Core C# Programming	
Arrays in C#	Array : a collection of the same data type , stored in contiguous memory location and can be accessed via zero based index. Why we need array when we have so many types of variable in C# ? int [] marks = new int [100]; **We need arrays to simplify data storage, processing, and scalability when dealing with multiple related values.** What are the features of Arrays/ in what scenarios we should use them ? Limitation of using arrays in C#? types of arrays in C#
	Single array
	Multi-dimension
	Jagged array
Looping Construct	Loops and Statement
	IF statement
	For loop
	Foreach loop
	Switch statement
	While loop (entry controlled)
	Do-while loop (exit controlled)

Major diff for loop and for each

1. we don't need an index. hence it is a read only loop.
2. forward only loop
3. loop starts from 1st element and goes till last element without any condition.
4. they are majorly used for collections, where we dont have index to iterate.

Access modifiers in C#	Access modifiers
	Private
	Public
	Protected
	Internal
	Protected internal
Methods in C#	Methods
	Void method
	With parameter

	Return type method
	Static method
	Instance method
	Namespaces
	Unit testing

Inheritance in C#.NET	Object Oriented Concept OOP paradigm helps us in organizing software design around objects and classes. core principle of OOPs: -inheritance -Encapsulation -Abstraction -Polymorphism We implement inheritance for re-usability and parent -child relationship.
	Inheritance Are constructor inherited in C# ?
	Single : base class -> child class
	Multilevel: chain of inheritance base class -> Intermediate class -> child class -this supports logical layering -suports hierarchical code reuse - supported in C#
	Multiple (using Interface) class : interface 1, interface 2 -Avoids ambiguity issue (Diamond issue) - A B C D If B and C overrides a method from A and D calls that method Which implementation should be used ?

	Hierarchical : System.object act a hierarchical base class Derived class 1: Base class Derived Class 2: base class
	Hybrid : combination of two or more inheritance types most common in enterprise applications base class | Derived class -> Interface
OOPs principles ie Polymorphism in Action	
Polymorphism in C#.NET	Polymorphism An object takes many forms in C# we can achieve polymorphism via any one of the way : 1. Virtual 2. Override 3. method signatures calculating interest() -> saving account and corporate Account() ● Run time polymorphism ● Compile time polymorphism
	Method override - Virtual, override (Derive class provides a specific implementation of a base class method) - Inheritance SavingsAccounts overrides CalculateIntrest() with new functionality Reduced Taxes (GST) _ >

	Method overload (Compile time Polyphism) Speak() | Area() | Class Animal Class Bird Class Fish
	Encapsulation ● Wrapping up of a data & methods ● private and Public , properties ● Account balance can be accessed via getbalance() instead of direct variable Note: Figure out Ecommerce and Fintech scenario based on polymorphism
	Abstraction ● Hiding implementation details and showing only essential behavior ● Abstract & interface ● Account defines calculateIntrest() without implementation

Note: Figure out Ecommerce and Fintech scenario based on polymorphism

Access Modifier	Accessible Within Same Class	Same Assembly (Project)	Derived Class (Same Assembly)	Derived Class (Different Assembly)	Outside Class	Typical Use Case
private	✅ Yes	❌ No	❌ No	❌ No	❌ No	Hide implementation details within a class
public	✅ Yes	✅ Yes	✅ Yes	✅ Yes	✅ Yes	Expose APIs, services, models
protected	✅ Yes	❌ No	✅ Yes	✅ Yes	❌ No	Allow inheritance-based access

internal	✅ Yes	✅ Yes	✅ Yes	❌ No	❌ No	Share logic within same project/assembly
protected internal	✅ Yes	✅ Yes	✅ Yes	✅ Yes (derived only)	❌ No	Library APIs meant for extension or internal use

1. Private -> Most restrictive, default for class members
2. Public -> Least restrictive, (use carefully)
3. Protected ->Inheritance scenario
4. Internal -> best layered architecture within the solution(MVC, MVT, MVVM)
5. Protected internal -> Accessible either by
    a. any class in the same assembly
    b. or derived classes in other assemblies.

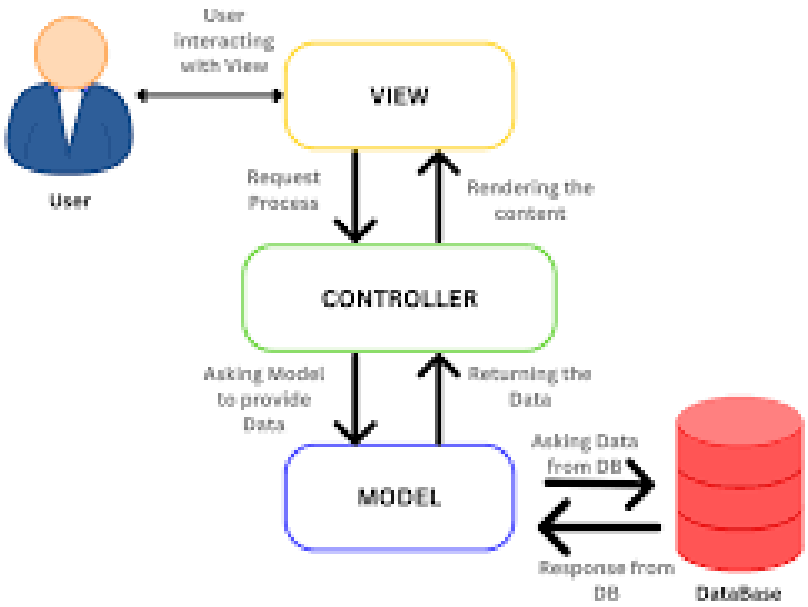
Why is layered architecture the back bone of Enterprise computing till now ?

Breakfast joint - 200-250
Poha
Alu-paratha
Dosa

Kitchen — 500-750
Service
marketing

Franchise

console application (BL is common in the application)
Web Application (Server —> client)
Web Application with API (Api →server —->clients)
layered architecture



Case study :

A bank is building a **Bank Account Management System**.
The system must:

- Secure sensitive data
- Allow controlled inheritance for different account types
- Restrict access across assemblies
- Expose only required functionality to external applications

Requirement	Access Modifier Used
Account PIN must be hidden	`private`
Account number accessible everywhere	`public`
Interest calculation for derived accounts	`protected`
Bank audit logic shared within bank project	`internal`
Extension by partner banks & internal use	`protected internal`

Case study based on polymorphism in C#

An e-commerce platform supports :

1. Multiple products types
2. Different payment options
3. Secure order data
4. Flexible discount and shipping logic.

Polymorphism	One interface, multiple implementations			
Method overloading	Multiple ways to place an order	Placeorder(in tqyy), placeorder(int qty, string discount)		
method overriding	Product specific discount	Electronic overrides CalculateDiscount(), orderData()		
Encapsulation	Order price protected from direct modification	price accessed via getter and setter (properties)		
Abstraction		Abstract product class		

Abstract class in C# :

1. These classes cannot create objects directly.
2. They are designed to be part of inheritance only.

Difference between Abstract class and Interface

Aspect	Abstract Class	Interface
Purpose	Provides a base class with partial implementation	Defines a pure contract or behavior
Keyword Used	`abstract`	`interface`
Object Creation	Cannot be instantiated	Cannot be instantiated
Method Implementation	Can have implemented and abstract methods	Methods have no implementation (except default methods from C# 8+)
Fields (Variables)	Allowed	Not allowed
Properties	Allowed	Allowed (without backing fields)
Access Modifiers	Allowed (`public`, `protected`, etc.)	Members are public by default
Constructors	Allowed	Not allowed
Inheritance	Supports single inheritance only	Supports multiple inheritance
Multiple Inheritance	Not supported	Supported
Method Types	Can have abstract, virtual, and non-virtual methods	Methods are abstract by default
State Management	Can maintain state	Cannot store state
Use Case	Used when classes share common behavior and logic	Used when classes share capability or rule
Performance	Slightly faster	Slightly slower due to indirection
Versioning Impact	Adding a method does not break derived classes	Adding a method breaks existing implementations

Example Use Case	Product, Employee, Account	IDiscountable, ILogger, IPaymentGateway

Scenario	Inheritance Type
Account → SavingsAccount	Single
Vehicle → Car → ElectricCar	Multilevel
PaymentGateway implementing IOnlinePayment, ILogger	Multiple (Interface)
Shape → Circle, Rectangle	Hierarchical
Employee → Manager + IApprovable	Hybrid