* There are 6 languages that comes under .NET (C#, F#, VB.NET, J#, JScript.NET, ASP.NET, VC++, powershell, etc)
  ↳ officially support by Microsoft

Date: .....
Page: .....

- C# is a part of .NET framework and currently in version 4.6.2.

* We can use C# for building variety of applications
- (Desktop) Windows application: using console application or winform application.
- Mobile application: Xamarian lets you build native apps for both Android & iOS.
- web application: using ASP.NET web forms or ASP.NET MVC
- Gaming application: unity

C# could theoretically be compiled to machine code, but in real life, it is always used in combination with .NET framework. Therefore, applications written in C#, requires the .NET framework to be installed on the computer running the application.

Language interoperability is the ability of code to interact with code that is written by using different programming language. eg. we can implement the code written in C# with ASP.NET.

* .NET Framework.

→ .NET is a programming framework created by Microsoft that developers can use to create applications more easily. A framework is just a bunch of code that the programmer can call without having to write it explicitly.

- It provides a controlled programming environment where software can be developed, installed and executed on Windows-based operating systems.

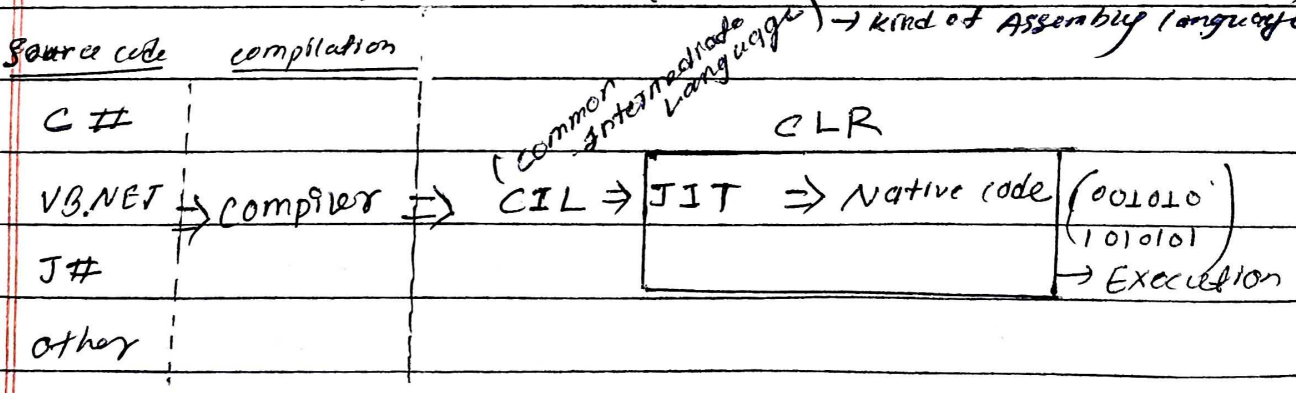
Date : .....
Page: .....

- A programmer can develop applications using one of the language supported by .NET (eg. C#, J#, etc)
- Microsoft introduced C# as a new programming language to address the problem posed by traditional languages.

* CLR (Common Language Runtime)

- It is the foundation of the .NET framework
- Acts as an execution engine for the .NET framework
- Manages the execution of program and provides a suitable environment for programs to run.
- provides a multi-language execution environment

* How .NET language work (Architecture of .NET Framework)

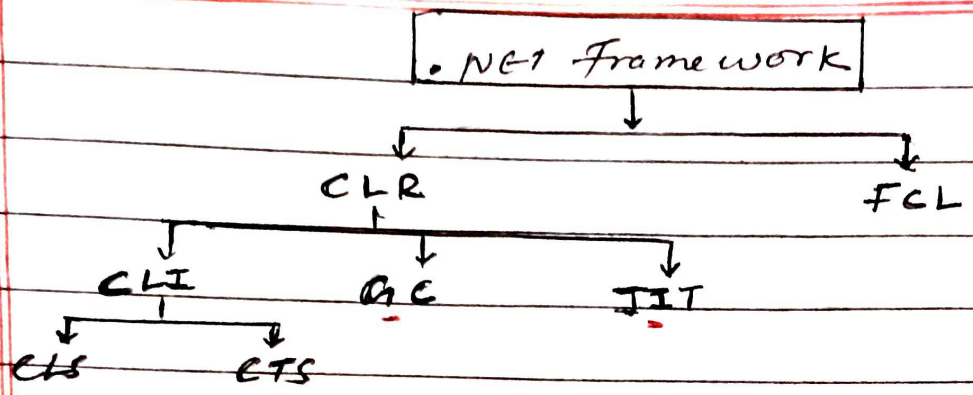


The source code is converted into CIL by specific language compiler. The CIL code is then converted to a native code to the OS. This is done by JIT compiler present in CLR. The CLR converts the CIL code to the machine code. once this is done, the code can be directly executed by the CPU.

Date: .....

Page: .....

✱ Components of .NET framework



→ The two major components of .NET Frame work are CLR and FCL.

- CRL (common Language Runtime)

→ It is the execution engine that handles running applications. It provides services like thread management (memory), garbage collection type-safety, exception handling and more. It is also responsible for converting source code into machine readable code

- FCL (.NET framework class Library)

→ It is a collection of pre-built classes, methods and components use to develop application. It provides rich set of functionalities that handles complex tasks, from file handling to database connectivity. The core libraries are sometimes collectively called Base class Library (BCL) & entire framework is called FCL.

- CLI (common language Infrastructure)

→ It standardizes the execution and development process of .NET applications. It includes JIT compiler which translates Intermediate Language into native machine code enabling cross-platform compatibility.

Date : .....

Page : .....

- GC (Garbage Collector)

→ It is used to provide the Automatic memory management feature deallocating unused objects and reducing risk of memory leaks.

- JIT (Just in - Time compile)

→ It is responsible for converting CIL into machine code using CLR environment.

- CLS (Common Language Specification)

→ It defines a set of rules and guidelines that programming languages must follow to be compatible within the .NET environment.

- CTS (Common Type System)

→ It is responsible for understanding the data types of .NET programming and converting them into CLR understandable format i.e Value Type and Reference Type.

* Namespace

→ It is collection of classes, interfaces, structs, enums and delegates used to organize code.

Note; If you don't want to use namespace you can use fully Qualified Name (FQN)

Eg: using System; Here, System is a namespace
Console.Write("Hello"); ⇒ System.Console.Write("Hello");

Eg2, namespace demo { → declaring own namespace helps to
class program { control the scope of class & method
name

- C# → C# compiler → CIL → CLR → Machine code (01)
  - CIL → common Intermediate Language
  - CLR → common Language Runtime

C#

Date : .....

Page : .....

* .NET is a developer platform made up of tools, programming languages and libraries for building many different types of applications.

* .NET Framework is the original implementation of .NET. It supports running websites, services, desktop apps and more on windows.

* .NET Core is a cross-platform implementation for running websites, services and console apps on Windows, Linux, and mac OS. .NET Core is open source on GitHub.

* Xamarin/Mono is a .NET implementation for running apps on all the major mobile OS, including iOS and Android.

  .NET Standard is a formal specification of the APIs that are common across .NET implementations. This allows the same code and libraries to run on different implementations.

* Namespace : It is used in C# in two ways.
  - 1st to organize its many classes

eg: System.Console.WriteLine("Hello World!");

Here, System is a namespace & Console is a class in namespace.

The using keyword can be used so that complete name isn't required.

eg:
```
using System;
Console.WriteLine("Hello World!");
```

- 2nd, declaring your own namespace can help you control the scope of class and method names.

eg: namespace 'sample' {
    class program {

Date: .....
Page: .....

→ C# is a simple and powerful object oriented programming language developed by "Microsoft" that runs on .NET framework.

- C# is type-safe OOP Language eg. int num = 'raj'; X (error)
  it means string type 'raj' is not safe with int num.
- It is developed by Anders Hejlsberg & his team during development of .NET Framework.
- The 1st version of C# was "C# 1.0" released with Visual studio.NET 2002. and latest version is 'C# 9.0' released in sept-2020
- It can be used to develop desktop, web and mobile applications, database application, games and much more
- C# adopt almost all the features of C, C++, and Java programming but there are some advanced features which are available only in C#.
- It is case sensitive programming language.
- .cs is the extension of C# file.

Syntax :

```
using System;
namespace anyname
{
    class program {
public static void Main (string[] args) {
        Console.WriteLine ("HelloWorld");
}}}
```

→ Namespace is a container for set of related classes & other types for better code organization

→ namespace is used to organize code and avoid naming conflict.

+ → Concatination operator
$ { } → string interpolation

↳ CLS
↳ Clear terminal

Date: .....
Page: .....

Data types & variables

→ Datatype: Specifies the different sizes and values that can be assigned on the variable. eg. int a = 10;
(int → datatype, a → variable)

Datatype	sizes	
int	4 bytes	
char	2 bytes	precision → (before .) + (after .)
bool	1 bit	
float	4 bytes	e.g. float b = 34.4f
double	8 bytes	e.g. double d = 54.5D (not mandatory)
string	2 bytes each character	
long	8 bytes	
decimal	16 bytes (temporary storage)	

precision: 3.7 86.485 → precision = 7

float → 7 digits
double → 15-16 digits
decimal → 28-29 decimal places

→ variable refers to the memory location name where we store values. eg: int num, string name;

```
→ using System;                    // Importing namespace
  Class Test                       // class declaration
  {
    static void Main()             // method declaration
    {
      int a = 5*10;                // // → single line comment
      Console.WriteLine(a);        /* ... */ → multiline comment
    }
  }

  double num = 98420 43002;
  Console.WriteLine("The no. is" + num);
```

→ All of variables must be identified with unique names called identifiers. It can be short name (like x & y) or more descriptive names (age, sum, totalVolume) but must begin with "letter, _ or @"

Note: Keyword can't be used as variable (identifiers)

Date: .....

Page: .....

* Keywords (77)

→ Keywords are pre-defined set of reserved words that have special meaning in a programe.

- C# Keyword list

using	string	break	as	checked	null
int	true	continue	is	class	object
char	false	if	abstract	const	private
float	static	else	base	enum	protected
double	this	while	case	default	public
short	byte	do	try	namespace	goto
long	bool	for	catch	void	return

* Contextual Keywords

→ It is used to provide a specific meaning in the code, but it is not reserved word in C#.

some contextual keywords, such as partial and where have special meanings in two or more contexts.

add	get	notnull	var
and	global	on	where
alias	group	or	when
ascending	into	orderby	with
by	join	remove	yield
descending	let	select	
equals	not	set	
from	nameof	value	

- String → ReadLine()
- Integer → ToInt16/32/64()

Note:- When you take input from the user you will receive in string format & then you change it into integer.

* User input → string format

(Input & output at run time)

1) For string type

```
class program
{
    static void Main()
    {
        String name;
        Console.Write("enter your name:");
        name = cosole.ReadLine();
        Console.Write("my name is :" + name);
        Console.ReadKey(); // wait for user to press any key
    }
}
```

- Console.ReadLine(); → is used to take input from users. & also to hold the window from disappearing (means it was waiting for user to input data so screen stay on hold & we can see the output for longer time)

2) For integer type

```
    int num;
    Console.Write("Enter any number:");
num = Convert.ToInt32(Console.ReadLine());
Console.Write("The number is:" + num);
}
```

or

```
String name;
Console.Write("Enter any number:");
name = Console.ReadLine();
int num = Convert.ToInt32(name);
Console.Write("The number is:" + num);
}
}
```

Console.Write("sum = {0}", sum); → Place holder syntax

→ If we declare any member of class as static we can access it without creating object of that class.

Date: .....
Page: .....

→ program to add two numbers

```
class Program
{
    public static void Main(String[] args)
    {
        int a=10, b=20;
        int sum = a+b
        Console.Write(sum);
    }
}
```

// using function (passing parameters) ⇒

```
static void sum(int a, int b)
{
    int sum = a+b;
    Console.Write("sum=" + sum + (a+b));
}
static void Main(S[] args){
    sum(10,20);
}
```

// Taking input from user

```
int a, b, sum;
Console.Write("enter Any Two Numbers: ");
a = Convert.ToInt32(Console.ReadLine());
b = Convert.ToInt32(Console.ReadLine());
sum = a+b;
Console.Write("sum of Two Numbers: " + sum);
```

→ concatenation syntax

$ string interpolation $"sum of {a} and {b} is {sum}";

✗ Type casting (conversion) (To convert from one datatype to another type)

• Implicit casting (Auto)

→ char - int - long - float - double

eg: int x = 3;
double y = x; (int - double) ✓
int z = y; (double - int) ✗ not allowed

• Explicit casting

→ int x = (int) 3.5;
(double - int)
→ generate by user

• float a = Convert.ToInt32(3.55) → O/P → 4

Convert.ToDouble()
convert.ToString()

- shortcut key for comment ↳ ctrl + k ctrl + c
  ↳ uncomment ↳ ctrl + k ctrl + u

Date: .....

Page: .....

* Control flow Statement

(I) Conditional statement
- if
- if-else
- nested if
- else if ladder
- switch statement

(II) Looping statement
- while
- do while
- for loop
- foreach loop

(III) Jumping statement
- break
- continue
- return
- goto

(1) Conditional Statement / Selection Statement

```
class Program {
    public static void Main(string[] args) {
        int ch;
        Console.WriteLine("Enter your choice!");
        ch = Convert.ToInt32(Console.ReadLine());
        switch (ch) {
            case 1:
                Console.WriteLine("I am present");
                break;
            case 2: Console.WriteLine("I am absent");
                break;
            default: Console.WriteLine("Invalid Input");
                break;
        }
    }
}
```

Date : .....

Page : .....

(11) Looping Statement

1, while :

```
class program {
    Public static void Main( ) {
        int i=1;
        while( i<=10) {
            Console.write(i+" ");
            i++;
        }}
```

2, do while :

```
int i=1;
do {
    console.write (i);
    i++;
} while (i<=10);
}}
```

3, for loop :

```
for (int i =1; i<=10; i++) {
    Console. write (i);
}
```

4, foreach :

```
foreach ( type  var  in  array-name )
        int     i    in   a
{
    // statements;
}
int[]a = {1,2,3,4,5};
foreach (int i in a) {
    console.writeline(i);
}
```

eg:

Date :

Page :

(iii) Jumping statement

→
```
for (int i=1; i<=5; i++){
    if (i==3){
        break;
    }
    Console.WriteLine(i+" ");
}
```
o/p: 1 2

```
if(i==3 || i==5)
    continue;
```
o/p: 1 2 4

- return

→
```
static void Main(){
    int r = add(10,20); // calling function
    Console.Write(r);
}
static int add(int x, int y){ // defining function
    return x+y;
}
```

- goto → to display repeated task repeatedly without using loop

→
```
static void Main(){
    int i=1;
    loop:
    if(i<=10){
    Console.Write(i+" ");
    i++;
    goto loop;
    }}
```
o/p: 1,2,3,4,5,6,7,8,9,10

→
```
int i=1;
if(i<=10){
Console.Write(i+" ");
i++;
}
```
o/p: 1

x + y → operator
↓
operands

Date: .....
Page: .....

* Operator

→ It is a symbol that operates on a value or variable.

- It is used to perform logical & mathematical operation

* Types

- Arithmetic (+, -, *, /, %, ++, --)
- Relational (==, !=, >, <, >=, <=)
- Logical (&&, ||, !)
- Assignment (=, +=, -=, *=, /=
- Bitwise (&, |, ^, <<, >>)
- Miscellaneous (sizeof(), typeof(), &, *, ?: (Ternary), is, as)

Eg: Ternary operator

```
int a = 100, b = 50;
int r = (a>b) ? a : b;
Console.Write(r);
```

o/p: 100 → 500

o/p: 500

* Stack: Block of memory for storing local variables and parameters.

* Heap: Block of memory in which objects (i.e. reference-type instances) reside. Whenever new object is created, it is allocated on the heap and a reference to that object is returned.

Date: .....

Page: .....

* Array

-) Array is a variable which help us to store multiple elements of same type.

Note: The base index of array starts with 0 and ends with (n-1)

* Types of array

- 1 D array
- 2 D / multidimensional array

Eg: using System;
namespace Array {
class Program {
static void Main() {
int[] a = {10,20,30,40,50}; // {'a','b','c','d','e'}
foreach (int i in a) → foreach (type var in array)
OR, for (int i=0; i<a.Length; i++)
{
Console.Write(i + " "); }
Console.WriteLine(a[i]);
}}}

Eg2: Program for calculating sum of all array elements

using System;
class Program {
static void Main (string[] args) {
int[] a = new int[10];
int n, i, sum=0;
Console.WriteLine("Enter total elements:");
n = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("enter array elements:");
for(i=0; i<n; i++)
a[i] = Convert.ToInt32(Console.ReadLine());
for(i=0; i<n; i++)
sum = sum + a[i];
Console.WriteLine("sum = "+sum);
}

int [,] matrix = new int [,];

Date : .....

Page : .....

- 2D Array / Multidimensional Array

Rectangular array

```
static void Main()
{
    int[,] a = new int[2,2];
Console.WriteLine("Enter Array Elements:");
for(int i=0; i<2; i++)  // rows
{
for(int j=0; j<2; j++)  // column
{
 a[i,j] = Convert.ToInt32(Console.ReadLine());
}}
Console.WriteLine("Matrix Elements:");
for(int i=0; i<2; i++)
{
for(int j=0; j<2; j++)
{
 Console.Write(a[i][j] + " ");
}
Console.WriteLine();
}
}
```

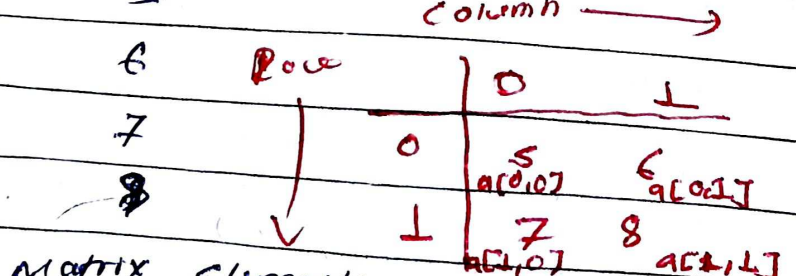
O/P: Enter Array Elements.

5

6

7

8



Matrix Elements:

5 6

7 8

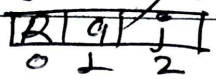
Date : .....

Page: .....

String

→ collection of character enclosed by double quotes " "

Eg. `string str = "Raj";`



`Console.Write(str);` → `(str[2])` → j

length → `Console.Write(str.Length);` // 3

`string str2 = "Ram";`

compare → `Console.Write(str.Equals(str2));` // false

concat → `Console.Write(str + " " + str2);` // Raj Ram

Uppercase → `Console.Write(str.ToUpper());` RAJ

→ Multidimensional arrays come in two varities → Rectangular, → Jagged

- Rectangular arrays represent an n-dimensional block of memory
- Jagged arrays are "arrays of arrays."

```
using System;
    class RectangularArray {
        static void Main(string[] args) {
        int[,] arr = new int[,] { {1,2,3}, {4,5,6}, {7,8,9} };
        for(int i=0; i<3; i++) {          // arr.GetLength(0)
            for(int j=0; j<3; j++) {      // arr.GetLength(1)
                Console.Write(arr[i,j] + " ");
            }
            Console.WriteLine();
        }
    }
```

you can directly assign value

Note: Rectangular arrays are declared using commas to separate each dimension. eg: `int[,] matrix = new int[3,3];` → you can omit array size

Date: .....

Page: .....

* Nested loop

```
int i, j;
for(i=1; i<=3; i++)
{
for(j=1; j<=i; j++)
{
Console.Write("* ");
}
Console.WriteLine();
}
```

* Jagged Array

```
using System;
class JaggedArray {
static void Main(string[] args) {
int[][] arr = new int[][] {
new int[] {1,2},
new int[] {1,2,3},
};
for (int i=0; i<arr.Length; i++) {
for(int j=0; j<arr[i].Length; j++) {
Console.Write(arr[i][j] + "\t");
}
Console.WriteLine();
}}}
```

foreach(int[] row in arr){ foreach(int col in row){ Console.Write(col + " "); }

Note: Jagged array are declared using successive square brackets to represent each dimension

eg: int[][] matrix = new int[3][]; outermost dimension but you can omit it.

OOP x'concept

- properties → variable
- behavior → method

Date: .....

Page: .....

* class (or prototype)

→ It is a blueprint of object where we define the properties and behavior of objects.

* object → collection of data & method

→ It is an instance of class to represent real entity containing properties & behavior
(in c#, an obj is created using "new" keyword)

Note: class is a virtual but object is real

```
eg: public class person {          'class name'
access modifier   string name;
                  int age;          > 'properties / data'
                  +
                  void eat() {
                  void sleep() {    > behavior / method
                  } }
```

```
eg: 2   class Mobile {
            name;
            cost;
            storage;
            +
            calling();
            chatting();
            music();
        }
```

Date : .....

Page : .....

Eg: class Mobile {

int price; → you can put value here also

string storage, color;

public void calling() {

console.Write("voice msg");

}

public void chatting() {

console.write("Text msg");

}

public void music() {

console.Write("song---");

}

static void Main(string[] args) {

Mobile m = new Mobile();

m.price = 10000;

m.storage = "8 GB RAM";

m.color = "black";

console.WriteLine(m.price);

console.WriteLine(m.storage);

" (m.color);

m.calling();

m.chatting();

m.music();

}}

O/p: 1000

8GB RAM

Black

voice msg Text msg song

Date: .....
Page: .....

* constructor → no return type

→ It is a special type of method whose name is same as class name and invoked (called) automatically at the time of object creation.

Note: The main purpose of constructor is to initialize the data member of new object.

* Types
  - Default
  - parameterized

```
class A {
    A ( ) {
    // statement
    } }
```

```
class A {
    A(int a) {
    // statement
    }
}
```

eg:

→
```
class Sample ( ) {
    sample ( ) {
    console.write("Raj");
    }
    static void Main ( ) {
    Sample s = new Sample( );
    }
}
```
not compulsory

o/p: Raj

→ If we haven't constructor
```
class Sample ( ) {
    public void Show ( ) {
    console.write ("Raj");
    }
    static void main ( ) {
    Sample s = new Sample( );
    s.show ( );
    }
}
```
you can simply write new Sample( );

Note: C# automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, default constructor is no longer used.

Date : .....

Page : .....

* Parameterize constructor

→ class A {

A(int num, string name)

{

console.write(num + " " + name);

}

Static void Main()

{

A r = new A(50, "Raj");

}}

O/P : 50

* Sum of two number

→ class Sum {

sum(int a, int b) {

int sum = a + b;

Console.WriteLine("sum = " + sum);

}

Static void Main(string[] args) {

Sum s = new Sum(10, 20);

}}

O/P : Sum = 30

* overloded constructor

→ class sum {

sum(int a, int b) {

console.write("sum = " + (a+b));

}

sum(int x, int y, int z) {

console.write("sum = " + (x+y+z));

}Static void Main(string[] args) {

new Sum(10,20);

new Sum(10,20,30);

}}

O/P : sum = 30

sum = 60

Date: .....

Page: .....

Encapsulation

→ Data member & method is wrapped in a single unit inside a class so that the data can't be accessed by other class.

Note: All variable should be declared as private

Syntax:

```
class A
{
    private datas
        +
    public methods
}
```

```
class person {
    private string name;
    public void setName
        (string Name) {
    name = Name;
    console.write(name);
    public static void Main() {
    person p = new person();
    p.setName("raj");
    }
}
```

Eg: →

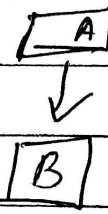
```
class A {
    private int atmPIN;
    public int getPin() {
        return atmPIN;
    }
    public void setPin(int pin) {
        atmPIN = pin;
    }
}
class B
public static void Main(string[] args) {
    A r = new A();
    r.setPin(4444);
    Console.Write("ATM pin = " + r.getPin());
}
```

Date: .....
Page: .....

* Inheritance → code reusability

→ Acquiring the properties & behavior of parent class from child class

(1) single inheritance



parent / base / super-class

child / derived / sub-class

```
class A
{
  // code
}
class B:A
{
  // code + code
}
```

Eg:

```
class Animal {
public void sounds() {
  Console.WriteLine("Animals sounds");
} }
class Dog : Animal {
public void barks() {
  Console.Write("Dog barks");
}
Static void Main(string[] args) {
  Dog d = new Dog();
  d.sounds();
  d.barks();
}
}
```

O/p: Animals sounds
Dog barks

Date : .....

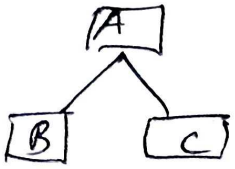
Page : .....

(ii) Multi-level

```
class A {
    // code
}
class B : A {
    // code
}
class C : B {
    // code
}
```

Eg :→

```
class Animal {
  void sound() {
  Console.WriteLine("Animals sound");
  } }
class Dog : Animal {
  void bark() {
  Console.WriteLine("Dogs bark");
  } }
class cat : Dog {
  void meow() {
  Console.WriteLine("cat meow");
  } }
  Static void Main(string[] args) {
    Cat c = new Cat();
    c.meow();
    c.bark();
    c.sound();
  }
```



Date : .....

Page : .....

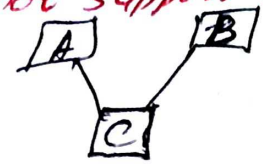
(III) Hierarchical

```
class A {
  // code
}
class B : A {
  // code
}
class C : A {
  // code
}
```

Eg: →

```
class Animal {
  void sound() {
  Console.WriteLine("Animal Sound");
}}
class Dog : Animal {
  void bark() {
  Console.WriteLine("Dog bark");
}}
class cat : Animal {
  void meow() {
  Console.WriteLine("cat meow");
}}
Static void Main(string[] args) {
Cat c = new Cat();
Dog d = new Dog();
c.meow(); d.bark();
c.sound(); d.sound();
}
```

→ multiple inheritance is not supported in c# directly for classes, but support through interface.



Date: .....

Page: .....

(IV) multiple

```
interface A {
    //code
}
interface B {
    //code
}
class C: A, B {
    //code
}
```

→ Interface is like a abstract class which only contain abstract methods & doesn't contain body.

eg:→

```
interface A {
    void add();
}
interface B {
    void sub();
}
class C: A, B {
    public void add() {
        int a=10, b=20;
        Console.WriteLine(a+b);
    }
    public void sub() {
        int a=10, b=20
        Console.WriteLine(a-b);
    }
    static void Main() {
        C r = new C();
        r.add();
        r.sub();
    }
}
```

o/p : 30
-10

```
interface shape {
    void draw();
}
interface color {
    void fillcolor();
}
class circle : shape, color {
    public void draw() {
        console.write("Drawing circle");
    }
    public void fillcolor() {
        console.write("filling color");
    }
    class program {
        static void main() {
            Circle c = new Circle();
            c.draw();
            c.fillcolor();
        }
    }
```

friend → person → teacher
parent → student

many-form

* Polymorphism

→ same object having different/multiple behaviors or functionality

* Types

- compile time
  - overloading
- Runtime
  - overriding (inheritance)

* compile time / static binding (method overloading)

→
```
class A {
    int a=20, b=20;
    public void add() {
        int sum = a+b;
        Console.WriteLine(sum);
    }
    public void add(int a, int b) {
        int sum = a+b;
        Console.WriteLine(sum);
    }
}
class main {
    public static void Main() {
        A r = new A();
        r.add();
        r.add(40,60);
    }
}
```

o/p: 30
100

→ works only on Inheritance

Date : .....

Page : .....

* Runtime / Dynamic binding (method overriding)

→
```
class A {                          virtual
    public void show() {
        Console.WriteLine("A class");
    }
}
class B : A
{
                                   override
    public void show() {                    base.show();
        Console.WriteLine("B class");
    }
}
class main
{
    public static void Main() {
        B r = new B();
        r.show();
    }
}
```

A class
B class

o/p : B class

- Method overriding is only possible in derived classes, not within the same class where the method is declared
- By default, methods are non-virtual. We can't override a non-virtual method.
- Virtual method is created in the base class using 'virtual' keyword that can be overriden in derived class using 'override' keyword

A + B

operator ← (A + B) → operand

↳ compile time polymorphism

Date :

Page :

* Operator overloading (static overloading)

⇒ To assign more than one operation on same operator.

It known as operator overloading (declared using operator keyword)

Unary (+, -, !, ~, ++, --)	Binary (+, -, *, /, %)
(contain only one operand)	(contains two operand)

eg. (+) operator is used to add two integers as well as join two strings.

① "shiv" + "korala" = "shiv korala" ⇒ concatenate

② 1 + 2 = 3 ⇒ add

③ obj1 + obj2 = obj3

obj1.num1 = 10
obj1.str1 = "shiv"

+

obj2.num1 = 20
obj2.str1 = "korala"

=

obj3.num1 = 30
obj3.str1 = "shiv korala"

```
using system;
class newClass
{
    public string str;
    public int num;
public static newClass operator + (newClass obj1,
                                   newClass obj2)
{
newClass obj3 = new newClass();
 obj3.str = obj1.str + obj2.str;
 obj3.num = obj1.num + obj2.num;
  return obj3;
}
}
class program
```

Date : .....
Page: .....

```
static void Main()
{
    NewClass obj1 = new NewClass();
    obj1.str = "Raj";
    obj1.num = 10;
    NewClass obj2 = new NewClass();
    obj2.str = "Kumar";
    obj2.num = 20;
    NewClass obj3 = new NewClass();
    obj3 = obj1 + obj2;
    Console.WriteLine(obj3.str);
    Console.Write(obj3.num);
}
}
```

* Sealed class in ~~c#~~

→ A sealed class is a class that prevents inheritance.

- The sealed class cannot be a base class as it cannot be inherited by any other class. If a class tries to derive a sealed class, c# compiler generates an error.

Syntax →

```
using system;
sealed class A
{
    // code
}
class B : A        // compiler shows error
{
}
```

Date: .....
Page: .....

→ used in method overriding

* Sealed method in C#

→ When the derived class overrides a base class method, then new method can be declared as sealed to prevents from further overriding.

→ An overridden method can be sealed by preceding the override keyword with sealed keyword.

Eg: → using system

```
class A {
public virtual void show() {
    Console.Write("A class");
}}
class B : A {
public sealed override void show() {
    Console.write("B class");
}}
class C : B {
    public override void show() { // Error
    console.write("C class");   ↳ can't override
}}                                Sealed method
class program {
public static void Main() {
    C c = new C();
    c.show();
}}
```

O/p: C class

→ Private class can't be accessed from outside the class, however, if we need it can be done with properties.

Date: .....

Page: .....

* C# properties (Get & set) → prop short cut key

→ properties are like data fields (variables), but have logic behind them.

→ It is like a combination of a variable and a method.

→ They use accessors (get and set) through which the values of the private fields can be read, written or manipulated.

syntax:

```
→ public datatype propertyName
  {
    get { /* return value; */ }
    set { /* perform actions; */ }   // assign value
  }
```

Example (RIN)

```
→ class Person {
    private string name = "Raj"; // field
    public string Name { // property
    {
      get { return name; }
      set { name = value; }
    }
  }
  class Program {
    static void Main() {
      Person p = new Person();
      p.Name = "Raj";
      Console.Write(p.Name); // O/P: Raj
    }
  }
```

→ It is better to use same name for both the property and private field, but with upper case 1st letter.

↳ if you use get only function you don't have to set value

Here, 'Name' is a property of the 'Person' class that encapsulates the 'name' field & used to access & update the private field 'name'.

Types of properties
- R/W get{} set{}
- Read only get{}
- write only set{}
- Auto implement get; set;

↳ Auto Implument

Date: .....

* Automatic properties (Short Hand)

→ Here, we donot have to define the field for the property, we only have to write get; and set; inside the property. It automatically genereates a private backing field for the property.

→
```
class Person {
    public string Name  // property
    { get; set; }
}

class Program {
    Static void Main() {
        Person p = new Person();
        p.Name = "Raj";
        Console.write(p.Name);
    }
}
```

Note: In c#, properties and indexers are mechanism to encapsulate data and provide controlled access to it. They allow to define how the data is accessed and modified outside of the class

Why Encapsulation?

- Better control of class members (reduce the possibility of code mess up)
- Fields can be made read-only (if only get method is used) or write-only (if only set method is used)
- Flexible: programmer can change one part of code without affecting other parts.
- Increased security of data.

Date: .....

Page: .....

* Making Set accessor private.

→

```
class student {
    public string firstName { get; private set; }
    public string lastName { get; private set; }
// constructor
    public student (string fname, string lname) {
        firstName = fname;
        lastName = lname;
    }
}
class program {
        static void Main ( ) {
    Student s = new student("Raj", "Kumar");
    s.firstName = "Ali";          // error
    s.lastName = "Khan";          // if you don't have
    console.WriteLine(s.firstName);   use private set Name
    console.write(s.lastName);        would have replaced
}                                     as Ali Khan.
}
```

In this example, 'student' property has public getter ('get') and a private setter ('private set'). This means that code outside the 'student' class cannot directly set the value of 'student', but the class itself can modify it internally.

→ This approach is often used when you want the property to be read-only from outside the class

→ It allows for greater encapsulation and ensures that the property's value can only be set from within class.

Date: .....
Page: .....

↑ adds logic that how can array or list store the value.

* Indexers ↳ indexer (double tap ⇆)

→ Indexers allow object to be just like an array. i.e they provide a way to access elements of a class using brackets '[]'

- Indexers are similar to properties, but are accessed via an index argument rather than a property name.
- Indexer is created using this keyword

Syntax:

→
```
public returnType this [type index]
{
    get { /* return value; */ }
    set { /* perform action; */ }
}
```

Example

→
```
class person{
    private int[] Age = new int[3];
    public int this [int index]{
get { return Age[index]; }
set { Age[index] = value; }
}}
```

(Since age cannot be -ve & size of array is 3, so for that we can add logic)

```
set{ if(index >=0 && index < Age.Length)
    if(value > 0){
        Age[index] = value;
    } else{
        console.write("Invalid value");
    }} else{
        console.write("Invalid index");
    }
```

```
class Program{
    static void Main(){
        person p = new person();
        p[0] = 5;
        console.write(p[0]);
    }}
```

(p work as indexers)

o/p: 5 // error (Invalid index if ≤3) // Invalid value (-5)

Date : .....

Page : .....

Example 2

```
using system;
class person {
    private string[] name = new string[3];
    public string this[int i] {
        get { return name[i]; }
        set { name[i] = value; }
    }
}
class program {
    static void Main() {
        person p = new person();
        p[0] = "Raj";
        p[1] = "Ram";
        p[2] = "Hari";
        for(int i = 0; i < 3; i++) {
            Console.WriteLine(p[i]);
        }
    }
}
```

o/p: Raj
Ram
Hari

Date : .....

Page : .....

Enums (Enumeration)

→ An enum is a special "class" that represents a group of constants (unchangable/read-only variables)

• To create an enum, use the enum keyword (instead of class or interface) and seperate the enum items with a comma (,).

Syntax :

→
```
enum EnumName
{
    value 1,
    value 2,
    value 3,
    // more values can be added
}
```

Example

→
```
using System;
    enum Level {
        Low, //0
        Medium, //1
        High //2
    }
    class Program {
        Static void Main() {
Console.WriteLine(Level.Medium); // Medium
Console.WriteLine((int)Level.Medium); // 1
    }
}
```

// Syntax : static class className {
// static data members
// static methods }

Date : .....
Page .....

* Static member of a class

→ If we declare any members of a class as static we can access it without creating object of that class

Eg:
```
using System;
class Addition {
public static void sum (int a, int b) {
    int sum = a+b;
    Console.write(sum);
}}
class program {
    Static void Main() {
    Addition.sum(10,20);
}}
```

OR

```
class Addition {
    static void sum(int a, int b) {
    Console.write(a+b);
}
    static void Main() {
    sum(10,20);
}}
```

Eg2:
```
using System;
static class person {
    public static int id;
    public static string name;
    public static void display() {
    Console.WriteLine($"Id = {id}, Name = {name}");
}}
class program {
    static void Main() {
    // person.id = 001;
    // person.name = "Raj";
    person.display();
}}
```

o/p: Id = 1, Name = Raj

→ you can create your own generic interfaces, class, methods, events and delegates

Date: .....

Page: .....

* Generic

→ Generics allow you to write classes, structs, methods and interfaces without specifying the data type they operate on.

* Generic class

→ Generic class can be defined by putting <T> sign after the class name. But it isn't mandatory to put "T" word in generic type definetion.

- The angle <> brackets are used to declare a class or method as generic type.

syntax: public class className <T> {
// T can be used as a placeholder for any data type
}

eg:→ using System;
class Generic <T> {
//constructor public Generic (T msg) {
Console.WriteLine (msg);
}}
class program {
static void Main( ) {
Generic <string> gen = new Generic <string> ("Hello");
Generic <int> gen1 = new Generic <int> (101);
Generic <char> gen2 = new Generic <char> ('A');
}}

O/P: Hello
101
A

Date : .....
Page : .....

Generic method

→
```
using System;
    class Generic {
    public void show<T>(T msg) {
        console.WriteLine(msg);
    }}
    class program {
    static void Main() {
        Generic gen = new Generic();
    gen.show("Hello");
    gen.show(101);
    gen.show('A');
    }}
```

C# Collection

→ In c#, collection represents group of objects by which we can perform various operations on objects such as storing, updating, deleting, retrieving, searching and sorting.

→ Collection has advantage over array. Array has size limit but collection can grow or shrink dynamically. (auto resizing)

Types of collection

- Generic collection (System.Collections.Generic) → (It holds the data of same datatype)
  - List<T>
  - Dictionary<TKey, TValue>
  - Stack<T>
  - Queue<T>
- Non-Generic collection (System.Collections) → (It holds data of different datatypes)
  - ArrayList
  - Hashtable
  - stack
  - queue

Namespace →

Date: .....
Page: .....

* Arraylist (non-generic)

→ C# Arraylist is non-generic collection. It can contain elements of any datatypes. It is defined in the 'System.Collections' namespace

using system;

→ using system.collections;

```
class program {
    public static void Array<T>(T[] arr){
foreach (T i in arr){   → for(int i=0;i<a.Length;i++){
    Console.WriteLine(i);     Console.WriteLine(a[i]);
}
static void Main(){   program p = new Program();
int[] num = {10,20,30,40};
string[] name = {"Raj", "Ram", "Hari"};
char[] alph = {'a', 'b', 'c', 'd'};   → single inverted comma in char
double[] point = {2.3, 3.5, 4.5};
    Program.Array(num);
    Program.Array(name);
    Program.Array(alph);
    Program.Array(point);
}}
```

```
class Program {
    static void Main(){
ArrayList al = new ArrayList();
al.Add(10);   //0
al.Add("Raj"); //1
al.Add(5.10);  //2
foreach (var i in al){
    Console.WriteLine(i);
}}}
```

Arraylist Example →

→ To remove 'Raj'

al.Remove("Raj"); al.RemoveAt(2);

al.Clear(); → To remove all items

al.Sort(); → for shorting

# Syntax: List<type> varname = new List<type>();
varName.Add(value);

Date: .....
Page: .....

List<T> (Generic)

→ It is dynamic array that can store elements of any datatype. 'T'. It allows adding, removing and accessing elements by index.

→
```
using System;
using System.Collections.Generic;
class List {
    static void Main() {
    List<int> num = new List<int>(); // for string
        num.Add(1);   List<string> str = new List<string>();
        num.Add(2);       Str.Add("Raj");
        num.Add(3);       Str.Add("Ram");
    foreach (int n in num) {     foreach (var s in str) {
        Console.WriteLine(n);        C.W(s);
    }                            }
// simply you can write
    var names = new List<string>() {"Raj", "Ram"};
    foreach (var name in names) {
        Console.WriteLine(name);
    }
```

- num.Remove(2); // remove an item
- num.RemoveAt(1); // remove 2nd item
- num.clear(); // remove all items
- num.Sort(); // arrange in ascending order
- num.Insert(0,0); // insert item at index 0

→ console.Write(num.Count); // returns total item

→ console.write(num[0]); // returns item at '0' index

Date :

Page :

* Dictionary<TKey, TValue> :

→ It represents a collection of key-value pairs where each key is unique. It allows fast lookup and retrieval of elements based on keys.

→
```
using System;
using System.Collection.Collections.Generic;
class Program {
    static void Main() {
Dictionary<int, string> d = new Dictionary<int, string>();
d.Add(1, "Raj");
d.Add(2, "Ram");
d.Add(3, "Hari");
foreach (var kv in d) {
    Console.WriteLine(kv.Key + " : " + kv.Value);
}}}
```

→
```
for(int i = 1; i <= d.Count; i++) {
    Console.WriteLine(i + " : " + d[i]);
}
```

* Stack<T>

→ Values are kept in stack using LIFO (last in first out).

- It offers Push(), Pop() & Peek() method to add & retrive values.
- It allows adding elements to the top of the stack and removing from top of the stack.

Date: .....

Page: .....

→ using System;

using System.Collections.Generic;

class Program {

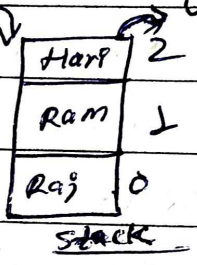
Static void Main() {

Stack <string> s = new stack <string>( );

s.Push ("Raj");

s.Push ("Ram");

s.Push ("Hari");



foreach (var name in s) {

Console.WriteLine (name);

}

- s.Pop( ); // remove last inserted element "Hari"
- s.Peek( ); // peak at the top element without removing it

→ Console.Write (s.Pop( )); // Remove & returns "Hari"

→ Console.Write (s.Peek( )); // Returns "Hari" without removing

✗ Queue <T>

→ Values are kept in queue in FIFO order.

- It offers Enqueue( ) and Dequeue( ) methods to add and remove values from collection.
- It allows adding element to the end of the queue and removing from beginning of the queue.

→ Queue <string> q = new Queue <string>( );

- q.Enqueue( ); // insert item in FIFO order
- q.Dequeue( ); // Remove 1st inserted element "Raj".

→ var evenNO = Num.FindAll (x => x % 2 == 0); Date: .....

Page: .....

* Program to find even number from the list

Instead of all these you can use LINQ as

```
using System;
using System.Collections.Generic;
  class Program {
      static void Main() {
List<int> num = new List<int>{1, 2, 3, 4, 5};
List<int> evenNO = new List<int>();
// Iterate over the list & filter even numbers
foreach (var n in num) {
    if (n % 2 == 0) {
        evenNO.Add(n);
    }
}
// Display the result
foreach(var i in evenNO) {
    Console.WriteLine(i);      //O/P: 2, 4
}
}
```

* By using array

```
using System;
class Program {
    static void Main() {
int[] num = {1, 2, 3, 4, 5};
foreach (var n in num) {
    if (n % 2 == 0) {
        Console.WriteLine(n);
}
}
}
```

Advance C#

→ improve execution timing

Date: .....

Page .....

* Delegates (holds a reference to a method) → like function pointer in C

- A delegate is a type that represents references to methods with a particular parameter list and return type.
- To associate a delegate with particular method, the method must have the same return type and parameter lists as that of the Delegate.
- When you instantiate a delegate, you can associate its instance with any method with a compatible (parameter) signature and return type.
- You can invoke (or call) the method through the delegate instance → you can use Invoke() method
- Delegates are used to encapsulates methods.

Ex:

```
using System;
public delegate void Calculator(int a, int b);   → signature
                                                 → doesn't have body of {}
class Program {
public static void Addition (int a, int b){
    int result = a+b;
    Console.WriteLine("Addition: " + result);
}
static void Main() {
    Calculator c = new Calculator(Program.Addition);
    c(10,20); or c.Invoke(10,20);
}}
```

(its reference is passed to obj of delegate)

(is assigned)

This is style cast delegate because it point to single method at a time.

→ When delegates point to more than one function at a time then it is called multicast delegate

→ Assignment operator (+= & -=) are used to implement it.

Date: .....

Page: .....

Eg 2:

```
using system;
public delegate void Calculator (int a, int b);
class Program {
    public static void Add (int a, int b) {
    console.WriteLine("sum:" +(a+b)); //sum: 30
    }    or ($"sum: {a} + {b} = {a+b}"); //sum: 10+20=30
public static void sub(int a, int b) {
console.writeline("Difference:" +(a-b));
}
public static void Mul (int a, int b) {
 console.writeLine("product:" +(a*b));
}
public static void Division (int a, int b)
console.writeline("Division:" +(a/b);
}
  static void Main() {
  Calculator obj = new Calculator (Program.Add);
  obj(10,20);
  obj = sub;
  obj(20,10);
  obj = Mul;
  obj(10,20);
  obj = Div;
  obj(20,10);
```

```
obj += sub;
obj += Mul;
obj -= Div;
obj(20,10);
```

(+=) operator combine delegate instance whereas (-=) operator remove the delegate.

or,

```
Calculator obj3 = new Calculator (Program.Div);
obj3(20,10);
}
}
```

→ Delegates pointing method without name
→ (function without a name)
Anonymous Delegates (function/method)

Date: .....
Page: .....

Eg: 3
```
using system;
public delegate void Calculator(int a, int b);
class Program {
    static void Main() {
        Calculator cal = delegate (int x, int y) {
            // Inline content of the method;
            Console.WriteLine(x+y);
        };
        cal(10,20);
    }
}
```

Function => Delegate => Event

* Events i.e. delegate depends on function & events depend on delegate
→ The event is a notification that depends on delegates and cannot be created without delegates.
• Event is like a wrapper over the delegate to improve its security
• Events enable a class or object to notify other classes or objects when something of interest occurs.
• The class that sends (or raises) the event is called publisher and the classes that receive (or handle) the event are called subscribers.
• An event can be declared in two steps:
i) Declare a delegate
ii) Declare a variable of the delegate with **event** keyword.

Syntax
```
public delegate void Notify(); // delegate
public event Notify MyEvents; // event
```

Date : .....
Page : .....

Eg:

```
using System;
public delegate void EventHandler();
class program
{
    public static event EventHandler add;
static void fname(){
    Console.WriteLine("Raj");
}
static void lname(){
    Console.WriteLine("Kumar");
}
    static void Main(){
    add += fname;  or, add += new EventHandler(fname);
    add += lname;
    add();          or, add.Invoke();
}
}
```

→ introduced by C# 3.0

* Lambda Expression (shorthand)

⇒ They are a concise way to represent anonymous methods. Here, you don't need to specify the type of the value that you input thus making more flexible.

- '=>' → lambda operator used in all lambda expression
- The lambda expression is divided into two parts :
  - Left side → Input
  - Right side → Expression

Date: .....

Page: .....

- Lamda Expression can be of two types

1) Expression Lambda : consists of the input and the expression.

Syntax: input => expression ;

Eg→
```
using System;
public delegate int MyDelegate (int a, int b);
class program {
    static void Main ( ) {
    MyDelegate obj = (a,b) => a+b;
    Console.WriteLine(obj(5));        // OP: 25
                          ↳ (10,20)
    }}
```

2) Statement Lambda : consists of the input and a set of statements to be executed.

Syntax : input => { statement };

→
```
using System;
public delegate int Calculator (int a, int b);
class program {
    static void Main() {
      Calculator cal = (a, b) => {
      Console.WriteLine(a+b);  → return a+b;
    };
      cal(10,20);          → Console.Write(cal(10,20));
    }}
```

* Func delegate → includes built in generic data types Func & action so you don't need to define custom delegate manually.

Eg→
```
using system;
class program {
    static void Main() {
    Func<int, int, int> add = (a,b) => a+b;
    Console.WriteLine(add(10,20));
    }}
```
→ it is also examples of expression lambda

→ An exception is an event that disrupts the normal flow of a program's execution due to an error or unexpected condition.

Date: .....

Page: .....

* Exception Handling

→ An exception is a problem that arises during the execution of a program.

→ Exception handling allows to gracefully handle runtime errors that occur during program execution.

- C# provides 'try', 'catch', 'finally' & 'throw' keyword to implement exception handling

- try → identifies a block of code for which particular exception is activated.
- catch → handle the exception
- finally → execute code with or without exception
- throw → throws an exception when a problem shows up.

Eg →

```
using System;
class Program {
    static void Main() {
        int a=10, b=0, c;
        try {
            c = a/b;
            Console.WriteLine(c);
        } catch (Exception e) {
            Console.WriteLine("An error occured: " + e.Message);
        } finally {
            Console.WriteLine("Finally always run!");
        }
```

O/P: An error occured: Attempted to divide by zero.
Finally always run!

Note: If exception occur, catch block will execute if not, try block will execute.

Date: .....
Page: .....

Eg 2

→
```
using system;
class Program{
    static void Main(){
    try{
     int[] a = {1, 2, 3, 4};
    console.writeLine (a[4]);
  } catch (Exception ex){
     Console.WriteLine("An error occured : "+ex.Message);
},};
```

O/p: An error occured: Index was outside the bounds of the array.

Eg 3:

→
```
using system;
class program{
     static void checkAge(int age){
    if(age < 18){
    throw new Exception("not eligible to vote");
   }
   else{
   Console.WriteLine("Eligible to vote");
};}
   static void Main(){
       checkAge(15);
   }
}
```

O/p: Unhandled exception. System.Exception : not eligible to vote at program.checkAge(Int32 age)

Language-Integrated Query

Date: .....

Page: .....

* Introduction to LINQ

- It is a query to retrive data from various resources and format.
- LINQ is a structured query syntax built in c# to retrive data from different data sources using a syntax similar to SQL queries.
- Data sources include object collection, ADO.Net Dataset, XML Docs, SQL Database and more.

* Steps for writing the query syntax

1) use System.Linq namespace
2) create the data source on which to perform operation

ArrayList → Eg: string[] names = { "Raj", "Ram", "Hari", "shiv" };

List<> → List<string> names = new List<string>() {
"Raj", "Ram", "Hari", "shiv" };

3) create query for the data source using keyword like select, from, where, orderby, groupby, Join, Aggregate, etc.

```
Eg: var res = from name in names
              where name.Contains("a")
              select name;
```

4) Execute the query using for each loop

```
Eg: foreach(var name in res){
        console.WriteLine(name);
    }
```

* Instead of query syntax we can use lambda expression (known as method syntax)

```
Eg: var res = names.Where(s => s.Contains("a"));
OR (s => s.Contains("Raj") || s.Contains("Ram"));
```

Date: .....

Page: .....

Eg:- program to find even no. from the list

→
```
using System;
using System.Collection.Generic;
using System.Linq;
class Program {
    static void Main() {
// Data source
List<int> num = new List<int>() {1,2,3,4,5,6};
// LINQ Query
var res = from n in num
          where n%2 == 0
          select n;
OR // LINQ Method (Lambda Expression)
// var res = num.FindAll(x => x%2 == 0);
// Query Execution
foreach(var i in res) {
    Console.WriteLine(i);
}}}
```

LINQ operators

- Filtering → Where
- Sorting → OrderBy, OrderByDescending, Reverse
- Grouping → GroupBy
- Projection → Select
- Aggregation → Aggregate, Average, Count, Max, Min, Sum
- Quantifiers → Contains, All, Any
- Join → Join

Object list example

2020

Q) WAP to select employees whose salary is greater than 20000 and address is Kathmandu using LINQ.

Date:
Page:

```
using System;
using System.Collections.Generic;
using System.Linq;
class Employee{
  public String name{get; set;}
  public string Address{get; set;}
  public double Salary{get; set;}
}
class program{
        Static void Main(){
  List<Employee> list = new List<employee>(){
new Employee{name="Raj", Address="ktm", salary=40000},
new Employee{name="Ram", Address="Dmk", salary=5000},
new Employee{name="Hari", Address="ktm", salary=50000},
  };
var result = list.Where(s=> s.Salary > 20000 &&
// s.Address.Equals("ktm")        s.Address=="ktm");
Console.WriteLine("Name \t Address \t salary");
   foreach(var i in result){
Console.Write(i.name + "\t"+i.Address+"\t"+i.salary);
}}}
                    //using LINQ query
                    var result = from l in list
```

O/p:

Name	Address	Salary
Raj	ktm	40000
Hari	ktm	50000

Where l.Salary > 20000
&& l.Address == "ktm"
orderby l.Name
select l;

Also, to order by name in ascending order

```
var result = list.Where(s=>s.Salary>20000 && s.Add=="ktm").OrderBy(s=>s.Name);
```

- OrderBy → ascending order
- OrderByDescending → descending order

Date: .....
Page: .....

* using OrderBy

```
var result = list.orderBy(s => s.salary);
// other code same as before.
```

* using GroupBy

```
var result = list.GroupBy(s => s.Address);
foreach (var item in result){
    Console.WriteLine("Group Key : " + item.Key);
Console.WriteLine("Name \t Address \t salary");
foreach (var i in item){
Console.Write(i.Name + "\t" + i.Address + "\t" + i.salary);
}}
```

o/p: Group key: ktm

Name	Address	salary
Raj	ktm	40000
Hari	ktm	50000

Group key: Dmk

Name	Address	salary
Ram	Dmk	5000

* Joining multiple lists - (concat, union, join)

```
using System;
using System.Collections.Generic;
using System.Linq;
class Program{
    static void Main(){
List<string> str1 = new List<string>(){"Ram", "shyam"};
List<string> str2 = new List<string>(){"Hari", "Bopal"};
```

- using concat → `var result = str1.Concat(str2);` "Ram"

Using Union

```
// var result = str1.union(str2);
```

Date: .....
Page: .....

```
foreach (var i in result){
    Console.WriteLine(i);
}
```

Using Join

```
var result = str1.Join(str2,
                s1 => s1,
                s2 => s2,
                (s1, s2) => s1);
```

o/p : Ram

→

```
using System;
class Program {
    Static void Main(){
object[] array = {"Hello", "Hi", 12345};
var result = string.Join(",", array);
Console.WriteLine(result);
}}
```

4 "/"
"_"

o/p : Hello, Hi, 12345

* Aggregate function

→

```
using System;
using System.Collections.Generic;
using System.Linq;
class Program {
    Static void Main() {
List<int> marks = new List<int>() {10, 20, 30, 40, 50};
```

Date : .....
Page: .....

```
int max = marks.Max();
int min = marks.Min();
int sum = marks.Sum();
int total = marks.Count();
double avg = marks.Average();

Console.WriteLine("Max : " + max);
Console.WriteLine("Min : " + min);
Console.WriteLine("sum : " + sum);
Console.WriteLine("Total count : " + total);
Console.WriteLine("Avg : " + avg);
}}
```

Q) WAP to join the list of employees & departments and retrieve & the name of employees along with corresponding department names.

→

```
using System;
using System.Linq;
using System.Collections.Generic;
class Employee {
public string EName{get;set;}
public int EID{get;set;}
public int DID{get;set;}
}
public class Department {
    public int DID{get;set;}
    public string DName{get;set;}
}
```

class program {
static void main() {
List<Employee> employees = new List<Employee> {
new Employee {EID = 1, EName = "Raj", DID = 101},
new Employee {EID = 2, EName = "Hari", DID = 102},
new Employee {EID = 3, EName = "Ram", DID = 102}
};
List<Department> departments = new List<Department> {
new Department {DID = 101, DName = "Management"},
new Department {DID = 102, DName = "science"}
};
var result = employees.Join(departments,
emp => emp.DID,
dep => dep.DID,
(emp, dep) => new {emp.EName, dep.DName}
);
foreach (var res in result) {
Console.WriteLine(res.EName + " " + res.DName);
}}}

OR var result = from emp in employees
join dep in departments
on emp.DID equals dep.DID
select new {emp.EName, dep.DName};

* To find/select the person having maximum age

var maxAge = persons.Max(x => x.Age);
var result = persons.Where(x => x.Age == maxAge);
OR// from p in persons where p.Age == maxAge select p;

* Group a product by category & find average price of each

var result = products.GroupBy(x => x.category).select(x => new {
category = x.Key,
price = x.Average(s => s.price)
});

Date: .....
Page: .....

Working with Databases

Requirement → Express edition → SQL Server Mgmt Studio

- Download SQL Server & SSMS and install them (Basic)
- Open SSMS & change the server name as ".\SQLEXPRESS" or "localhost\SQLEXPRESS"
- Click on new query & create a new database
  - eg: `Create database database_name` → select it & execute
- Choose the created database from dropdown menu [master]

* To create a table

```
create table tb_name (
    id int,
    name varchar(20),
    address varchar(20),
    phone bigint,
    dob date,
    .....
);
```

* To insert data into table

```
insert into tb_name (id, name, address) values (1, 'Raj', 'Ktm'), (2, 'Ram', 'ktm');
```

* To view data

```
select * from tb_name;
```

* To delete one data from table

```
delete from tb_name where <condition>;
```
→ id = 1 or name = 'raj'

* To delete all data from table

```
truncate table tb_name;
```

* To delete entire table

```
drop table tb_name;
```

- Data source ⇒ Server
- Initial Catalog ⇒ Database
- User ID
- password
- ⇒ Integrated security = True

Date: .....
Page: .....

* Working with connection & command

1) Working with connection

→
```
using System;
using System.Data.SqlClient;
class Program {
    static void Main() {
string path = "Data Source = .\\SQLExpress ; Initial Catalog = raj;
                Integrated Security = True";
SqlConnection con = new SqlConnection(path);
 try {
    con.Open();
    Console.WriteLine("Connection opened successfully!");
 } catch (Exception ex) {
    Console.WriteLine("Error : " + ex.Message);
}}
```

(servername ↑, db_name ↑)

↳ Error: cannot open database "raj" requested by login. The login failed. → If database name is incorrect

o/p: Connection opened successfully! → if database name is correct.

2) Working with command

→
```
using System;
using System.Data.SqlClient;
class Program{
    static void Main() {
string path = "Server = .\\SQLEXPRESS ; Database = raj; Integrated Security = True";
SqlConnection con = new SqlConnection(path);
con.Open();
string sql = "insert into data (id, name) values (1, 'Raj')";
SqlCommand cmd = new SqlCommand(sql, con);
```

```
int rows = cmd.ExecuteNonQuery();
Console.WriteLine($"Rows inserted : {rows}");
```

Date : .....

Page : .....

```
cmd.ExecuteNonQuery();
con.Close();
}}
```

* CRUD operation for SQL server

→

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
    public class student {
    public int id {get; set;}
    public string name{get; set;}
public student(int id, string name){
    this.id = id;
    this.name = name;
}}
class program {
    public static SqlConnection con;
    public static void create(on(){
String path = "Data Source = .\\SQLEXPRESS; Initial Catalog = college;
    Integrated Security = True";
con = new SqlConnection(path);
con.Open();
public static void insertData(int id, string name){
 create(on();
String sql = $"insert into student (id,name) values({id},'{name}')";
SqlComman cmd = new SqlCommand(sql,con);
 cmd.ExecuteNonQuery();
}con.Close();
```

Date : .....

Page : .....

```
public static void readData(){
    create(con);
    String sql = "select * from student";
    SqlCommand cmd = new SqlCommand(sql, con);
var SqlDataReader = cmd.ExecuteReader();
While (reader.Read()){
Console.WriteLine($"id={reader["id"]}, name={reader["name"]}");
}
con.Close();
}
public static void updateData(int id, string name){
Create(con);
String sql = $"update student set name='{name}' where
                id={id}";
SqlCommand cmd = new SqlCommand(sql, con);
cmd.ExecuteNonQuery();
}con.Close();
public static void deleteData(int id){
createCon();
String sql = $"delete from student where id={id}";
SqlCommand cmd = new SqlCommand(sql, con);
cmd.ExecuteNonQuery();
con.Close();
}
public static void Main(){
    List<student> list = new List<student>(){
    new student(1, "aaj"),
    new student(2, "ram")
    };
```

Date : .....
Page: .....

```
foreach (var item in list) {
    InsertData (item.id, item.name);
}
readData();
updateData(1, "Rajkumar");
readData();
deleteData(2);
readData();
}}
```

2020 11 6

WAP to show insert and select operation in database.

→

```
using System;
using System.Data.SqlClient;
class Program{
        static void Main() {
string path = "server=.\\SQLEXPRESS;Database=raj;Integrated Security=True";
SqlConnection con = new SqlConnection(path);
con.open();
// Insert operation
String InsertSql = "INSERT INTO data(id, name, age)
VALUES(1,'Raj',20),(2,'Ram','22')";
SqlCommand cmd = new SqlCommand(insertsql, con);
int rows = cmd.ExecuteNonQuery();
Console.WriteLine($"{rows} row inserted");
// Select operation
String selectSql = "SELECT * FROM data";
SqlCommand cmd = new SqlCommand(selectSql, con);
```

Date : .....
Page : .....

```
var reader = cmd.ExecuteReader();
Console.WriteLine("selected Data : ");
While (reader.Read()){
  Console.WriteLine($"Id : {reader["id"]}, Name:
    {reader["name"]}, Age : {reader["age"]} ");
}
con.close();
}
```

2022 11,

1* WAP to connect database Bank and insert 5 customer record (Account no, Name, Address, Balance) in customer table and display record whose balance > 5000.

```
// Insert operation
String sql = "INSERT INTO customer (account no, name, address,
balance) VALUES (001, 'Raj', 'Tpj', 6000), (002, 'Ram',
'Drn', 5000), (003, 'Hari', 'Btm', 2000),
(004, 'Shyam', 'Ktm', 10000), (005, 'Gopal', 'Brt', 8000)";
SqlCommand cmd = new SqlCommand(sql, con);
cmd.ExecuteNonQuery();
// Select operation
String sql1 = "SELECT * FROM customer WHERE balance > 5000";
SqlCommand cmd = new SqlCommand(sql1, con);
var reader = cmd.ExecuteReader();
While (reader.Read()){
Console.WriteLine($"Account no. : {reader["accountno"]}
Name : {reader["name"]}, Address : {reader["address"]}
Balance : {reader["balance"]} ");
}
```

Date : .....

Page : .....

ASP.NET (Active Server Pages Network Enabled Technologies)

→ ASP.NET is a server-side web application framework designed for web development to produce dynamic web pages.

- It was developed by microsoft to allow programmers to create dynamic websites in a more efficient way.
- It is mainly used because of its high speed, language-independent nature, and low cost. We can make ASP pages in any language so it is not dependent on a particular language.

Requirement

↳ Goto Tools ↳ Get Tools and features ↳ workloads & click on ASP.NET & web development ☑

↳ Goto Individual component next after workloads & click on ☑ .NET Framework project and item templates

↳ Then create a new project ↳ ASP.NET Web Application (.NET Framework) & Enter a name & location (optional) for your project ↳ create

↳ click on web forms ↳ create

↳ Right click on project name ↳ Add ↳ Web Form & give title (e.g. demo)

- demo.aspx → contain the markup (HTML) & server controls that defines UI of web page
- demo.aspx.cs → contain server side logic & event handlers for corresponding '.aspx'.
- demo.aspx.designer.cs (Not important)

- .aspx files: These are ASP.NET web forms pages. They contain markup (HTML) and server controls defining the UI of web page.
- .aspx.cs files: These files are code-behind files containing server-side logic and event handlers associated with UI elements.

They work together to seperate the presentation ('.aspx') from the application logic ('.aspx.cs'), enabling easier maintenance and development of dynamic web pages.

Date : .....
Page : .....

Eg:1, - creating a basic form

Click the submit button below!

[submit] ⇒ Button clicked!

[submit]

1) demo.aspx

→ <%@ Page Language = "C#" AutoEventWireup = "true"
codeBehind = "demo.aspx.cs" Inherits = "namespace.demo" %>
<!DOCTYPE html>
<html>
<head>
<title> Basic form </title>
</head>
<body>
<form id = "form1" runat = "server">
<asp: Label ID = "lbl" runat = "server" Text = "Click Button below!">
</asp:Label>

<asp: Button ID = "btn" runat = "server" Text = "Submit"
onClick = "Click" />
</form>
</body>
</html>

2. demo.aspx.cs

→ using System;
using System.Web.UI;
namespace ASP.NET {
public partial class demo : page {
protected void Click (object sender, EventArgs e) {
lbl.Text = "Button clicked!";
}}

Date : .....
Page: .....

Eg:2, Handling event (sum of two number)

- sum.aspx

→
```
<html>
<head>
<title> sum </title>
</head>
<body>
<form id = "form1" runat = "server">
<div>
<asp:Label runat = "server" Text = "1st num :"></asp:Label>
<asp:Textbox Id = "num1" runat = "server"></asp:Textbox><br/>
<asp:Label runat = "server" Text = "2nd num :"></asp:Label>
<asp:Textbox Id = "num2" runat = "server"></asp:Textbox><br/>
<asp:Label ID = "result" runat = "server" Text = "Result :">
</asp:Label><br/>
<asp:Button Id = "btn" runat = "server" Text = "Calculate"
onClick = "click" />
</div></form></body></html>
```

- sum.aspx.cs

→
```
protected void click(object sender, EventArgs e){
    int a = Convert.ToInt32(num1.Text);
    int b = Convert.ToInt32(num2.Text);
    int res = a + b;
    result.Text = "Result : " + res;
}
```

O/P:

1st num : [5]

2nd num : [5]

Result : 10

[Calculate]

Date: .....

Page: .....

2022

Eg 4. WAP to create form for calculating simple interest in one ASP.NET page and display the simple interest in another page of ASP.NET.

- Input.aspx

```
<form id="form1" runat="server">
<div>
<label for="principal">Principal :</label>
<asp:Textbox ID="principal" runat="server"></asp:Textbox><br>
<label for="time">Time:</label>
<asp:Textbox ID="time" runat="server"></asp:Textbox><br>
<label for="rate">Rate:</label>
<asp:Textbox ID="rate" runat="server"></asp:Textbox><br>
<asp:button ID="btn" runat="server" Text="Calculate"
    Onclick="click" />
</div>
</form>
```

- Input.aspx.cs

```
protected void click(object sender, EventArgs e){
decimal p = convert.ToDecimal(principal.Text);
decimal r = Convert.ToDecimal(rate.Text);
int t = Convert.ToInt32(time.Text);
decimal si = (p*t*r)/100;
Response.Redirect($"output.aspx?interest={si}");
}
```

- output.aspx

```
<form id="form1" runat="server">
<div>
  <asp:label ID="result" runat="server" Text=""></asp:label>
</div>
</form>
```

Date : .....

Page: .....

- output.aspx.cs

```
protected void Page_Load (object sender, EventArgs e) {
    string res = Request.QueryString["interest"];
    result.Text = "Simple Interest : " + res ;
}
```

* User Registration form with validation

- Registration.aspx

```
<form id = "form1" runat = "server" >
<div> <h2> User Registration Form </h2>
<label for = "username" > Usera name : </label>
<asp : TextBox ID = "username" runat = "server" ></asp:Textbox>
<asp : RequiredFieldvalidator runat = "server" ControlTo validate
= "username" ErrorMessage = "username is required" /> <br/>

<label for = "email" > Email : </ label>
<asp: TextBox ID = "email" runat = "server"></asp:Textbox>
<asp: RequredField validator runat = "server" controlTOvalidate
= "email" ErrorMessage = "email is required" /> <br/>
<asp: RegularExpressionValidator runat = "server" controlTo validate
= "email" Errormessage = "Invalid email format"
ValidationExpression = "[\w_\-\.]+@[a-z]{5,}\.[a-z]{2,3}"></br>

<label for = "password" > password : </label>
<asp: TextBox ID = "password" runat = "server" TextMode = "password">
</asp:TextBox>
<asp: RequiredField validator runat = "server" controlTo validate = "password"
Error message = "password is required." /> <br/>
<asp: RegularExpressionvalidator runat = "server" controlTo validate = "password"
```

- '^' → matches the start of line
- $ → matches the end of line

Date : .....

Page : .....

ErrorMessage = "Minimum 8 characters required." ValidationExpression = "^.{8,}$" />

<asp:Button ID = "btn" runat = "server" Text = "Register" />

OnClick = "Click" />

<asp:ValidationSummary runat = "server" ShowSummary = "true"

ForeColor = "Red" />

</form>

- Registration.aspx.cs

```
protected void Click (object sender, EventArgs e) {
    String username = username.Text;
    String Email = email.Text;
    String Password = password.Text;
Response.Redirect ($"Confirmation.aspx? username={username}
    &email={Email}&password={Password}");
}
```

- Confirmation.aspx

```
<form id="form1" runat="server">
<div>
    <h2> Registration confirmation </h2>
<asp:Label ID="result" runat="server" Text=""></asp:Label>
</div> </form>
```

- Confirmation.aspx.cs

```
protected void Page_Load (object sender, EventArgs e) {
    String username = Request.QueryString["username"];
    String Email = Request.QueryString["email"];
    String Password = Request.QueryString["password"];
result.Text = $"Username : {username} </br>
        Email : {email} <br/> password : {password}";
}
```

(form control)

Date : .....

Page : .....

* web server control in ASP.NET

→ web server controls in ASP.NET are server-side components that run on the web server and generate HTML output to be rendered in client's browser.

• These controls are categories as server and client based. some of them are:

1) Label : used to display text on HTML page

⇒ <asp : Label runat = "server" Text = "name : "></asp : Label>

2) TextBox : used to create a text input in the form

⇒ <asp : TextBox runat = "server" ID = "name"></asp : TextBox>

3) Button : used to create a clickable button to perform certain action when clicked.

↳ <asp : Button ID = "btn" runat = "server" Text = "submit" OnClick = "btn_Click" />

4) DropdownList : create dropdown list of options to select one.

<asp : DropDownList ID = "gender" runat = "server" >
<asp : ListItem Text = "Male"></asp : ListItem>
<asp : ListItem value = "female"></asp : ListItem>
</asp : DropDownList>

↳ string Gender = gender.SelectedValue; // to retrive data from control

5) RadioButton : Create radio button to select one option from group

<asp : RadioButtonList ID = "gender" runat = "server" >
<asp : ListItem value = "Male"></asp : ListItem>
<asp : ListItem value = "female"></asp : ListItem>
</asp : RadioButtonList>

6) CheckBox : create check box to select more than one options

<asp : CheckBoxList ID = "subject" runat = "server" >
</asp : CheckBoxList>

Date: .....
Page: .....

⇒ used to check whether the user inputs are valid or not.

Validation controls in ASP.NET

- RequireFieldValidator: Makes an input control a required field (mandatory).
- RegularExpressionValidator: Ensures that the value of an input matches a specified pattern. (use in email)

Eg: <asp:TextBox ID="name" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator runat="server" ControlToValidate="username" ErrorMessage="username is required." />
<asp:RegularExpressionValidator runat="server" ControlToValidate="name" ErrorMessage="Max length 10 characters" ValidationExpression="^.{0,10}$" />

- CompareValidator: compares the value of one input control to another. (use for password confirmation)

Eg: <asp:TextBox ID="psw" runat="server" TextMode="password" />
<asp:TextBox ID="confirmpsw" runat="server" TextMode="password" />
<asp:CompareValidator runat="server" ControlToValidate="confirmpsw" ControlToCompare="psw" ErrorMessage="password do not match." />

- RangeValidator: check if the value entered falls within a specific range.

Eg: <asp:TextBox ID="age" runat="server"></asp:TextBox>
<asp:RangeValidator runat="server" ControlToValidate="age" Type="Integer" MinimumValue="18" MaximumValue="100" ErrorMessage="Age must be between 18 to 100" />

- CustomValidator; use for complex validation not covered by oth.
- ValidationSummary: Displays a summary of all validation error

2nd <asp:ValidationSummary runat="server" ForeColor="red" />