* Introduction to .NET

* What is .NET ?

- .NET is a free, Open-source development plateform by microsoft used to build :
  - website
  - Desktop apps
  - Mobile apps
  - Games
  - Cloud apps
  - IoT apps.

- Variants:

Name	Description
.NET Framework	Original ~~Frame~~ Version (Windows-only) - used for legacy apps.
.NET Core	Lightweight, Cross-plateform
.NET 5+ (Unified)	Latest, unified plateform (include .NET6, .NET7, 8)

* .NET Architecture

- CLR (Common Language Runtime
  : Executes .NET Programs, handles memory, security, threds, etc

- CTS (Common Type system)
  : Ensures all .NET language use same data types

- CLS (common Language Specification)
  : Set of rules every .NET language must follow.

- FCL (Framework class library)
  : A big collection of prebuit libraries to use in your apps.

+ Visual Studio / VS code setup

~~• Tools:~~

1. setup using Visual studio (Full IDE)

- Download Visual studio
- Run the installer
  - Select ".NET desktop development" workl
  - Also select Asp.NET and web development for web apps
- Install the .NET SDK (if needed)
- Create your First project
  - open Visual Studio
  - click "Create a new project"
  - Select "Console App (.NET core or .NET6/7/8)
  - Name the project & click Create.
- Run your code

2. Setup using VS code

- Install VS code
- Install .NET SDK
  - Required for using CLI

- Install C# Extension in VS code
- create and Run project

Imp * Understanding MSIL and JIT

+ MSIL - Microsoft Intermediate Language.
  + code written in C# is compiled into MSIL, not machine code directly.
  + MSIL is plateform - independent

+ JIT - Just-In-Time Compiler
  + At runtime, MSIL is converted to machine code by JIT.
  - JIT makes sure your code runs on the actual OS/CPU.

C# code compiled → MSIL → JIT → Machine Code

* Cross-plateforms Support in .NET

- .NET core / .NET 5+ allows you to build apps for
  - Windows
  - Linux
  - MacOS

- Tools
  - CLI (command Line Interface)
  - Docker.

* Language Support In .NET

Language	Use case
C#	Most popular, general purpose
F#	Functional programming
VB.NET	Easy-to-redd syntax

* Updates before .NET 6 and After .NET 6

- Before .NET 6 Format of code is

```
using system;
namespace MyFirstApp
{
Public class program
  {
    Public Static Void Main(string[] args)
      {
      console.WriteLine("Hello, world");
      }
  }
}
```

But

- After .NET 6 or newer, Microsoft made it easier by hiding the boilerplate code like
  - namespace
  - class
  - main() method.

So this code

```
console.WriteLine("Hello, world");
```

- It actually understood by compiler as above old version format

+ Basic structure :

• Using system ;
- Its like importing a toolbox that has many useful functions.

- system is a namespace that contain built in .NET classes like Console, Math, DateTime.

Example:
• Console.WriteLing() is inside the system namespace

• Without using system; you would need to write System.Console.WriteLine (....)

• namespace MyFirstApp.
- A namespace is like a folder or container that groups related code together.

Example:

```
namespace school
{
    // All code here belong to
       "schoo" group
}
```

name of class.

- Public class program
  - A class is like blueprint for building programs.
  - It group code (Method variables) into one structure.

- Public static void Main (string[] args)
  - This is the main method - The entry poin of your program.
  - when you click Run, this is where execution starts.

* C# Programming Language

* Data Types and Variables

- int = int age = 25
- double = double price = 19.99;
- char = char grade = 'A';
- string = string name = "Aditya";
- bool = bool isstudent = true;
-

* Type Conversion

- Implicit Conversion : safe automatic conversion

Example:

```
int x=10
double y= x;
```

- Explicit Conversion : manual and sometimes unsafe

Example:

```
double pi = 3.14;
int intpi = (int)pi    //3 ans
```

* Operators

- Arithmetic : + - * / %
- Comparison : == != > < >= <=
- Logical : && || !

* Namespaces and assemblies.

- Namespace groups related classes, avoids name conflicts.

- Assembly is compiled code library (DLL or EXE).

- use using keyword to include name spaces.

* Control structures.

- If-Else
- Else If Ladder
- Switch
- Loops (for, while, do-while)
- pattern Matching

>>> for above control structure skip next 5 pages (page no' at top)

Before Moving next Read below

1. What is an Entry point?
   - An entry point where you program starts Running
   - In C#, this usually a method called Main()
   - The runtime looks for this Main() method to know where to begin execution.

2. Traditional Way (Before .NET 5):
   - your always had to write a class with a Main() method

Example:

```
using system;
namespace Myprogram
 {
   class program
     {
       static void Main(string[] args)
         {
           console.Writeline ("Hello, world");
         }
     }
 }
```

- Every program must have exactly one Main() method as the entry point
- If you have multiple Main() methods, you must tell compiler which one to use (using startup object in visual studio or /main compiler option)

3. Multiple Main() Methods in Multiple Classes

- you can have several classes, each with its own main() method.

- But only one can be the actual entry point when running the program
- To choose which to run:
  - In visual studio: project properties → Application tab → startup object → select desired class with Main()

  - Using command Line: use the /main compiler option.

4. Top-Level statements (Introduced in C# 9/.NET 5+)

- To reduce boilerplate code, Microsoft introduced top level statements.
- you can write you program without explicitly declaring a class or Main() method.

Example:

```
using system;
console.Writeline("Hello From top level");
```

- Behind the scens, the compiler Creates a main() method For you.

5. Limitations of Top Level statements.

- only one file in your project can contain top level statements.

- If multiple files have top-level statemen you get an error:

"Only one compilation unit can have top-level statements"

- Because the compiler generates Main() method for each file with top-level statements, it leads to multiple entry points which is not allowed.

6. Mixing Top-Level statements and multiple Main() Methods:

- If your project contains explicit Main() method in some classes and a file with top-level statements, you get errors like:

"Program has more than one entry point defined".

To avoid this:

- either use only top-level statements in single file and no explicit Main()
- or use explicit Main() method in classes and avoid top-level statements.

17. Recommended Approches for organizing code;

a) Using Top-Level statements (Modern style)
- Use top-level statements in only one file.
- call methods from other classes normaly
- Other files contain classes with regular methods, but no Main()

Example;

```
//Program.cs - top level
IFElse.checkNumber(10);
LoopExample.PrintNumbers();

// IFElse.cs
public static class IFElse
  {
    Public static void checkNumber(int num)
        {/*    */}
  }

// LoopExample.cs
public static class LoopExamples
    {
      Public static void PrintNumbers()
          {/*    */}
    }
```

b) using Explicit Main() Methods (classic style)

- No top-level statements anywhere
- Each class have its own Main()
- choose the startup object before running

* Control structures:

1: If-Else

- used for decision-making. It checks conditi and executes code blocks accordingly.

Syntax:

```
if (condition) {
        //true block
  } else if (condition2) {
        // else-if block
  }
  else
  {
        // False-block
  }
```

2. Switch

- Replace many if-else blocks when comparing a single variable against multiple values.

```
Program system;
namespace Switchdemo{
    class program{
        static void Main(string[] args){
            int day = 3;

        switch(day){
            case 1:
                C.W("Monday");
                break;
            case 2:
                C.W("Tuesday");
                break;
                :
            default:
                C.W("other Day");
                break;
            }
        }
    }
}
```

3. LOOPS: For, while, do-while, foreach

- Loops help run block code, repeatedly under certain conditions

For loop;

```
for(int i = 1; i<=5; i++){
        C.W("i="+i);
    }
```

Whileloop:

```
static void Main(string[] args) {
    int i=1;
    while (i<=5) {
        C.W(i);
        i++;
    }
```

do-While loop:

```
static void Main(string[] args) {
    int i=1;
    do {
        C.W(i);
        i++
    } while (i<=5);
}
```

For-each loop:

```
static void Main(string[] args){
    string[] name = {"Aditya", "Sam", "Neha"};

    For each(string name in names) {
        C.W("Nam: "+name);
    }
```

4: break, continue, goto

- break : exits a loop / switch
- continue : skips to next iteration
- goto : jumps to a label.

Code:

```
using system
namespace BreakContinueGoto {
    class program {
        static void Main(string[] args) {
            for(int i=1; i<=5; i++){
                if(i==2)
                    continue;  //skip 2
                if(i==4)
                    break;  //stop at 4
                CW("i=" +i);
            }

            goto Label;
            C.W("This Line wont be printed");

        Label:
            C.W("you jumped to this Label");
        }
    }
}
```

* Methods (Functions)

- A method is a block of code that perform s a task. It helps reduce code duplic- ation and makes programs moduler and reusable

1: Creating and Calling Methods.

- A method is defined using a return type, name, and optional parameters.

- It is called by using it name with paran theses : MethodName();

Code + Explanation :

```
using System
namespace MethodExample {
 class program {
    static void Main (string[] args) {
      GreetUser(); // Method call
    }

      //Method defination
      Static void GreetUser () {
          C.W("Hellow, Aditya");
      }
    }
}
```

2: Method parameters: ref, value, out, params

- Pass by value (Defaut)

- When you pass value to a method by value, a copy of the variable is passed.
- Any modification inside the method does not affect the original value.

Program:

```
using System;
class program
 {
static void ModifyValue(int x)
 {
    x= x +10;
    C.W("Inside Method : x= " +x );
 }

Staticvoid Main(string[] args)
    {
     int num = 20;
     C.W("Before Method : num= " +num);

     ModifyValue(num); // pass by Value

     C.W("After Method : num " +num);
    }
 }
```

output:

Before Method : num = 20
Inside Method : x = 30
After Method : num = 20

• pass by ref

- When you pass a value by refrence, you are passing the actual memory address (refrence)
- Any modification made inside the meth reflect back in the original variable.

Program:

```
using system;
class program
{
 Static void ModifyValue(ref int x)
    {
     x = x + 10;
     C.W("Inside Method : x = " + x);
    }

  Static void Main(string[] args) {

     int num = 20;
     C.W("Before Method num = " + num);

     ModifyValue(ref num); // Pass by ref

     C.W("After Method : num = " + num);
    }
}
```

Output:

Before Method : num = 20
Inside Method : x = 30
After Method : num = 30

- Out keyword
- used to return multiple values from method
- variable doesn't need to be initialized before passing
- It must be assigned inside the method before returning.

Program:

```
using system:
class program{
 static void GetDetails(Out string name,
                        Out int age)
    {
      name="Aditya"
      age=22
    }

Static void Main(string[] args)
  {
   string myName;
   int myAge;

   GetDetails (Out myName, Out myAge);

   C.W("Name: "+myName);
   C.W("Age: "+age);
  }
}
```

Output:
Name: Aditya
Age: 22

• params keyword

- Allows a method to accept variable number of arguments.
- All arguments are passed as a single array
- Must be last parameter in method signature

Program:

```
using System;
class program
{
  Static void printNumbers(Params int[] nums)
  {
    C.W ("you passed " + nums.Length + "numbers")
    Foreach(int n in nums)
      {
        C.W(n)
      }
  }

  Static void Main(String[] args)
    {
      PrintNumbers (10,20,30); // pass 3 arguments
      Print Numbers (1,2); // pass 2 arg
      print Numbers (); // pass 0 args.
    }
}
```

* Array and String

1: One-Dimensional (1D) Array

- A linear collection of elements of same type
- Fixed size

syntax:

```
int[] arr = new int[5]; // Declare
int[] arr2 = {1, 2, 3, 4, 5}    // Initialization
```

2: Two-Dimensional (2D) Array

- Like a matrix (rows x colums)

syntax:

```
int[,] matrix = new int[2,3] //2 rows 2col
int[,] matrix2 = { {1, 2}, {3, 4} };
```

3. Jagged Array

- Array of arrays
- Each row can have diff length

syntax

```
int[][] jagged = new[2][];
jagged[0] = new int[] {1, 2};
jagged[1] = new int[] {3, 4, 5};
```

- String Operations

```
string str="Hello Csharp";
Console.WriteLine(str.Length); //11
C.W                (str.Substring(6)); //Csharp
C.W                (str.IndexOF('C')); //6
C.W                (str.Replace("Csharp","World"));

string[]parts = str.Split(' '); //["Hello","Csharp"]
```

* Object-Oriented Programming (OOP)

1: Classes and Objects.

class : Blueprint / template for creating objects.

object : Instance of class (realworld entity).

2: Constructors.

- Constructor is special Method that auto called when
  - ~~Default~~
  - ~~Parameterized~~
  - ~~static~~
  - ~~copy~~

when an object of a class is created.

- It is used to initialized object data.

1. Default Constructor

- Constructor with no parameters.
- Automatically provided by C# if you dont create any constructor yourself.

```
using system:
namespace ConstructorDemo {
    class Car() {
        Public Car() {
            C.W("This is a default Constructor");
        }
    }
```

```
class program {
    Static void Main(string [] args) {
        Car myCar = new Car(); // constructor
                                  gets called.
    }
}
```

2. Parameterized Constructor

- Constructor with parameters to pass values while creating object.

Program:

```
using system;
namespace ConstructorDemo {
    class Student {
        String name;
        int age;

        Public Student(string n, int a) {
            name = n;
            age = a;
        }

        Public void Show() {
            C.W("Name : " + name + ", Age : " + age);
        }
    }

    class Program {
        Static void Main(string [] args) {
            Student S1 = new Student ("Aditya", 21);
            S1.show();
        }
    }
}
```

3. Copy Constructor.

- Creates a new object by copying another object.
- Not automatically provided in C#.

Program:

```
using system;
namespace ConstructorDemo {
    class Book {
        string title;
        public Book (string t) {
            title = t;
        }

        //copy constructor
        public Book (Book b) {
            title = b.title;
        }

        public void show() {
            C.W("Title: " + title);
        }
    }

    class program {
        static void Main (string[] args) {
            Book b1 = new Book ("C# Basics");
            Book b2 = new Book (b1); //copy constr

            b1.show();
            b2.show();
        }
    }
}
```

4. Static Constructor

- used to initialize static data.
- called once automatically any static or normal method runs.
- cannot take parameters.
- has no access modifier (like public, private)

Program:

```
using system;
namespace ConstructorDemo {
    class Company {
        static string CompanyName;

        static company () {
            companyName = "Microsoft";
            C.W ("Static Constructer called);
        }

        public static void Show() {
            C.W("Company: " + CompanyName);
        }
    }

    Class program {
        static void Main (string[] args) {
            company.show();
        }
    }
}.
```

Output:
static Constructor called.
company : Microsoft

5. Private Constructor
- prevents object creation from outside the class.
- Mostly used to singleton Design pattern.
- Can be accessed only inside the same class.

Program:

```
using System;
namespace ConstructorDemo{
    class Secret{
        private secret (){
            C.W ("private Constructor called.");
        }

        Public Static void ShowMessage(){
            Secret s = new Secret(); //can be cre-
            C.W ("Hello Private");      ated in class
        }
    }

    class program{
        static void Main (string[] args){
            //secret obj = newsecret( ); X error
            Secret.showMessage ();
        }
    }
}
```

3. 3. Inheritance (single, Multilevel)

- Inheritance means one class (child) can ~~to create an~~ use features of another class (parent).
- It is used for code reusability and helps create a hierchy of class.

1. Single Inheritance

- only one parent and one child.

Example:

Class Vehicle → parent

class car → child (Inherits From Vehicle)

Program:

```
Using System;
namespace InheritanceExample {
    class vehicle {
        public void start () {
            C.W("Vehicle started");
        }
    }
    class car : vehicle {
        public void Drive() {
            C.W ("car is driving");
        }
    }

    Class program {
        Static void Main(string[] args) {
            Car c= new car();
            C.Start();
            C.Drive();
        }
    }
}
```

Output:
Vehicle started.
Car is driving.

2. Multilevel Inheritance
Parent → child → Grandchild

Program:
```
using system;
namespace InheritanceExample {
    class Animal {
        public void Eat() {
            C.W ("Eating")
        }
    }

    Class Dog : Animal {
        public void Bark() {
            C.W ("Barking...");
        }
    }

    Class puppy : Dog {
        public void Weep() {
            C.W ("Weeping..");
        }
    }

    class program {
        static void Main(string[] args) {
            puppy p = new Puppy();
            p.Eat();
            p.Bark();
            p.Weep();
```

Output:
Eating...
Barking...
Weeping...

3. Hierarchical Inheritance
- One parent multiple child.

Programe:

```
using system:
namespace InheritanceExample {
    class shape {
        public void showShape () {
            C.W("This is a shape");
        }
    }

    class circle :Shape {
        public void DrawCircle() {
            C.W("Drawing circle");
        }
    }

    class square: shape {
        public Void Draw square() {
            C.W("Drawing square");
        }
    }

    class program {
```

```
static void Main(string[] args) {
    Circle c= new Circle();
    Square s= new Square();

    c.showShape(); c. DrawCircle;
    s. showshape; s.drawsquare();
    }
  }
}.
```

Output:

This is a shape
Drawing Circle

This is a shape
Drawing square.

4. Multiple Inheritance (X Not support directly)

- C# does Not support multiple inheritance using classes to avoid confusion.

4. Polymorphism

- " one thing - many forms
- it means same method name behaves differently in diffrent classes or situation.

Type	How it works	Also Called as
1. Compile-time	Method overloading	static polym
2. Run time	Method overriding	Dynamic poly

- Method Overloading (Compile-Time poly)
- same methodname
- Different parameters (number or type)
- Done in the same class.

Program:

```
using system
namespace polymorphismDemo {
    class Calculator {
```

```
public int Add (int a, int b) {
        return a+b;
    }

Public double Add (double a, double b) {
        return a+b;
    }

Pubic int Add (int a, int b, int c) {
        return a+b+c;
    }
}

class program {
    static void Main (string[] args) {
        calculator calc = new Calculator ();
        C.W (calc.Add (5, 3));
        C.W (calc.Add (4.5, 2.5));
        C.W (calc.Add (1, 2, 3));
    }
}
}
```

Output:

8
7
6

Explanation: Same method name Add works in different way based on parameters - this is method overloading

- Method overriding (Run Time polymorphism)
- happens in inheritance
- child class overrides method of parent class using override keyword.
- parent should be marked as virtual

Program :

```
using System;
namespace polymorphismDemo{
    class Animal {
        Public virtual void speak () {
            C.W ("Animal speaks");
        }
    }.

    class Dog : Animal {
        public Override void speak () {
            C.W ("Dog barks");
        }
    }

    class Cat : Animal {
        public Override void speak() {
            C.W ("Cat meows");
        }
    }

    class program {
        Static void Main (string[] args) {
            Animal a1 = new Dog ();
```

```
Animal a2 = new Cat();

    a1.Speak();
    a2.Speak();
    }
  }
}
```

Output
Dog barks.
Cat meows.

Explanation:
- Method Speak is defined in Animal, but overriden in Dog and Cat
- Based on object type, right method gets called at runtime.
- This is method overriding (run-time)

5. Abstraction in C#
- Abstraction means hiding the internal details and showing only important features.
- It helps us focus on what an object does, not how it does it

Real life example:
- ~~you pr~~ When you drive a car:-
  - you press the break - you dont care how it works inside.
  - You just know it stop the car.

In C#, Abstraction is achieved in 2 ways:

Type	keyword	Description
1. Abstract class	abstract	can have abstract + normal methods
2. Interface	interface	Fully abstract (only method declaration)

1. Abstract Class
- A class marked with abstract keyword.
- Cannot be directly instantiated (you can't create object from it).
- Can contain:
  - Abstract methods (no body)
  - Non-abstract methods (with body)

Syntax:

```
abstract class ClassName{
    public abstract void AbstractMethod();
    public void NormalMethod() { // has body
        // code
    }
}
```

Abstract Method
- method without body
- declared using abstract keyword
- must be implemented in child class.

Program: 1.

```
using System;
namespace AbstractionExample{
  //step1: Create abstract class
  abstract class Animal {
     public abstract void MakeSound();

     Public Void Eat(){
        C.W("This animal eats food");
     }
  }

  //step2: Create child class that implements
  the abstract method
   Class Dog : Animal {
     public override void Makesound() {
        c.W("Dog barks");
     }
   }
//nobody
   class program {
     static void Main(string[] args) {
       Dog d= new Dog(); //create obj of child
       d.Makesound(); // overridden method
       d.Eat(); // inherited normal method
     }
   }
}
```

Output: Dog barks;
This is animal eat food.

Program 2:

```
using system;

abstract class Shape {
        public abstract void Draw();
  }

class Circle : Shape {
    public override void Draw() {
        Console.Writeline("Drawing a circle");
        }

class square : shape {
    public override void Draw() {
        C.W ("Drawing a Square");
    }
}

class program {
    static void Main(string[] args) {
        Shape S1 = new Circle();
        shape S2 = new Square();

        S1.Draw();
        S2.Draw();
        }
    }
```

2. Interfaces in C#

- Interface is like a blueprint for class
- It only contains method declarations, not implementation
- A class that implements the interface must provide code for all its methods.

Why Interface?

- To achieve 100% abstraction
- To support multiple inheritance
- For loose coupling and clean architecture

Syntax :

```
interface IInterfaceName {
    void Method();  //No body
}
```

- Interfaces typically start with (eg. IDevice, IShape).
- A class use Interfacename to implement

Program:

```
using System;
//Step 1 : Create Interface
interface IAnimal {
        void MakeSound();  // method without body.
}
```

```
//Step 2: Implement Interface
class Dog : IAnimal {
    public void MakeSound () {
        C.W ("Dog barks.");
    }
}

class Cat : IAnimal {
    public void Makesound () {
        C.W ("cat meows");
    }
}

Class program {
    static void Main (string[] args) {
        IAnimal dog = new Dog ();
        IAnimal cat = new cat ();

        dog.MakeSound();
        cat.MakeSound ();
    }
}.
```

Output:

Dog barks.
Cat meows.

* Explicit Interface.

Program :-

```
namespace Idem
{
  internal class program
    {
      static void Main(string[] args)
        {
          IDemo i1 = new Demo();
          i1.display();
          IDemo i2 = new Demo();
          i2.display();
        }
    }.

  interface IDemo
    {
      void display();
    }
  interface IDemo1
    {
      void display();
    }

  class Demo : IDemo, IDemo1.
    {
  public void ~~display()~~ IDemo.display(){
        C.W("i am in display method");
    }
```

```
void IDemo1.display ()
    {
      Console.WriteLine("i am in display
                        method of IDemo1");
    }
}
```

* Partial Interface

- A partial interface allows us to split the defination of an interface across the multiple files an multiple parts in the same file using the partial keyword

- This is ~~you~~use when;
  - interface is too large
  - you want to organize code better
  - you want to work in a team, and each developem can work on different parts of same interface.

Program: (File1)

```
namespace partialdemo;
{
 internal class Program
    {
      static void Main(string[ ] args)
      {
        student S1=new Student();
        S1.getdata();
        S1.putdata();
      }
    }
```

```
partial classstudent
  {
int rno;
string name;
int marks;
Public void getdata()
   {
    C.W("EnterNO");
    rno= Convert.ToInt32(C.R());
    C.W("Entername");
    name = C.R();
    C.W("Enter marks");
    marks = Convert.ToInt32(C.R());
   }
 }
}
```

(File 2)

```
internal partial class Student
  {
   Public void putdata()
    {
     C.W("rno is {0}", rno);
     C.W("name is {0}, name);
     C.W("marks is {0}, marks);
    }
  }
```

5. Sealed class

- A class that cannot be inherited is known as sealed class.
- It is declared with sealed keyword.
- you can create object of sealed class and access its members.

6. Encapsulation in C#

- Encapsulation means:
- Wrapping up data (variables) and code (methods) into a single unit (class)
- protecting the internal state of an object from outside interference.
- Give access to data only through public methods or properties.

- Access Modifiers

	Accessible Within	Used for
public	Everywhere	open access
private	Inside the same class	Hide internal data
protected	same class + child class	partial access (inherit)
internal	within same project	used for components of app