Dot Net

Date
DELTA Pg No 1

Week-1 Introduction to .Net Core and Fundamentals

Day 1 Overview of .Net Core Ecosystem Topics:

Introduction to .Net ~~Code~~ core!

.Net is a free, cross-platform, open source developer platform for building many kinds of applications. It can run programs written in ~~many~~ multiple languages, with C# being the most popular. It relies on high-performance runtime that is used in production by many high-scale apps.

The .Net platform has been designed to deliver productivity, performance, security and reliability. It provides automatic memory management via a ~~gc~~ garbage collector. It is type-safe and memory-safe due to using a CLR and strict language compilers. It offers concurrency via async/await and Task primitive. It include large set of library that have broad functionality and have been optimized for performance on multiple operating systems and chip architectures.

.Net Core is an open source, cross platform framework for building applications. it is a modern alternative to the traditional .NET framework.

.Net has the following design points:

- Productivity is full stack with runtime, libraries, language, and tools all contributing to developer user experience.
- Safe code is the primary compute model, while unsafe code enables additional manual optimization.
- Static and dynamic code are both supported, enabling a broad set of distinct scenarios.
- Native code interop and hardware intrinsics are low cost and high-fidelity (raw API and instruction access.)
- Code is portable across platforms (OS and chip architecture), while platform targetting enables specialization and optimization.
- Adaptibility across programming domains (cloud, client, gaming) is enabled with specialized implementations of the general-purpose programing model.
- Industry standards like open Telemetry and gRPC are favored over bespoke solutions.

.NET is maintained by Microsof and the community. It is regularly updated to ensure users deploy secure and reliable applications to production.

Components

.NET includes the following components.

- Runtime — executes application code.
- Libraries — provides utility functionality like JSON parsing.
- Compilers — compiles C# (and other languages) source code into (runtime) executable code.
- SDK and other tools - enable building and monitoring apps with modern workflows.
- App stacks - like ASP.NET core and windows forms, that enable writting apps.

Free and open source

.NET is free, open source, and is a .NET foundation project. .NET is maintained by Microsoft and the community on GitHub in several repositories.

Support

.NET is supported by multiple organizations that work to ensure that .NET run on multiple operating systems and is kept up to date.

New versions of .Net are released annually in November.

.Net Ecosystem

There are multiple variants of .NET, each supporting a different type of app. The reason for multiple variant is part historical, part technical.

.NET implementations:

- .NETFrameWork - The original .NET. It provides access to the broad capabilities of windows and window server. It is actively supported, in maintainance
- Mono - The original community and open source .Net. A cross-platform implementation of .NET framework. Actively supported for Android, ios, and WebAssembly.
- .NET (Core) - Modern .Net. A cross platform and opensource implementation of .NET, rethrough for the cloud age while remaining significantly compatible with .NET FrameWork.

Why use .NET Core?

- Cross-platform (windows, macOS, Linux)
- High performance and scalibility.
- Microservices and cloud friendly
- Open-source with strong community support.

Difference b/w .NET framework & .NET Core

Basic Comparision	.NETFramework	.NET Core
Platform or framework	.NET framework is a full-fledge development framework. The framework provides all the basic requisements for the development of applications - UI, DB Connectivity, services, APIs etc	.NET Core is a platform on top of which there are framework such as ASP.NET core and Universal Window platform that leverage and extend the feature of .Net core.
Opensource	.NET framework is not open-source It is more of a community-developed software project	.Net core is open-source
cross-platform	Although .Net framework was designed to develop software and applications for all the operating systems, yet it ended up favouring windows. Thus .Net framework is majorly used to develop windows-	.Net core follows the principle of build once run anywhere. Thus it is cross platform

Types of Application	.NET framework is used to build both Desktop as well as web applications. windows forms and WPF appn are very well supported.	.Net core focuses more on web, windows mobile and windows store appln.
Packaging and Shipping	.NET framework is packaged as a whole. All the libraries are bundled together and shipped together. Even if you do not require any library for your appln. it still comes as a part of the packages.	.Net core is delivered as a set of Nuget pacages. It has been factored, modularised and shipped as several Nugets packages. Although comman runtime library are still a part of the bundle the developer has the freedom to selectivity include other libraries as per need. This make .NET core very lightweight. No extra baggage.
Support for microservices	.NET framework doesn't support the creation and deployment of microservices in different languages.	.Net Core has support for microservices. .NET core allows a mix of technologies that can be minimalized for each microservice.

Support for WCF/REST services	.NET framework is an excellent choise when WCF services are involved. It also supports REST services.	.NETCore has no support for WCF services. You would always need to create a REST API.
CLI Tools	.NET framework is too heavy for Command line Interface. Some developers prefer working on CLI rather than an IDE.	.NET core supports a very lightweight CLI for all platforms. There is always an option to switch to an IDE as well.

Terminal Comonand :-

```
dotnet new Console -o himanshu
dotnet run
```

Output Hello World!

.NET CLI Overview.

The .NET Command line interface (CLI) is a cross-platform toolchain for developing, building, running and publishing .NET applications.

The .NET CLI is included with the .NET SDK. After installing the SDK, you can run CLI commands by opening a terminal and entering the commands at the terminal prompt.

Command structure

CLI command structure consist of the driver ("dotnet"), the command, and possibly command arguments and options. you see this pattern is most CLI operations, such as creating a new console app, and running it from the command line. The following commands show when the console app was run from a directory named my-app:

```
dotnet new console
dotnet build --output ./build-output
dotnet ./build-output/my-app.dll
```

Dsiver

The driver is named dotnet and has two responsibilities, either running a framework-dependent app or executing a command

To run framework dependent app, specify the path to the app's .dll file after the driver without specifying a command, for example, `dotnet ./path/to/my-app.dll`. When executing the command from the folder where the app's DLL resides, just execute `dotnet my-app.dll`. ~~For more information, see the dotnet command.~~

When you supply a command to the driver, dotnet.exe starts the CLI command execution process.

For Example:-

```
dotnet build
```

First the driver determines the version of the SDK to use. If there's no global.json file, the latest version of the SDK available is used. After the SDK version is determined, it executes the command.

Command

The command performs an action. For example, dotnet build builds code. dotnet publish publishes code.

Arguments

The arguments you pass on the command line are the arguments to the command invoked or to options specified with the command. For example, when you execute dotnet publish my-app.csproj, the my app.csproj argument indicates the project to publish and is passed to the publish command.

Options

The options you pass on the command lines are the options to the command invoked. for example when you execute dotnet publish --output /build-output, the --output option and its value provided by the /build-output argument are passed to the publish command.

CLI Commands
The following commands are installed by default

Basic commands
- new • restore • build • publish • run • test
- vstest • pack • migrate • clean • sln • help
- store • watch • format

Project modification commands
- add package • add reference • remove package
- remove reference • list package • list reference

Nuget Commands
- nuget delete • nuget locals • nuget push
- nuget add source • nuget disable source
- nuget enable source • nuget list source
- nuget remove source • nuget update source
- nuget verify • nuget trust
- nuget sign • nuget search
- nuget why

Work load management Commands
- workload • workload config • workload install
- workload list • workload update
- workload restore • workload repair
- workload uninstall • workload search

Advanced Commands
- sdk check • msbuild • build-server • dev-certs
- dotnet install script

Toolmanagement Commands:
- tool install • tool list • tool update • tool restore
- tool run • tool uninstall • tool search

Day2: Introduction to C# Basics Topics!

Q what is C#? full form.
C# is a simple & powerfull object oriented programming language developed by "Microsoft" that runs on .NET framework.

It is type safe language
Ex:- int num = "abc";
error

Anders Hejlsberg is the developer of C# language.

The first version of C# was "C# 1.0" released with visual studio .NET 2002.
And the latest version is "C# 13.0" which was released on Jan 2024

C# programing lang. was created for software development in the .NET framework.

C# adopt almost all the features of C, C++, and Java programming languages but there are some advanced features which are available only in C#.

C# is a case sensitive programming language.

.CS is the extension of C# file

Why C#

Using C# we can create
1. Mobile application
2. web application
3. Desktop application
4. Games
5. Database applications etc

C# Syntax :-

```
Using System;
namespace First
{
  Class Program
  {
    Static Void Main (String [] args)
    {
      Console.WriteLine ("Hello");
    }
  }
}
```


Q what is data type?
Ans Data type specifies the different sizes and values that can be assigned on the variable.

Example :

Data type	Sizes
int	4 bytes
char	2 bytes
bool	1 bit
float	4 bytes
double	8 bytes
String	2 byte each char
Long	8 bytes

Q what is variable?
Ans Variable refers to the memory location name where we stored values.

Keywords
keywords are the pre-defined set of reserved words that have special meaning in a program

C# keyword list :-

using	Struct	abstract	decimal
int	Switch	base	enum
char	throw	case	default
float	Uint	try	implicit
double	u long	catch	in
long	unchecked	checked	internal
short	unsafe	class	lock
virtual	ushort	const	new (etc.)

Coments in C#
Comments are the part of code which is not executed and used for the future understanding of the code.

// This is a single line comment.

/* This is
a multi
line comment */

Input/Output in C#

String value = ReadLine()
Integer value = ToInt32() // 32, 16, 64
# By Default the Input is string.

for string

```
Class Program {
    Public static void Main (string[] args)
    {
        String name;
        Console.Write("Enter your name:");
        name = Console.ReadLine();

        Console.Write("My name is:" + name);
    }
}
```

Terminal:- Enter your name: Himanshu
My Name is Himanshu

for Integer

```
Class Program {
    Public static void Main (string[] args) {
        String name;
        Console.Write("Enter any Number:");
        name = Console.ReadLine();
    int num = Convert.ToInt32(name);

        Console.Write("My Value is:" + num);
    }
}
```

OR

```
int num;
Console.write("Enter any Number:");
num = Convert.ToInt32(Console.ReadLine());
Console.Write("My value is:" + num);
```

Terminal:- Enter any Number: 32
My Value is 32

Control flow statement:-

1) Conditional statement
- if
- if-else
- nested if
- else if
- switch statement

② Looping statement
- while
- do while
- for loop

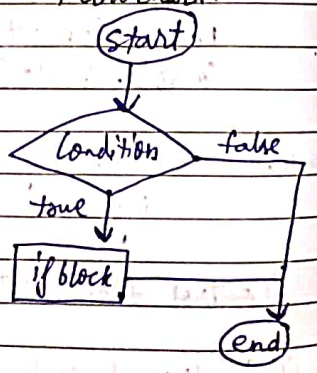
③ Jumping statement
- break
- Continue
- return

Conditional statement

① if statement :-

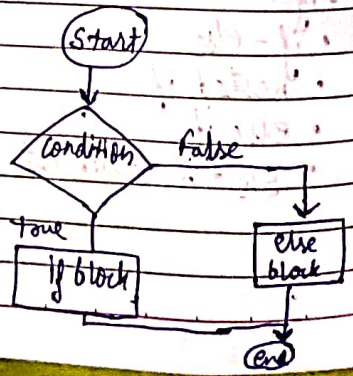
Syntax :-
```
if (condition) {
    // statements;
}
```

flowchart :-



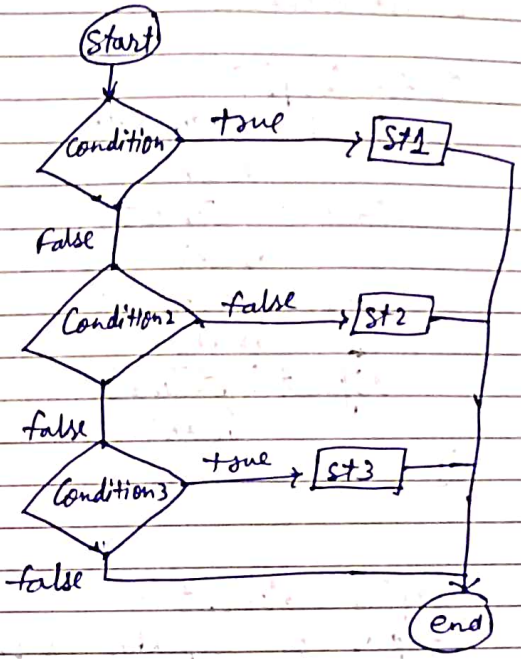
② if else statement :-

```
if (condition) {
    // true;
} else {
    // false;
}
```



③ else-if ladder :-

```
if (condition)
{ // statement 1;
} else if (condition)
{ // statement 2;
} else {
    statement 3;
}
```

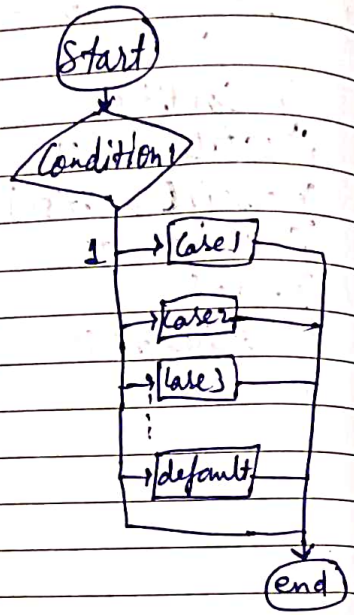


④ nested if statement :-

```
if (condition 1) {
    if (condition 2) {
        // statement
    } else {
        // code
    }
} else {
    if (condition 3) {
        // statement
    } else {
        // code
    }
}
```

Switch Statement:

```
Switch (exp) {
    case 1 : Statement;
             break;
    case 2 : Statement;
             break;
    case 3 Statement;
             break;
    default : Statement;
              break;
}
```

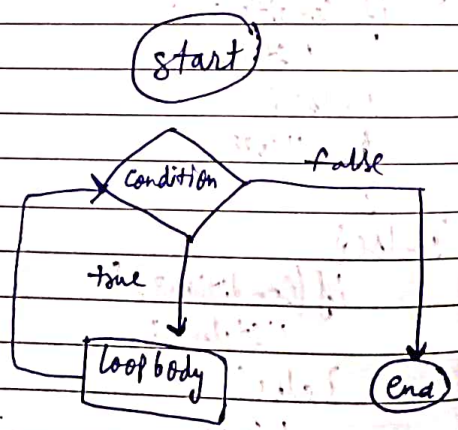


Looping Statement :-
Looping or repeating a particular code until a condition is met.

Types :-

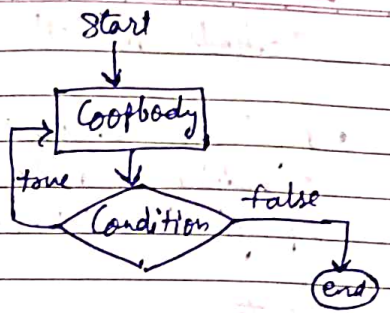
1) while

```
while (condition) {
    // Statement;
    inc/dec;
}
```



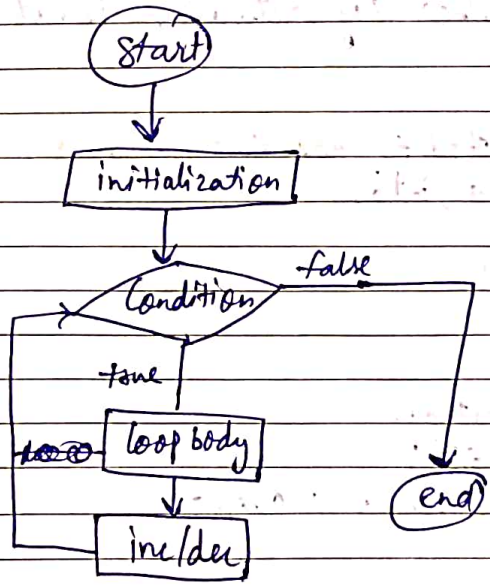
2) do while :-

```
do
{ // Statement;
  inc/dec;
} while (condition);
```



3) for loop :

```
for (initialization ; Condition ; inc/dec) {
        // Statement;
}
```



4) for each

```
for each (type var in array-name)
{
    // Statement
}
```

Methods in C#

- **Defining Methods:-**
A method is a block of code that performs a specific task. It helps in code reusability and modularity.

Syntax

```
// Method defination
return Type: MethodName (parameters)
{
    // Method body
    return value; // if return Type is not void)
}

// Example method to add two numbers
int Add (int a, int b)
{
   return a+b;
}

// Calling the method.
int sum = Add (5,3);
Console. Writeline (sum); // Output 8
```

Note:-

- **Method Overloading**
Method overloading allows multiple methods with the same name but different parameters (types, number, or order).

Syntax:

```
Class Program
{
  // Overload. method two parametrs
  Static int Multiply (int a, int b)
  {
    return a*b;
  }

  // Overload method with three parameters
  static int Multiply (int a, int b, int c)
  {
    return a*b*c;
  }

  Stati void Main ()
  {
    Console. Writeline (Multiply (2,3)); // Calls first method
                                         → Output : 6

    Console. Writeline (Multiply (2,3,4)); // Calls second method
                                           → Output : 24
  }
}
```

Note:- The method signature must differe (number/type of parameters), but the return type doesnot effect overloading.

- Optional Parameters in methods
Optional parameters allows you to call a method providing all arguments, as default values are assigned.

Syntax:

```
Void Greet (string name, string message = "welcome!")
{
  Console.WriteLine ($"{message}, {name}!");
}
Greet ("Himanshu"); // Output: welcome!, Himanshu!
Greet ("Himanshu", "Good Morning"); // output Good Morning
                                    // Himanshu!
```

Note:- Optional parameters must be declared at the end of the parameter list.

- Error & Exception Handling in C#
Exceptions are runtime errors that disrupt program execution - Try-Catch-Finally blocks help handle errors gracefully.

Key points

- Try → Contains code that may cause an exception
- Catch → Handles the exception (specific or generic).
- Finally → Executes whether or not an exception occurs

Syntax:

```
Try
{
  // code that may throw an exception
  int result = 10/0; // Division by zero (throws exception)
}
Catch (DivideByZeroException ex)
{
  Console.WriteLine ($"Error: {ex.Message}");
}
Finally
{
  Console.WriteLine ("Exception completed."); // Always execute
}
```

Custom Exceptions in C#

You can define your own custom exceptions class by extending Exception.

Key Points:

- Custom Exceptions allow you to define specific error-handling logic.
- They should inherit from Exception and pass the message to the base class.

Syntax:

```
Class CustomException : Exception {
    Public CustomException (string message) : base(message)
}
Class Program
{
    static void Validate (int age)
    {
        if (age < 18)
        {
            throw new CustomException("Age must be 18 or above.");
        }
        Console.WriteLine("Access granted.");
}
static void Main()
{
    try
    {
        Validate (16); // Throws exception
    }
    Catch (CustomException ex)
    {
        Console.WriteLine ($"Custom Exception : {ex.Message}");
    }
}
}
```


Day 3: Object-Oriented Programming (OOP) in C#

* What is object-oriented Programming (OOP)?

Object oriented programming (OOP) is a programing paradigm that focuses on organizing code into objects, which are instances of classes. It enhances modularity, reusability, and maintainability of code.

The four OOP Principles:

1. Encapsulation: It bundles data and methods into a single unit.
2. Inheritance - Allowing a class to derive properties and methods from another class.
3. Polymorphism - Using the same interface for different data types or methods.
4. Abstraction - Hiding complex implementation details and showing only the necessary parts.

* Classes & objects in C#

Definitions:

- Class: A blueprint for creating objects.
- Objects: An instance of a class that holds actual data

Syntax:

```
// Define a class
class Person
{
  Public string Name;
  Public int Age;

  // Method inside the class
  Public void Greet()
  {
    Console.WriteLine($"Hello, my name is {Name}
                        and I am {Age} years old.");
  }
}

// Create an object of the class

Class Program
{
  Static Void Main()
  {
    Person P = new Person();
    P.Name = "Himanshu";
    P.Age = 22;
    P.Greet(); // Output: Hello, my name is Himanshu
                // and I am 21 years old.
  }
}
```

Note:- Objects allow us to access and modify class properties and methods.

* Constructors and Deconstructors in C#

Constructors:-

- A Constructor is a special method called when an object is created.
- It initializes the objects properties.

Syntax:

```
Class Person
{
  Public string Name;
  Public int Age;

  // Constructor (same name as class)
  Public Person(string name, int age)
  {
    Name = name;
    Age = age;
  }

  Public Void Greet()
  {
    Console.WriteLine($"Hello, my name is {Name}
                        and I am {Age} years old.");
  }
}
Class Program
{
  Static void Main()
  {
    Person P = new Person("Himanshu", 21);
    P.Greet();
  }
```

Note:- The Constructor is called automatically when an object is created.

Destructor:
- Destructor is used to clean up resources when an object is destroyed.
- It is defined using ~ class Name().

Syntax:

```
Class Sample
{
  // Destructor
  ~Sample()
  {
    Console.WriteLine("Destructor Called - Object is being destroyed.");
  }
}
```

Note:- Destructors cannot be overload and are called automatically when the object goes out of scope.

* Inheritance in C#

Definition:-
Inheritance allows a child class to use the properties and methods of a parent class.

Syntax:

```
// Base class (Parent)
class Person {
  Public string Name;
  Public int Age;

  Public void Greet()
  {
    Console.WriteLine($"Hello, I am {Name}.");
  }
}
// Derived class (Child)
class Student : Person {
  Public string Grade;
  Public void ShowGrade()
  {
    Console.WriteLine($"{Name} is in grade {Grade}.");
  }
}
class Program {
    Static Void Main() {
        Student s = new Student();
        S.Name = "Himanshu";
        S.Age = 22;
        S.Grade = "A";

        S.Greet(); // output: Hello, I am Himanshu.
        s.ShowGrade(); // output: Himanshu is in
                       //grade A.
    }
}
```

Note:- • The child class (student) inherits properties (Name, Age) and methods (Greet()) from the parent class (Person).

• New properties (Grade) and methods (ShowGrade) can be added in the child class.

* Access Modifiers in C#

Definition:
Access modifiers control the visibility of class members.

Types of Access Modifiers:

- Public :- Accessible from anywhere
- Private :- Accessible only within same class.
- Protected :- Accessible within the same class and derived classes.
- internal :- Accessible within the same assembly.

```
Class Example {
    Public int PublicVar = 10; // Can be accessed anywhere
    Private int PrivateVar = 20; // Can be accessed only in
                                 // this class.
    Protected int ProtectedVar = 30; // Can be accessed in
                                     // derived classes.
}
```


Note:- Use private to protect sensitive data and Public for widely accessible members.

* Polymorphism in C# (Method Overriding)

Definition:
Polymorphism allows methods to behave differently based on the object calling them.

Example:-

```
// Base class
Class Person {
    Public virtual void Speak() {
        Console.WriteLine("Person is speaking...");
    }
}
// Derived Class
class Student : Person {
    Public override void Speak() {
        Console.WriteLine("Student is asking a question.");
    }
}
Class Program {
    Static void Main() {
        Person p = new Person();
        p.Speak(); // output: Person is speaking...

        Student s = new Student();
        s.Speak(); // output: Student is asking a question...
    }
}
```

Note:- • Use virtual in the base class method.
• Use override in the derived class method to modify behavior.

* Abstraction in C# (Abstract Classes & Interfaces)

Definition:-
Abstraction hides implementation details and only exposes relevant information.

Using Abstraction Classes:-

```
abstract class Animal {
    Public abstract void MakeSound();
        // AbstractMethod (no body)
}
// Derived class must implement abstract method
class Dog : Animal {
    Public override void MakeSound() {
        Console.WriteLine("Dog barks!");
    }
}
Class Program {
    static void Main() {
        Dog d = new Dog();
        d.MakeSound(); // Output: Dog barks!
    }
}
```

Note:- • Abstract classes cannot be instantiated.
• Derived classes must implement abstract methods.

Day 4: .NET Core Project Structure

* Anatomy of a .NET Core Project

When we create a new ASP.NET Core project, you'll see a well-organized folder structure.
Understanding these files is essential for working with .NET applications.

Key files & folders in a .NET Core project

Program.cs → The entry point of the application, where the app is configured and started.

appsettings.json → Stores configuration settings such as databases connections and logging settings.

Controllers/ → Contains MVC Controllers handling HTTP requests (for Web APIs and MVC projects).

Models/ → Defines data structures used in the applications.

Views/ → Contains Razor views for rendering UI (only in MVC projects).

wwwroot/ → Holds static files like CSS, JavaScript, and images.

Dependencies/ → Manages installed Nuget packages and external libraries

Basic .NET Core Project Example

In Program.cs

```
Var builder = WebApplication.CreateBuilder(args);

// Configure services
builder.Services.AddControllers();

Var app = builder.Build();

// Configure the HTTP request pipeline
app.UseRouting();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Note :-
- UseRouting() enables endpoint routing.
- UseAuthorization() enables authentication & authorization.
- MapControllers() map HTTP requests to controllers.

* Understanding Dependency Injection in .NET Core

What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern used in .NET Core to manage dependencies efficiently.

It helps reduce tight coupling and makes code more maintainable.

Types of Dependency Injection in .NET Core

- Transient - New instance is created every time it is required.
- Scoped - Instance is created once per request.
- Singleton - One instance is created and shared across the application.

Registering Dependencies in .NET Core

```
// Define an interface
Public interface IGreeting service {
    String Greet();}

// Implement the interface
Public class GreetingService : IGreeting Service {
    Public String Greet() => "Hello, welcome to .NET Core!";
}

// Registering dependency in Program.cs
Var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IGreeting Services, GreetingServices>();
                                        // Singleton instance
Var app = builder.Build();

app.MapGet("/", (IGreeting Service service) => service.Greet);

app.Run;
```

Note:- • Services are registered in builder.Services.
• Injected via Constructor in classes or directly in MapGet().
• DI makes it easy to manage dependencies and unit test components.

* Middleware in .NET Core.

What is Middleware?
Middleware is a software that processes HTTP requests and responses in a pipeline. Each middleware can modify the request, response, or pass it to the next middleware.

Default Middleware in .NET Core

UseRouting() → Enables request routing.

UseAuthentication() → Handles authentication (JWT, OAuth, etc).

UseAuthorization() → Applies authorization rules.

UseStaticFiles() → Serves static files from wwwroot

UseEndpoints() → Defines how requests map to Controllers, Razor Pages, or SignalR hubs.

Creating Custom Middleware in .NET Core
Custom middleware allows you to add your own logic in the HTTP request pipeline.

Example: Logging Middleware

```
Public class LoggingMiddleware {
    Private readonly RequestDelegate next) {
      _next = next;
    }
    Public async Task Invoke (HttpContext context){
        Console.WriteLine ($"Request : {Context.Request.
                - Method} {Context.Request.Path}"
        await _next (context); // Call next middleware
        Console.WriteLine ($"Response! {Context.Response.
                - StatusCode}" );
    }
}
// Register Middleware in Program.cs
Var builder = WebApplication.CreateBuilder (args);
Var app = builder.Build ();
app.UseMiddleware<LoggingMiddleware> (); // Add Custom
                                         // Middleware
app.MapGet ("/", () => "Hello, .NET Core!");
app.Run()
```

Note:- • Invoke() method Processes requests and responses. Pipeline
• await _next (context) calls the next middleware in the
• Middleware must be Register with Program.cs using app.UseMiddleware<T>()