the matrix form, where the first index represents the row of the matrix and the second index represents the column of the matrix.

Syntax:

int[,] array_name = new int[row count][column count];

Example:

int [,] a = int [3,4] = { {0, 1, 2, 3} , /* initializers for row indexed by 0 */

{4, 5, 6, 7} , /* initializers for row indexed by 1 */

{8, 9, 10, 11} /* initializers for row indexed by 2 */ };

3) Jagged Arrays:

Jagged array is a array of arrays such that member arrays can be of different sizes. In other words, the length of each array index can differ. The elements of Jagged Array are reference types and initialized to null by default. Jagged Array can also be mixed with multidimensional arrays. Here, the number of rows will be fixed at the declaration time, but you can vary the number of columns.

Syntax:

```
data_type[][] name_of_array = new data_type[rows][]
```

Strings:

In C#, you can use strings as array of characters, however, more common practice is to use the string keyword to declare a string variable. The string keyword is an alias for the System. String class.

Creating a String Object:

1) By assigning a string literal to a String variable

2) By using a String class constructor

3) By using the string concatenation operator (+)

4) By retrieving a property or calling a method that returns a string

5) By calling a formatting method to convert a value or object to its string representation

Example:

```
using System;

namespace StringApplication

{

        class Program

        {

                static void Main(string[] args)

                {

                        string fname, lname;

                        fname = "Nagaraj";

                        lname = "H";

                        string fullname = fname + lname;

                        Console.WriteLine("Full Name: {0}", fullname);

                        char[] letters = { 'H', 'e', 'l', 'l','o' };

                        string greetings = new string(letters);
```

```
                Console.WriteLine("Greetings: {0}", greetings);
                string[] sarray = { "Hello", "From", "Karnatak", "University" };
                string message = String.Join(" ", sarray);
                Console.WriteLine("Message:", message);
                DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
                string chat = String.Format("Message sent at {0:t} on {0:D}",waiting);
                Console.WriteLine("Message: {0}", chat);
            }
        }
}
```

Output:

Full Name: Nagaraj H

Greetings: Hello

Message: Hello From Karnataka University

Message: Message sent at 5:58 PM on Wednesday, October 10, 2012

Properties of the String Class:

The String class has the following two properties:

1) Chars – gets the char object at a specified position in the current string object.
2) Length – gets the no of characters in the current string object.

Methods of the String Class:

1) public static int Compare(string strA, string strB)
2) public static string Concat(string str0, string str1)
3) public static bool IsNullOrEmpty(string value)
4) public string ToLower()

5) public string ToUpper()

C# structure

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The struct keyword is used for creating a structure.

Defining a Structure:

- To define a structure, you must use the struct statement.
- The struct statement defines a new data type, with more than one member for your program.
- For example, here is the way you would declare the Book structure:

```
struct Books
{
        public string title;
        public string author;
        public string subject;
        public int book_id;
};
```

Class vs Structure :

- Classes and Structures have the following basic differences:
- classes are reference types and structs are value types
- structures do not support inheritance
- structures cannot have default constructor

Enums:

An enumeration is a set of named integer constants. An enumerated type is declared using the enum keyword.

Declaring enum Variable:

The general syntax for declaring an enumeration is:

```
enum <enum _ name>
{
        enumeration list
};
```

Example:

```
using System;
namespace EnumApplication
{
        class EnumProgram
         {
                enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
                static void Main(string[] args)
                {
                        int WeekdayStart = (int)Days.Mon;
                        int WeekdayEnd = (int)Days.Fri;
                        Console.WriteLine("Monday: ", WeekdayStart);
                        Console.WriteLine("Friday: ", WeekdayEnd);
                }
        }
```

Classes:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

Class Definition

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces.

Following is the general form of a class definition:

```
<access specifier> class  class_name
{
   // member variables
   <access specifier> <data type> variable1;
   <access specifier> <data type> variable2;
   ...
   <access specifier> <data type> variableN;
   // member methods
   <access specifier> <return type> method1(parameter_list)
   {
      // method body
   }
   <access specifier> <return type> method2(parameter_list)
   {
      // method body
   }
   ...
   <access specifier> <return type> methodN(parameter_list)
   {
      // method body
   }
}
```

Constructors in C#

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. A constructor will have exact same name as the class and it does not have any return type.

Example:

```
namespace LineApplication

{

        class Line

        {

                private double length; // Length of a line

                public Line()

                {

                        Console.WriteLine("Object is being created");

                }
```

```
        public void setLength( double len )
        {
            length = len;
        }
        public double getLength()
        {
            return length;
        }
        static void Main(string[] args)
        {
            Line line = new Line();
            line.setLength(6.0);
            Console.WriteLine("Length of line :", line.getLength());
        }
    }
}
```

Default and Parameterized constructor:

A default constructor does not have any parameter but if you need a constructor can have parameters. Such constructors are called parameterized constructors.

Example

```
using System;
namespace LineApplication
{
    class Line
    {
```

```
        private double length; // Length of a line
        public Line(double len) //Parameterized constructor
        {
            Console.WriteLine("Object is being created, length = {0}", len);
            length = len;
        }
        public void setLength( double len )
        {
            length = len;
        }
        public double getLength()
        {
            return length;
        }
        static void Main(string[] args)
        {
            Line line = new Line(10.0);
            Console.WriteLine("Length of line : {0}", line.getLength());
            line.setLength(6.0);
            Console.WriteLine("Length of line : {0}", line.getLength());
        }
    }
}
```

Destructors in C#

A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope. A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.

Example:

```
namespace LineApplication
{
    class Line
    {
        private double length;
        public Line()
        {
            Console.WriteLine("Object is being created");
        }
        ~Line() //destructor
        {
            Console.WriteLine("Object is being deleted");
        }
        public void setLength( double len )
        {
            length = len;
        }
        public double getLength()
        {
            return length;
```

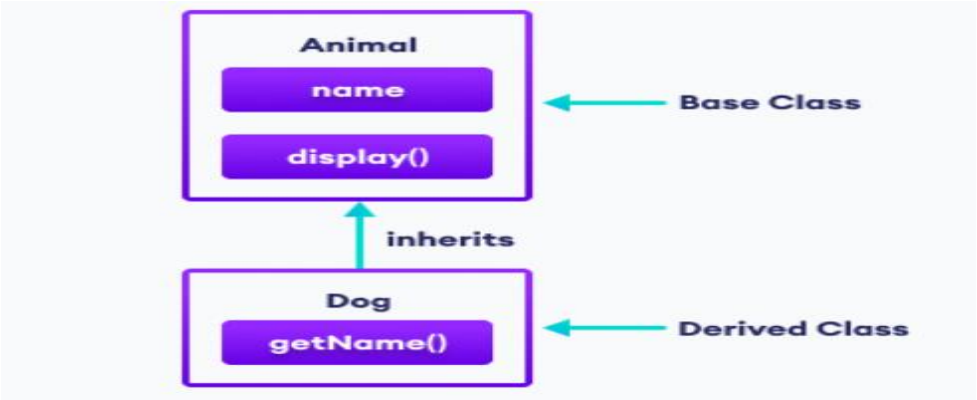
```
        }
        static void Main(string[] args)
        {
            Line line = new Line();
            line.setLength(6.0);
            Console.WriteLine("Length of line : ", line.getLength());
        }
    }
}
```

Static Members of a C# Class

We can define class members as static using the static keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

Inheritance

In C#, ***inheritance*** allows us to create a new class from an existing class. It is a key feature of Object-Oriented Programming (OOP). The class from which a new class is created is known as the *base class* (parent or superclass). The new class is called *derived class* (child or subclass) The derived class inherits the fields and methods of the base class. This helps with the code reusability in C#.



Syntax:

```
<access-specifier> class <base_class>
{
    ……..
}
class <derived_class> : <base_class>
{
    ……..
}
```

Example:

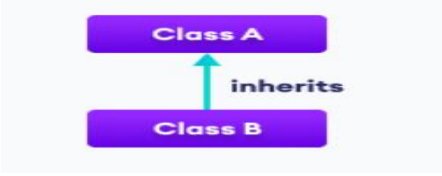
```
public class A
{
    //fields and methods
}
class B : A
{
    //fields and methods of A
    //fields and methods of B
}
```

Types of inheritance

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

Single inheritance

In single inheritance, a single derived class inherits from a single base class.

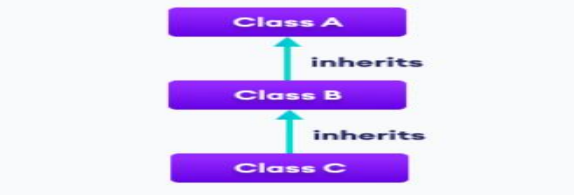


Example:

```
class A
{
        public void Display()
        {
                Console.WriteLine("This is class A");
        }
}
class B : A
{
        public void Show()
        {
                Console.WriteLine("This is class B");
        }
}
class Program
{
        static void Main(string[] args)
        {
                B obj = new B();
                obj.Display();
                obj.Show();
        }
}
```

Multilevel inheritance

In multilevel inheritance, a derived class inherits from a base and then the same derived class acts as a base class for another class.



Example:

```
class A
{
        public void Display()
        {
                Console.WriteLine("This is class A");
        }
}
class B : A
{
        public void Show()
        {
                 Console.WriteLine("This is class B");
        }
}
```

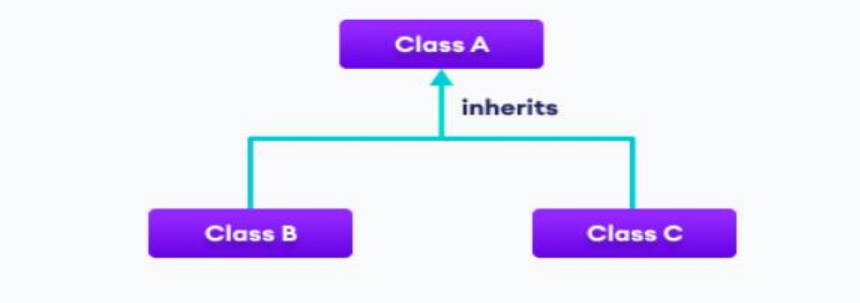
Example:

```
class A
{
        public void Display()
        {
                Console.WriteLine("This is class A");
        }
}
class B : A
```

```
{
        public void Show()
        {
                Console.WriteLine("This is class B");
        }
}
```

Hierarchical inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class.



Example

```
class A
{
        public void Display()
        {
                Console.WriteLine("This is class A.");
        }
}
class B : A
{
        public void Display()
        {
```

```
            Console.WriteLine("This is class B.");
        }
}
class C : A
{
        public void Display()
        {
            Console.WriteLine("This is class C.");
        }
}
class Program
{
        static void Main(string[] args)
        {
            A a = new A();
            B b = new B();
            C c = new C();
            a.Display();
            b.Display();
            c.Display();
        }
}
```

Multiple inheritance

In multiple inheritance, a single derived class inherits from multiple base classes. **C# doesn't support multiple inheritance.** However, we can achieve multiple inheritance through interfaces. An interface is a fully un-implemented class used for declaring only abstract members. It allows us to define only abstract members, especially abstract methods or abstract properties. The implementation of interface members (abstract methods) is provided by the child class of the interface. The class which implements the interface must provide the implementation of all the methods that are declared inside the interface without fail.

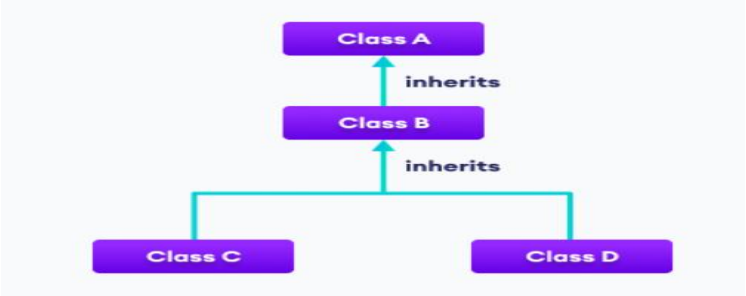
Example:

```
interface I1
{
        void Method1();
}
interface I2
{
         void Method2();
}
class MyClass : I1, I2
{
        public void Method1()
        {
                 Console.WriteLine("Method1() called.");
         }
         public void Method2()
         {
                Console.WriteLine("Method2() called.");
```

```
        }
}
class Program
{
   static void Main(string[] args)
   {
      MyClass obj = new MyClass();
      obj.Method1();
      obj.Method2();
   }
}
```

Hybrid inheritance

Hybrid inheritance is a combination of two or more types of inheritance. The combination of multilevel and hierarchical inheritance is an example of Hybrid inheritance.



Example:

```
interface IShape
{
        void Draw();
}
class Circle : IShape
```

```
{
        public void Draw()
        {
                Console.WriteLine("Drawing a circle.");
         }
}
class Rectangle : IShape
{
        public void Draw()
        {
                Console.WriteLine("Drawing a rectangle.");
        }
}
class Program
{
        static void Main(string[] args)
        {
                IShape circle = new Circle();
                IShape rectangle = new Rectangle();
                circle.Draw();
                rectangle.Draw();
        }
}
```

Polymorphism

The word polymorphism means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'. Polymorphism can be static or dynamic. In static polymorphism the response to a function is determined at the compile time. In dynamic polymorphism, it is decided at run-time.

Abstract class

Abstract classes in C# are special classes that provide a blueprint for other classes to inherit from. They cannot be instantiated, they can have both abstract and non-abstract methods, but abstract methods must be overridden by the derived classes. Abstract classes are useful for hiding the internal details and showing only the functionality of a concept or a system. They are declared with the abstract keyword before the class or method name.

Example:

```
abstract class Language
{
   // fields and methods
}
class Program : Language
{
        static void Main(string[] args)
        {
                Program obj = new Program();
                obj.display();
                Console.ReadLine();
   }
}
```

Implementing polymorphism by method overloading

Same as a constructor, we can also overload methods. Method overloading is basically having multiple methods with same name but different arguments.

Example:

```
class Rectangle
{
    public static void Area(int x, int y)
    {
        Console.WriteLine(x*y);
    }
    public static void Area(int x)
    {
        Console.WriteLine(x*x);
    }
    public static void Area(int x, double y)
    {
        Console.WriteLine(x*y);
    }
    public static void Area(double x)
    {
        Console.WriteLine(x*x);
    }
    static void Main(string[] args)
    {
        Area(2,4);
```

```
            Area(2,5,1);
            Area(10);
            Area(2.3);
        }
}
```

Method Overriding

It is a type of polymorphism. Method overriding means having two methods with same name and same parameters with base and derived classes. You can override the functionality of the base class to create a same method name with same parameters in a derived class. Virtual and Override keywords are used to achieve method overriding.

Example:

```
class Animal
{
        public virtual void sound()
        {
                Console.WriteLine("This is a parent class");
        }
}
class Dog : Animal
{
        public override void sound()
        {
                Console.WriteLine("Dogs bark");
        }
}
```

```
class Test
{
        static void Main(string[] args)
        {
                Dod d = new Dog();
                d.sound();
        }
}
```

MODULE-3

C# interfaces:

It is a blueprint for a class. It defines set of methods and properties that a class must implement. Interfaces help to achieve abstraction and provide a way to achieve multiple inheritance. C# interfaces can be divided into multiple sections such as methods, properties, events and indexes.

Declaring interface in C#:

By using the keyword interface we can declare an interface.

```
public interface InterfaceName
{
 //only abstract methods
}
// For example
public interface Example
{
 void show();
}
```

Implementing C# interface:

```
namespace InterfaceDemo
{
        public interface A
        {
                void method1();
                void method2();
        }
        interface B : A
        {
                void method3();
        }
```

```
class MyClass : B
{
        public void method1()
        {
                Console.WriteLine("implement method1");
        }
        public void method2()
        {
                Console.WriteLine("implement method2");
        }
         public void method3()
         {
                Console.WriteLine("implement method3");
         }
 }
 class Program
{
        static void Main(string[] args)
        {
                MyClass obj = new MyClass();
                 obj.method1();
                obj.method2();
                obj.method3();
                Console.WriteLine("Press any key to exist.");
                Console.ReadKey()
        }
}
}
```

Accessing C# interface:

By creating an instance of the class, you can access the interface members using dot notation,

Example:

```
public interface Flyable
{
        void Fly();
}
public class Bird : Flyable
{
        public void Fly()
        {
                Console.WriteLine("The bird is Flying!!");
        }
}
```

Derived interfaces:

Derived interfaces in c# are interfaces that extend or inherit from other interfaces. They can add additional members such as methods, properties on the top of the members inherited from the base interface.

Example:

```
public interface shape
{
        void draw();
}
public interface Circle : shape
{
        double CalculateArea();
}
```

Overriding the interface in C#:

Overriding in c# allows a derived class to provide its own implementation for a method that is already defined in its base class. To override a method in c# u have to use override keyword. Method in the derived class must have same name, return type and parameter list as the method in the base class.

Example:

```
using System;

namespace MyApplication

{

        class Animal // Base class (parent)

        {

                public void animalSound()

                {

                        Console.WriteLine("The animal makes a sound");

                }

        }

        class Pig : Animal // Derived class (child)

        {

                public void animalSound()

                {

                        Console.WriteLine("The pig says: wee wee");

                }

        }

         class Dog : Animal // Derived class (child)

         {
```

```
        public void animalSound()
        {
            Console.WriteLine("The dog says: bow wow");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Animal myAnimal = new Animal(); // Create a Animal object
            Animal myPig = new Pig(); // Create a Pig object
            Animal myDo g = new Dog(); // Create a Dog object
            myAnimal.animalSound();
            myPig.animalSound();
            myDog.animalSound();
        }
    }
}
```

Output:

The animal makes a sound

The pig says: wee wee

The dog says: bow wow

C# Structures:

Structure is a value type data type. It helps u to make a single variable hold related data of various data types. The struct keyword is used. Structures are used to represent a record.

Defining a Structure:

To define a structure u must use the struct statement. The struct statement defines a new data type with more than one member for Ur program

Example:

```
struct Books
{
        public string title;
        public string author;
        public string subject;
};
```

Features:

1. It Can have methods, indexers, operator methods and events
2. It Can have constructors
3. It Cannot have destructors and default constructors
4. It cannot be used as a base for other structures or classes
5. Structure can implement one or more interfaces
6. Structure members cannot be specified as abstract, virtual or protected

Example:

```
Public struct Rectangle
{
        public int width, height;
```

```
}
Public class TestStructs
{
    public static void main()
    {
        Rectangle r= new Rectangle();
        r.width=4;
        r.height=5;
        console.WriteLine("Area of Rectangle is:"+(r.width+r.height));
    }
}
```

Enum in C#:

An enum is a special "class" that represents a group of constants. To create an enum, use the enum keyword and separate the enum item with comma.

Example:

```
enum Level1
{
    Low,
    Medium,
    High
}
class program
{
    static void main(string[] args)
```

```
        {
                Level myVar=Level.Medium;
                console.writeLINE(myVar);
        }
}
```

Enum in switch statement:

```
enum level{ low, medium,high}
static void main(string[] args)
{
        level a= level. medium;
        {
                switch(a)
                {
                        case level. low: console.writeLine(" Low level");
                                                break;
                         case level. medium: console.writeLine(" Medium level");
                                                break;
                         case level. high: console.writeLine(" High level");
                                                break;
                }
        }
}
```

Exception handling in C#:

Exception is a problem that arises during the execution of the program. In c# exception is an event or object which is thrown at run time. c# exception handling built upon 4 keyword try, catch, finally and throw.

Example:

```
using System;

class ExceptionTestClass

{

        public static void Main()

        {

                int x = 0;

                try

                {

                        int y = 100 / x;

                }

                catch (ArithmeticException e)

                {

                        Console.WriteLine($"ArithmeticException Handler: {e}");

                }

                catch (Exception e)

                {

                        Console.WriteLine($"Generic Exception Handler: {e}");

                }

        }

}
```

Types of Exception handling:

1	**System.IO.IOException** Handles I/O errors.
2	**System.IndexOutOfRangeException** Handles errors generated when a method refers to an array index out of range.
3	**System.ArrayTypeMismatchException** Handles errors generated when type is mismatched with the array type.
4	**System.NullReferenceException** Handles errors generated from referencing a null object.
5	**System.DivideByZeroException**

	Handles errors generated from dividing a dividend with zero.
6	**System.InvalidCastException** Handles errors generated during typecasting.
7	**System.OutOfMemoryException** Handles errors generated from insufficient free memory.
8	**System.StackOverflowException**

C# delegates:

A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Syntax:

Delegate Syntax

```
[access modifier] delegate [return type] [delegate name]([parameters])
```

Example:

```
using System;
 namespace Delegates
{
        public delegate int operation(int x, int y);
        class Program
        {
                static int Addition(int a, int b)
                {
                        return a + b;
                }
                static void Main(string[] args)
                {
                        operation obj = new operation(Program.Addition);
                        Console.WriteLine("Addition is={0}",obj(23,27));
                        Console.ReadLine();
                }
        }
}
```

Multicast Delegates:

A Multicast Delegate in C# is a delegate that holds the references of more than one function.

Example:

```
namespace MulticastDelegateDemo
{
    public class Rectangle
    {
        public void GetArea(double Width, double Height)
        {
            Console.WriteLine(@"Area is {0}", (Width * Height));
        }
        public void GetPerimeter(double Width, double Height)
        {
            Console.WriteLine(@"Perimeter is {0}", (2 * (Width + Height)));
        }
        static void Main(string[] args)
        {
            Rectangle rect = new Rectangle();
            rect.GetArea(23.45, 67.89);
            rect.GetPerimeter(23.45, 67.89);
            Console.ReadKey();
    }    }
}
```

Events:

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications.

Example:

```
using System;
 namespace Delegates
{
        public delegate void DelEventHandler();
        class Program
        {
                public static event DelEventHandler add;
                 static void Main(string[] args)
                {
                         add += new DelEventHandler(USA);
                        add += new DelEventHandler(India);
                        add += new DelEventHandler(England);
                        add.Invoke();
                        Console.ReadLine();
                }
                static void USA()
                {
                        Console.WriteLine("USA");
                }
                static void India()
```

```
        {
            Console.WriteLine("India");
        }
        static void England()
        {
            Console.WriteLine("England");
        }
    }
}
```

MODULE-4

Introduction to Asp.net core:

ASP.NET Core is the new web framework from Microsoft. ASP.NET Core is the framework you want to use for web development with .NET. It is a cross-platform high-performance, open-source framework for building modern, cloud based, internet-connected applications.

With ASP.NET Core, you can:

1. Build web app and services, Internet of Things (IoT) apps, and Mobile backends
2. Use your favourite development tools on windows, macOS and Linux
3. Deploy to the cloud or on-premises
4. Run on .NET Core

Features of Asp.net Core:

1) **Cross platform:**
   Asp.net is a cross-platform, which means you can develop and deploy applications on different operating systems, like windows, macOS and Linux.
2) **MVC architecture:**
   Asp.net follows the Model-View-Controller(MVC) architectural pattern, which helps in organizing code, separating concerns, and making the development process more maintainable.
3) **Razor pages:**
   Razor pages is a new feature in Asp.net that simplifies the development of web pages by combining both the view and the code into a single file, making it easier to understand and maintain.
4) **Security:**
   Asp.net has built-in-security features like authentication and authorization, which helps in securing your application from unauthorized access and attacks.
5) **Entity Framework:**
   Asp.net integrates with the entity framework, a powerful ORM (Object Relational Mapping) tool that simplifies database interactions and provides an object-oriented approach to data storage.

6) **Web API:**

Asp.net provides built in support for building RESTful APIs (Application Programming Interface), allowing u to expose your applications functionality over HTTP and interact with other systems and clients.

7) **Caching:**

Asp.net offers various caching mechanisms to improve performance and reduce the load on the server by storing frequently accessed data in memory.

8) **Localization and Globalization:**

Localization and Globalization allowing u to build an application that can be easily translated into different languages and adapt to different cultures.

9) **Scalability:**

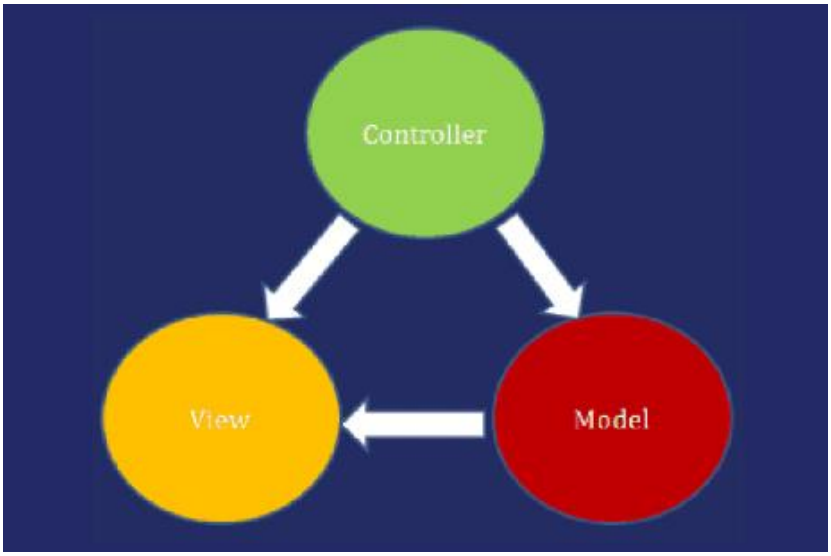
Asp.net designed to handle large scale applications. It provides features like load balancing, and session management to ensure your applications can scale efficiently

10) **Deployment options:**

Asp.net offers various deployment options, including self-hosting, cloud deployment and containerization giving you flexibility in how you deploy your application.

MVC Design Pattern

The Model View Controller (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different object.



Components of MVC Pattern:

1) **Model (Data):**
   it represents the data. It encapsulates the data and provides methods to access it.
2) **View (User Interface):**
   it displays the data from the model to the user and interacts with user to gather input.
3) **Controller (Logic):**
   it receives the user input from the view, processes it, and updates the model.

Steps to create Asp.net core application:

Step 1) The first step involves the creation of a new project in Visual Studio. After launching Visual Studio, you need to choose the menu option New->Project.

Step 2) The next step is to choose the project type as an ASP.Net Web application. Here we also need to mention the name and location of our project.

Step 3) In the next screen, you have to choose the type of ASP.net web application that needs to be created. In our case, we are going to create a simple Web Form application.

Step 4) Now, it's time to add a Web Form file to the project. This is the file which will contain all the web-specific code for our project.

Step 5) In the next screen we are going to be prompted to provide a name for the web form

Step 6) The next step is to add the code, which will do the work of displaying "Hello World." This can be done by just adding one line of code to the Demo.aspx file.

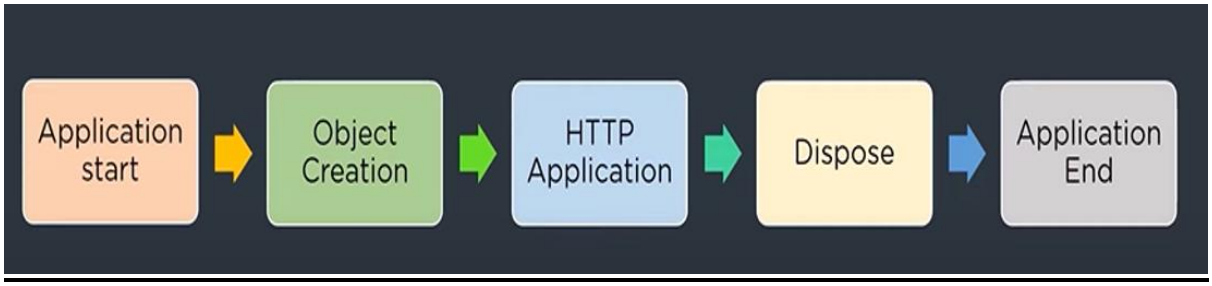
Example:

```
<html xmlns="www.w3.org/1999/xhtml">
<head runat="server">
        <title></title>
</head>
<body>
        <form id="form1" runat="server">
        <div>
        <%Response. Write( "HeIIo World"); %>
```

```
        </div>
        </form>
</body>
</html>
```

Life cycle of Asp.net Core:

A) Application Life Cycle:



1) **Application start:**
   Asp.net application starts when a request is made by a user. This request is to the web server for the asp.net application. This happens when the first user normally goes to the home page for the application for the first time. Usually in this method all global variables are set to their default values.
2) **Object Creation:**
   The next stage is the creation of the HttpContext, HttpRequest and HttpResponse by the web server. The HttpContext is just a container for the HttpRequest and HttpResponse objects. The HttpRequest contains the information about the current request including cookies and browser. The HttpResponse object contains the response that is sent to the client.
3) **HTTP Application:**
   This object is created by the web server. This object is used to process request sent to the application.
4) **Dispose Application:**
   This event is called before the application instance is destroyed.
   During this time one can manually release any undamaged resources.
5) **Application End:**
   This is the final part of the application. In this part, the application is finally unloaded from memory

Page Life Cycle:

1) **Page Request:** When the page is requested, the server checks if it is requested for the first time. If so, then it needs to compile the page, parse the response and send it across to the user. If it is not the first time the page is requested, the cache is checked to see if the page output exists. If so, that response is sent to the user.
2) **Page Start:** During this time, 2 objects, known as the Request and Response object are created. The Request object is used to hold all the information which was sent when the page was requested. The Response object is used to hold the information which is sent back to the user.
3) **Page initialization:** During this time, all the controls on a web page is initialized. So, if you have any label, textbox or any other controls on the web form, they are all initialized.
4) **Page Load:** This is when the page is actually loaded with all the default values. So, if a textbox is supposed to have a default value, that value is loaded during the page load time.
5) **Validation:** Sometimes there can be some validation set on the form. For example, there can be a validation which says that a list box should have a certain set of values. If the condition is false, then there should be an error in loading the page.
6) **Post event handling:** This event is triggered if the same page is being loaded again. This happens in response to an earlier event. Sometimes there can be a situation that a user clicks on a submit button on the page. In this case, the same page is displayed again. In such a case, the Post back event handler is called.
7) **Page Rendering:** This happens just before all the response information is sent to the user. All the information on the form is saved, and the result is sent to the user as a complete web page.
8) **Unload:** Once the page output is sent to the user, there is no need to keep the ASP.net web form objects in memory. So the unloading process involves removing all unwanted objects from memory.

Advantages of Asp.net core:

1. Since it is open-source, anyone can use it easily.
2. Bug-fixing is easy and fast.
3. It provides flexibility in development. Because a number of hosting options and web servers are available to host an ASP.net Core application.
4. As can be seen, many different kinds of applications can be built using Asp.net Core including hybrid and native applications.
5. Besides, the applications are more maintainable and easier to test.

Disadvantages of Asp.net core:

1. Security
2. Costly
3. Documentation is not exactly up to the mark
4. ASP.NET core not good enough
5. Making changes in the app
6. Porting ASP application from one server to another is expensive