C# Scenario based Interview Questions by Karthik M

1. You are designing a billing system. How would you enforce that every bill must implement CalculateTotal() method without providing any default behavior?

Answer:
I would create an interface like this:

```
interface IBill
{
    double CalculateTotal();
}
```

Now, every billing class like Invoice or PurchaseOrder must implement the CalculateTotal() method explicitly.

Tip : This ensures flexibility and forces a contract.

2. In a real-world app, you want to initialize database connections only once for all users. How will you implement it?

Answer:
I would use the Singleton pattern like this:

```
class DBConnection
{
    private static DBConnection instance;
    private DBConnection() {}

    public static DBConnection Instance
    {
        get
        {
            if (instance == null)
                instance = new DBConnection();
            return instance;
        }
    }
}
```

Note: This ensures a single connection object reused across the application.

3. How will you make sure that a Customer object is immutable once created?

Answer:
I would make all properties readonly and initialize them inside the constructor:

```
class Customer
{
    public string Name { get; }
    public int Age { get; }

    public Customer(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Note: Once constructed, object values can't be changed.

4. You have a Shape base class and Circle, Rectangle as derived classes. How would you design a method Draw() differently for each?

Answer:
Use virtual and override keywords properly like below:

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing Shape");
    }
}

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

class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Rectangle");
    }
}
```

Note: Different derived classes give different implementations dynamically (Polymorphism).

5. In an E-commerce platform, you want different products to have different price calculation logic. How will you design it?

Answer:
Use the Strategy Pattern by creating different price calculators:

```
interface IPriceStrategy
{
    decimal CalculatePrice(Product p);
}

class DiscountPrice : IPriceStrategy
{
    public decimal CalculatePrice(Product p)
    {
        return p.BasePrice * 0.9m;
    }
}

class PremiumPrice : IPriceStrategy
{
    public decimal CalculatePrice(Product p)
    {
        return p.BasePrice * 1.2m;
    }
}
```

Note: Different pricing logic can be injected at runtime without changing product code.

6. How can you restrict a class from being inherited further?

Answer:
Use the sealed keyword in C#.

Example:

```
sealed class BankAccount
{
    public void DisplayBalance()
    {
        Console.WriteLine("Balance shown.");
    }
}
```

Note: sealed prevents any other class from extending BankAccount.

7. How do you enforce multiple inheritance in C# without using classes?

Answer:
C# doesn't support multiple inheritance with classes but it supports it via interfaces.

Example:

```
interface IPrintable
{
    void Print();
}

interface IScannable
{
    void Scan();
}

class MultiFunctionDevice : IPrintable, IScannable
{
    public void Print()
    {
        Console.WriteLine("Printing...");
    }

    public void Scan()
    {
        Console.WriteLine("Scanning...");
    }
}
```

Note: Multiple interfaces allow a class to inherit multiple behaviors.

8. What would you use to define fixed data like Status {Pending, Approved, Rejected}?

Answer:
I would use an enum for this type of scenario.

Example:

```
enum Status
```

```
{
    Pending,
    Approved,
    Rejected
}
```

Note: Enums provide meaningful names to integral values, improving code readability.

9. You need to provide method overloading to handle both int and float types for the same function AddNumbers(). How would you design it?

Answer:
I would implement two overloaded versions of the method.

Example:

```
class Calculator
{
    public int AddNumbers(int a, int b)
    {
        return a + b;
    }

    public float AddNumbers(float a, float b)
    {
        return a + b;
    }
}
```

Note: Method Overloading allows multiple methods with the same name but different parameter types.

10. Can we overload constructors? Why is it useful in real-time applications?

Answer:
Yes, we can overload constructors to allow different ways of object initialization.

Example:

```
class Student
{
    public string Name;
    public int Age;

    public Student()
    {
        Name = "Unknown";
        Age = 0;
    }

    public Student(string name)
    {
        Name = name;
        Age = 0;
    }

    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Note: Real-world use: Providing flexibility to create objects with full or partial information.

11. In a library management system, you need to ensure that once a book is issued, it cannot be modified. How would you design it?

Answer:
I would create an immutable class for the book issue record.

Example:

```
class IssuedBook
{
    public string BookName { get; }
    public string IssuedTo { get; }
    public DateTime IssuedDate { get; }

    public IssuedBook(string bookName, string issuedTo, DateTime issuedDate)
    {
        BookName = bookName;
        IssuedTo = issuedTo;
        IssuedDate = issuedDate;
    }
}
```

Note: This way, once created, IssuedBook details cannot be changed.

12. How would you implement a base class that forces child classes to override a method but still allows base functionality?

Answer:
Use an abstract class with abstract methods.

Example:

```
abstract class Report
{
    public void PrintHeader()
    {
        Console.WriteLine("Company Confidential Report");
    }

    public abstract void PrintDetails();
}

class SalesReport : Report
{
    public override void PrintDetails()
    {
        Console.WriteLine("Sales Details Printed");
    }
}
```

Note: This enforces PrintDetails() to be overridden, but allows reuse of PrintHeader().

13. In a payment gateway integration, how would you implement dynamic switching between CreditCard and PayPal payments?

Answer:
I would use interface-based strategy switching.

Example:

```
interface IPaymentGateway
{
    void ProcessPayment(double amount);
}

class CreditCardPayment : IPaymentGateway
{
    public void ProcessPayment(double amount)
    {
        Console.WriteLine($"CreditCard Payment done for {amount}");
    }
}

class PayPalPayment : IPaymentGateway
{
    public void ProcessPayment(double amount)
    {
        Console.WriteLine($"PayPal Payment done for {amount}");
    }
}
```

Note: Depending on user choice, dynamically assign the appropriate payment method object.

14. Explain what happens when you mark a method as virtual but don't override it in child class?

Answer:
If a method is marked virtual in the base class but not overridden in the derived class, the base class version of the method is called.

Example:

```
class Employee
{
    public virtual void DisplayRole()
    {
        Console.WriteLine("Employee Role");
    }
}

class Manager : Employee
{
    // No override here
}
```

When you call DisplayRole() on Manager, it prints:
Employee Role

Note: Only if you override, the derived class version is called.

15. In a chat application, you want users to be able to "SendMessage" differently based on platform (Mobile/Desktop). How would you design it?

Answer:
I would define an interface and provide different implementations.

Example:

```
interface IChatPlatform
{
    void SendMessage(string message);
}

class MobileChat : IChatPlatform
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Sending via Mobile: {message}");
    }
}

class DesktopChat : IChatPlatform
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Sending via Desktop: {message}");
    }
}
```

Note: At runtime, based on device type, create appropriate object and send the message.

16. How would you design a system where certain functionality should only be accessible to Admin users at runtime?

Answer:
I would use interface segregation and role-based checks.

Example:

```
interface IUserFunctions
{
    void ViewProfile();
}

interface IAdminFunctions : IUserFunctions
{
    void DeleteUser();
    void AddUser();
}

class AdminUser : IAdminFunctions
{
    public void ViewProfile()
    {
        Console.WriteLine("Viewing Admin Profile");
    }

    public void DeleteUser()
    {
        Console.WriteLine("Deleting User");
    }

    public void AddUser()
    {
        Console.WriteLine("Adding User");
    }
}
```

Note: Based on role, we decide which interface methods are exposed to the user.

17. How would you reuse logic across multiple unrelated classes without inheritance?

Answer:
Use Composition rather than inheritance.

Example:

```
class Logger
{
    public void Log(string message)
    {
        Console.WriteLine($"Log: {message}");
    }
}

class OrderService
{
    private Logger _logger = new Logger();

    public void PlaceOrder()
    {
        _logger.Log("Order Placed");
    }
}

class PaymentService
{
    private Logger _logger = new Logger();

    public void ProcessPayment()
    {
        _logger.Log("Payment Processed");
    }
}
```

Note: Composition promotes code reuse without tying classes together rigidly.

18. What is the use of base keyword? Give a real example.

Answer:
base is used to access members of the base class from within a derived class.

Example:

```
class Vehicle
{
    public void Start()
    {
        Console.WriteLine("Vehicle started");
    }
}

class Car : Vehicle
{
    public void StartCar()
    {
        base.Start();
        Console.WriteLine("Car is ready to drive");
    }
}
```

Note: base.Start() allows reuse of the Vehicle's Start method inside Car without redefining.

19. What is the benefit of using abstract classes compared to interfaces?

Answer:

Abstract classes allow you to provide partial implementation (methods with body + methods without body).

Interfaces only allow method declarations (until C# 8.0).

Example:

```
abstract class Animal
{
    public void Breathe()
    {
        Console.WriteLine("Breathing...");
    }

    public abstract void MakeSound();
}

class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Barks");
    }
}
```

Note: Abstract classes allow code reuse plus enforce important abstract methods.

20. In a microservices project, services should be loosely coupled but must agree on contracts. How would you enforce this?

Answer:
I would define shared interfaces across services.

Example:

```
interface IUserService
{
    string GetUserName(int userId);
}
```

Each microservice can have different implementations but must respect the interface contract.

Note: Helps in loosely coupled but well-defined architecture.

21. In a banking system, you want to hide sensitive fields like AccountBalance but still allow displaying AccountHolderName. How would you design it?

Answer:
I would use access modifiers and properties.

Example:

```
class BankAccount
{
    public string AccountHolderName { get; set; }
    private double AccountBalance { get; set; }
    public BankAccount(string name, double balance)
    {
        AccountHolderName = name;
        AccountBalance = balance;
    }
}
```

Note: AccountBalance cannot be accessed directly outside the class.

22. Explain the difference between override and new keywords in method overriding.

Answer:

override extends the base class method and provides new behavior.

new hides the base class method intentionally (not polymorphic).

Example:

```
class Base
{
    public virtual void Show()
    {
        Console.WriteLine("Base Show");
    }
}

class Derived : Base
{
    public new void Show()
    {
        Console.WriteLine("Derived Show");
    }
}
```

Note: If you use a base class reference, it calls Base’s Show().

Note: For polymorphism, override should be used, not new.

23. You need to implement loose coupling between UI layer and Business Logic layer. How will you achieve it?

Answer:

I would use interfaces or dependency injection (DI).

Example:

```
interface IEmployeeService
{
    string GetEmployeeName(int id);
}

class EmployeeService : IEmployeeService
{
    public string GetEmployeeName(int id)
    {
        return $"Employee_{id}";
    }
}
```

In UI Layer:

```
class EmployeeUI
{
    private IEmployeeService _employeeService;
    public EmployeeUI(IEmployeeService service)
    {
        _employeeService = service;
    }

    public void ShowName(int id)
    {
        Console.WriteLine(_employeeService.GetEmployeeName(id));
    }
}
```

Note: This way, UI depends only on interface, not on concrete classes.

24. Can you create an object of an abstract class? How is it useful?

Answer:
No, you cannot create an object of an abstract class directly.
However, abstract classes are useful for defining a base template.

Example:

```
abstract class Document
{
    public abstract void Print();
}

class PdfDocument : Document
{
    public override void Print()
    {
        Console.WriteLine("Printing PDF Document");
    }
}
```

Note: Forces derived classes to implement important methods like Print().

25. Explain Interface Segregation Principle (ISP) with example.

Answer:
ISP says — don’t force a class to implement methods it doesn’t use.
Better to split into smaller interfaces.

Example (Bad Design):

```
interface IMachine
{
    void Print();
    void Fax();
    void Scan();
}

class OldPrinter : IMachine
{
    public void Print()
    {
        Console.WriteLine("Printed");
    }

    public void Fax()
    {
        throw new NotImplementedException();
    }

    public void Scan()
    {
        throw new NotImplementedException();
    }
}
```

Note: Problem: OldPrinter is forced to implement Fax() and Scan() unnecessarily.

Good Design:

```
interface IPrinter
{
    void Print();
}

interface IScanner
{
    void Scan();
}

class SimplePrinter : IPrinter
{
    public void Print()
    {
        Console.WriteLine("Printed");
    }
}
```

Note: Classes implement only what they need.

26. When would you prefer an abstract class over an interface?

Answer:
Use an abstract class when:

You want to provide default implementation for some methods.

You want to define common fields or constructors.

Example:

```
abstract class Animal
{
    public string Name;

    public Animal(string name)
    {
        Name = name;
    }

    public void Breathe()
    {
        Console.WriteLine($"{Name} is breathing");
    }

    public abstract void MakeSound();
}

class Dog : Animal
{
    public Dog(string name) : base(name) {}
    public override void MakeSound()
    {
        Console.WriteLine($"{Name} barks");
    }
}
```

Note: Abstract classes are perfect when some shared logic needs to be inherited.

27. What happens if you don't implement all methods of an interface in your class?

Answer:

The compiler will throw an error.

To fix it, the class must either be declared abstract, or all methods must be implemented.

Example:

```
interface IShape
{
    void Draw();
    void Resize();
}

class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }

    // Forgot to implement Resize() — ❌ Compiler Error
}
```

Note: Must implement both Draw() and Resize() or declare Circle as abstract.

28. Why can't a constructor be virtual in C#?

Answer:

Constructors are not inherited.

Since virtual relates to method overriding in derived classes, constructors cannot be virtual.

Example:

```
// This is invalid:
// public virtual MyClass() { } // ❌ Not allowed
```

Note: Instead, constructor chaining using base() can be used to control construction flow across base and derived classes.

Example:

```
class Base
{
    public Base()
    {
        Console.WriteLine("Base constructor");
    }
}

class Derived : Base
{
    public Derived() : base()
    {
        Console.WriteLine("Derived constructor");
    }
}
```

29. How would you handle a situation where a derived class needs to extend behavior but also call base behavior?

Answer:
Use base.MethodName() inside overridden methods.

Example:

```
class Notification
{
    public virtual void Send()
    {
        Console.WriteLine("Sending basic notification");
    }
}

class EmailNotification : Notification
{
    public override void Send()
    {
        base.Send();
        Console.WriteLine("Sending Email notification additionally");
    }
}
```

Note: First calls base logic, then extends behavior.

30. What is a real-world example of using sealed keyword practically?

Answer:
If you create a utility/helper class like MathHelper, and you don't want others to override or extend it accidentally, mark it as sealed.

Example:

```
sealed class MathHelper
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}

// class ExtendedHelper : MathHelper // ❌ Not allowed. Compile-time error.
```

Note: sealed ensures no one can inherit and misuse/change base behavior.

31. How do you call a base class constructor explicitly from a derived class?

Answer:
Use : base(parameters) in the derived class constructor.

Example:

```
class Person
{
    public string Name;

    public Person(string name)
    {
        Name = name;
    }
}

class Employee : Person
{
    public Employee(string name) : base(name)
    {
        Console.WriteLine($"Employee Name: {Name}");
    }
}
```

Note: Helps reusing initialization logic defined in base classes.

32. How do you achieve polymorphism using interfaces?

Answer:
By referring to objects using their interface type, not their concrete class.

Example:

```
interface IShape
{
    void Draw();
}

class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }
}

class Square : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Square");
    }
}
```

Usage:

```
IShape shape = new Circle();
shape.Draw();

shape = new Square();
shape.Draw();
```

Note: Same method Draw() behaves differently based on object type.

33. Explain the concept of explicit interface implementation.

Answer:
When two interfaces have methods with same signature, you can explicitly implement to avoid conflicts.

Example:

```
interface IPrinter
{
    void Print();
}

interface IScanner
{
    void Print();
}

class MultiFunctionMachine : IPrinter, IScanner
{
    void IPrinter.Print()
    {
        Console.WriteLine("Printing Document");
    }

    void IScanner.Print()
    {
        Console.WriteLine("Scanning Document");
    }
}
```

Usage:

```
IPrinter printer = new MultiFunctionMachine();
printer.Print();

IScanner scanner = new MultiFunctionMachine();
scanner.Print();
```

Note: Explicit Implementation clearly separates behavior.

34. How would you prevent a method in base class from being overridden?

Answer:
Declare the method as sealed in the derived class while overriding.

Example:

```
class Parent
{
    public virtual void Greet()
    {
        Console.WriteLine("Hello from Parent");
    }
}
```

```
class Child : Parent
{
    public sealed override void Greet()
    {
        Console.WriteLine("Hello from Child");
    }
}

// class GrandChild : Child { override Greet() ❌ Not Allowed }
```

Note: sealed override prevents further overriding in the inheritance chain.

35. How would you implement Dependency Injection manually in C#?

Answer:
You can inject dependencies via constructor without any framework.

Example:

```
interface INotification
{
    void Send(string message);
}

class EmailNotification : INotification
{
    public void Send(string message)
    {
        Console.WriteLine($"Sending Email: {message}");
    }
}

class UserService
{
    private INotification _notification;

    public UserService(INotification notification)
    {
        _notification = notification;
    }

    public void RegisterUser(string username)
    {
        Console.WriteLine($"{username} Registered");
        _notification.Send("Welcome to our app!");
    }
}
```

Usage:

```
INotification notifier = new EmailNotification();
UserService service = new UserService(notifier);
service.RegisterUser("Karthik");
```

Note: Manual DI improves testability and flexibility.

36. What is method hiding in C#? How is it different from overriding?

Answer:

Hiding uses the new keyword.

Overriding uses the override keyword and works via polymorphism.

Example of method hiding:

```
class Base
{
    public void Display()
    {
        Console.WriteLine("Base Display");
    }
}

class Derived : Base
{
    public new void Display()
    {
        Console.WriteLine("Derived Display");
    }
}
```

Usage:

```
Base obj = new Derived();
obj.Display();
```

Note: Output: Base Display (because hiding does not change base behavior with base reference)

37. Can you override a non-virtual method in C#?

Answer:
No, you cannot override a non-virtual method.
Only methods declared as virtual, abstract, or override can be overridden.

Example:

```
class Parent
{
    public void SayHello()
    {
        Console.WriteLine("Hello from Parent");
    }
}

class Child : Parent
{
    // public override void SayHello() ❌ Compile-time Error
}
```

Note: Always mark the method virtual first if you want it to be overridden.

38. What are extension methods? Why are they useful?

Answer:

Extension Methods allow you to add methods to existing types without modifying them.

Very useful for working with external or sealed classes.

Syntax:

```
public static class StringExtensions
{
    public static int WordCount(this string str)
    {
        return str.Split(' ').Length;
    }
}
```

Usage:

```
string message = "Hello Karthik How Are You";
int count = message.WordCount();
Console.WriteLine(count);
```

Note: Adds behavior without changing original class definition!

39. What is the difference between abstract class and interface?

Feature	Abstract Class	Interface
Inheritance	Single inheritance	Multiple inheritance
Constructors	Allowed	Not allowed
Access Modifiers	Can have public/private/protected	Always public methods (before C#8)
Default Implementation	Possible (Concrete methods)	Not possible (until C#8)

Note: Abstract classes are partially implemented templates.

Interfaces are pure contracts.

40. How can you prevent inheritance of a class?

Answer:
Use the sealed keyword.

Example:

```
sealed class BankAccount
{
    public void Deposit()
    {
        Console.WriteLine("Deposit Done");
    }
}

// class SavingsAccount : BankAccount ❌ Cannot inherit
```

Note: sealed classes are used when you want final implementations without any child class overriding or modifying the behavior.

41. What is multiple inheritance? Does C# support it?

Answer:

Multiple inheritance means a class inherits from more than one class.

C# does NOT support multiple class inheritance to avoid ambiguity (like Diamond Problem).

However, C# supports multiple interface inheritance.

Example:

```
interface IPrinter
{
    void Print();
}

interface IScanner
{
    void Scan();
}

class MultiFunctionMachine : IPrinter, IScanner
{
    public void Print()
    {
        Console.WriteLine("Printing...");
    }

    public void Scan()
    {
        Console.WriteLine("Scanning...");
    }
}
```

Note: So C# achieves multiple behavior inheritance via interfaces, not classes.

42. Why would you make a class abstract if it has no abstract methods?

Answer:
Even without abstract methods, marking a class as abstract is useful when:

You don't want the class to be instantiated directly.

You want it to act as a base only for derived classes.

Example:

```
abstract class Shape
{
    public void Move()
    {
        Console.WriteLine("Shape moved");
    }
}
```

Note: Shape is not supposed to be instantiated directly.

43. What is the use of readonly fields in a class?

Answer:

readonly fields can only be assigned during declaration or inside constructor.

Once assigned, they cannot be changed outside.

Example:

```
class Vehicle
{
    public readonly string EngineType;
    public Vehicle(string engineType)
    {
        EngineType = engineType;
    }
}
```

Usage:

```
Vehicle v = new Vehicle("Petrol");
// v.EngineType = "Diesel"; ✖ Not allowed
```

Note: Ensures immutability after object creation.

44. Can abstract classes have constructors? If yes, why?

Answer:
Note: Yes, abstract classes can have constructors!

Constructors are used to initialize fields common to derived classes.

Example:

```
abstract class Employee
{
    protected string Name;

    public Employee(string name)
    {
        Name = name;
    }
}

class Manager : Employee
{
    public Manager(string name) : base(name) {}
}
```

Note: Derived classes can call base constructor to reuse initialization logic.

45. How to force derived classes to implement certain methods?

Answer:
Declare those methods as abstract in the base abstract class.

Example:

```
abstract class Animal
{
    public abstract void MakeSound();
}

class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Dog Barks");
    }
}
```

Note: Now Dog must implement MakeSound(). Otherwise, the compiler will throw an error.

46. What is the difference between const and readonly in C#?

Answer:

Feature	const	readonly
Assignment	At compile time only	At runtime (in constructor) allowed
Type	Static by default	Instance level (by default)
Modifiable	Cannot modify even inside constructor	Can modify once inside constructor

Example:

```
class Example
{
    public const double Pi = 3.14;
    public readonly int RollNumber;

    public Example(int rollNumber)
    {
        RollNumber = rollNumber;
    }
}
```

Note: const is for fixed values.
Note: readonly is for values decided during object creation.

47. What is shadowing or method hiding in C#?

Answer:

Shadowing means redefining a method in a derived class without overriding.

Use new keyword.

Example:

```
class Base
{
    public void Display()
    {
        Console.WriteLine("Base Display");
    }
}

class Derived : Base
{
    public new void Display()
    {
        Console.WriteLine("Derived Display");
    }
}
```

Usage:

```
Base obj = new Derived();
obj.Display(); // Output: Base Display
```

Note: In shadowing, polymorphism does NOT happen.

48. What is constructor chaining in C#?

Answer:
Constructor chaining means calling one constructor from another constructor of the same class or base class.

Example:

```
class Student
{
    public string Name;
    public int Age;

    public Student() : this("Unknown", 0)
    {
    }

    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Note: Improves code reuse and reduces duplication.

49. What is an example of real-world polymorphism in C#?

Answer:
When you have common behavior with different implementations.

Example:

```
abstract class Payment
{
    public abstract void MakePayment();
}

class CreditCardPayment : Payment
{
    public override void MakePayment()
    {
        Console.WriteLine("Payment done through Credit Card");
    }
}

class PayPalPayment : Payment
{
    public override void MakePayment()
    {
        Console.WriteLine("Payment done through PayPal");
    }
}
```

Usage:

```
Payment payment = new CreditCardPayment();
payment.MakePayment();
```

```
payment = new PayPalPayment();
payment.MakePayment();
```

Note: Different classes respond differently to the same method call!

50. What is encapsulation? How is it implemented in C#?

Answer:

Encapsulation is hiding internal data and exposing only necessary parts via public properties/methods.

Protects data integrity.

Example:

```
class Employee
{
    private int salary;

    public int Salary
    {
        get { return salary; }
        set
        {
            if (value > 0)
                salary = value;
        }
    }
}
```

Note: No direct access to salary.
Note: Data is controlled via property logic.

51. What is the difference between early binding and late binding in C#?

Answer:

Early Binding: Method calls are resolved at compile time. (static typing)

Late Binding: Method calls are resolved at runtime. (dynamic typing or reflection)

Example of early binding:

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

Usage:

```
Calculator calc = new Calculator();
int result = calc.Add(3, 5);
```

Note: Compiler knows the method Add() at compile time.

Example of late binding using dynamic:

```
dynamic obj = new Calculator();
int result = obj.Add(4, 6);
```

Note: Method resolution happens at runtime.
Note: Useful when method availability is dynamic (ex: COM, plugins, reflection).

52. What is the difference between dynamic and var in C#?

Answer:

Feature	var	dynamic
When resolved	Compile-time	Runtime
Flexibility	Type cannot change after assigned	Type can change during execution
Errors caught	At compile-time	At runtime

Example:

```
var name = "Karthik";
// name = 100; // ❌ Compile-time Error

dynamic obj = "Karthik";
obj = 100; // Note:  No error until runtime
```

Note: Use dynamic when you want maximum flexibility, but lose type safety.

53. What is a sealed class in C#? When should you use it?

Answer:

A sealed class cannot be inherited.

Use sealed classes when you want to restrict modification and ensure final implementation.

Example:

```
sealed class Logger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

// class ExtendedLogger : Logger ❌ Error
```

Note: Typical usage: Helper classes, utility classes, or security reasons.

54. Explain shallow copy vs deep copy in C#.

Answer:

Shallow Copy: Copies only reference addresses (not new objects).

Deep Copy: Copies entire new objects recursively.

Example of shallow copy using MemberwiseClone():

```
class Person
{
    public string Name;
    public Person Clone()
    {
        return (Person)this.MemberwiseClone();
    }
}
```

Note: If you modify the reference type field, both original and clone are affected!

Note: Deep copy means you create separate copies of reference objects manually.

55. How do you implement a singleton class in C#?

Answer:
Singleton ensures only one instance of a class is created.

Implementation:

```
class Singleton
{
    private static Singleton instance;
    private static readonly object locker = new object();

    private Singleton() {}

    public static Singleton GetInstance()
    {
        if (instance == null)
        {
            lock (locker)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Note: Ensures thread-safe, lazy initialization, and single instance.

56. What is the difference between override and new keyword in C#?

Answer:

Feature	override	new
Purpose	Replace base class method logic	Hide base class method
Runtime behavior	Polymorphism (dynamic dispatch)	No polymorphism
When used	Base method must be virtual Base	method need not be virtual

Example of override:

```
class Base
{
    public virtual void Show()
    {
        Console.WriteLine("Base Show");
    }
}

class Derived : Base
{
    public override void Show()
    {
        Console.WriteLine("Derived Show");
    }
}
```

Example of new:

```
class DerivedNew : Base
{
    public new void Show()
    {
        Console.WriteLine("Derived New Show");
    }
}
```

Note: Use override for polymorphism, new for hiding methods.

57. How do you create a private constructor? When do you use it?

Answer:

Private constructors are used in Singleton Pattern or Utility classes.

Prevents object creation from outside.

Example:

```
class DatabaseConnection
{
    private static DatabaseConnection _instance = new DatabaseConnection();

    private DatabaseConnection() {}

    public static DatabaseConnection GetInstance()
    {
        return _instance;
    }
}
```

Note: You control the instance creation manually.

58. Can a class implement multiple interfaces with same method names?

Answer:
Note: Yes, but you have to use explicit implementation to resolve ambiguity.

Example:

```
interface IReadable
{
    void Display();
}

interface IWritable
{
    void Display();
}

class Document : IReadable, IWritable
{
    void IReadable.Display()
    {
        Console.WriteLine("Reading Document");
    }

    void IWritable.Display()
    {
        Console.WriteLine("Writing Document");
    }
}
```

Usage:

```
IReadable readDoc = new Document();
readDoc.Display();

IWritable writeDoc = new Document();
writeDoc.Display();
```

Note: Perfect separation of behaviors even if method names match.

59. What is the difference between interface and abstract class with examples?

Answer:

Feature	Interface	Abstract Class
Methods	Only signatures (until C#8)	Can have implementations
Fields	No fields	Can have fields and properties
Inheritance	Multiple interfaces supported	Only single class inheritance
Constructor	No constructor allowed	Constructors allowed

Example Interface:

```
interface IVehicle
{
    void Start();
}
```

Example Abstract Class:

```
abstract class Vehicle
{
    public abstract void Start();
    public void Horn()
    {
        Console.WriteLine("Beep Beep");
    }
}
```

Note: Use interface for behavior contract,
Note: Use abstract class for shared behavior + contract.

60. What happens if you don't provide implementation for all interface methods?

Answer:

If a class does not implement all interface methods,

The compiler throws an error unless the class is declared abstract.

Example:

```
interface IAnimal
{
    void Eat();
    void Sleep();
}

class Dog : IAnimal
{
    public void Eat()
    {
        Console.WriteLine("Dog eating");
    }

// Missing Sleep() => Compiler Error }
```

Note: Always implement all interface methods unless you make the class abstract.

61. Can an interface inherit another interface? Can a class inherit multiple interfaces?

Answer:
Note: Yes, interfaces can inherit other interfaces.
Note: Yes, a class can implement multiple interfaces.

Example:

```
interface IFirst
{
    void MethodA();
}

interface ISecond : IFirst
{
    void MethodB();
}

class Implementation : ISecond
{
    public void MethodA()
    {
        Console.WriteLine("MethodA executed");
    }

    public void MethodB()
    {
        Console.WriteLine("MethodB executed");
    }
}
```

Note: Interfaces can extend multiple interfaces too: interface ICombined : IFirst, IOther.

62. What is the difference between method overloading and method overriding?

Answer:

Feature	Method Overloading	Method Overriding
Purpose	Same name, different parameters	Modify inherited method behavior
Compile time/runtime	Compile-time polymorphism	Runtime polymorphism
Keyword used	None	override, virtual, abstract

Example of overloading:

```
class Calculator
{
    public int Add(int a, int b) => a + b;
    public float Add(float a, float b) => a + b;
}
```

Example of overriding:

```
class Parent
{
    public virtual void Show()
    {
        Console.WriteLine("Parent Show");
    }
}

class Child : Parent
{
    public override void Show()
    {
        Console.WriteLine("Child Show");
    }
}
```

Note: Overloading: same method name, different arguments.
Note: Overriding: same method signature, different behavior.

63. What is boxing and unboxing in C#?

Answer:
Note: Boxing: Converting a value type (e.g., int) into an object.
Note: Unboxing: Extracting the value type from an object.

Example:

```
int number = 10;
object obj = number; // Boxing
int result = (int)obj; // Unboxing
```

Note: Boxing moves value type to heap (object type),
Note: Unboxing extracts back the original value.

64. What is an indexer in C#?

Answer:

Indexers allow an object to be indexed like an array.

Syntax similar to array but at the object level.

Example:

```
class SampleCollection
{
    private string[] data = new string[5];

    public string this[int index]
    {
        get { return data[index]; }
        set { data[index] = value; }
    }
}
```

Usage:

```
SampleCollection collection = new SampleCollection();
collection[0] = "Hello";
Console.WriteLine(collection[0]);
```

Note: Indexers help in custom array-like behavior for your own classes.

65. What is a delegate in C#? How is it different from an event?

Answer:

A delegate is a function pointer (type-safe).

An event is a wrapper over a delegate to restrict direct invocation.

Example:

```
delegate void Notify();

class Process
{
    public static void Task()
    {
        Console.WriteLine("Process Started");
    }
}
```

Usage:

```
Notify notify = Process.Task;
notify();
```

Note: Delegates point to methods.
Note: Events control access so that only subscribers can trigger.

Example Event:

```
class Alarm
{
    public event Notify Ring;

    public void Trigger()
    {
        Ring?.Invoke();
    }
}
```

Note: Events restrict outsiders from accidentally invoking the delegate.

66. What is a multicast delegate in C#?

Answer:

Note: A multicast delegate points to multiple methods.

Note: When invoked, it calls all the methods sequentially.

Example:

```
delegate void Notify();

class MulticastExample
{
    public static void Method1()
    {
        Console.WriteLine("Method1 Called");
    }

    public static void Method2()
    {
        Console.WriteLine("Method2 Called");
    }
}
```

Usage:

```
Notify notify = MulticastExample.Method1;
notify += MulticastExample.Method2;
notify();
```

Note: Both Method1 and Method2 get called when notify() is invoked.

67. What are anonymous methods in C#?

Answer:

Anonymous Methods are methods without a name.

They are assigned directly to a delegate.

Example:

```
delegate void Notify(string message);

class Program
{
    static void Main()
    {
        Notify notify = delegate(string msg)
        {
            Console.WriteLine("Notification: " + msg);
        };

        notify("Task Completed!");
    }
}
```

Note: Anonymous methods are helpful for small inline tasks without creating separate methods.

68. What are lambda expressions in C#?

Answer:

Lambda expressions are shorthand for anonymous methods.

Syntax: (parameters) => expression

Example:

```
delegate int Square(int num);

class Program
{
    static void Main()
    {
        Square square = x => x * x;
        Console.WriteLine(square(5));
    }
}
```

Note: Lambda makes the code cleaner and readable.

69. What is the difference between Func, Action, and Predicate?

Answer:

Type	Signature	Return type
Func	Takes parameters, returns value	Value (int, string, etc.)
Action	Takes parameters, returns nothing (void)	Void
Predicate	Takes one parameter, returns bool	bool (true/false)

Examples:

```
Func<int, int, int> add = (a, b) => a + b;
Action<string> greet = name => Console.WriteLine("Hello " + name);
Predicate<int> isEven = num => num % 2 == 0;
```

Usage:

```
Console.WriteLine(add(3, 4));
greet("Karthik");
Console.WriteLine(isEven(10));
```

Note: Func: Useful for computations,
Note: Action: Useful for performing actions,
Note: Predicate: Useful for conditions/checks.

70. What is an event handler delegate signature in C#?

Answer:
The standard pattern for .NET event handlers:

Return type: void

Parameters: object sender, EventArgs e

Example:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Example of usage:

```
class Alarm
{
    public event EventHandler Ring;

    public void Trigger()
    {
        if (Ring != null)
        {
            Ring(this, EventArgs.Empty);
        }
    }
}
```

Note: Follows the EventHandler delegate signature in all standard .NET events.

71. What is covariance and contravariance in C#?

Answer:
Note: Covariance: Allows a method to return a more derived type than originally specified.
Note: Contravariance: Allows a method to accept parameters of less derived types.

Example of Covariance with return types:

```
IEnumerable<string> names = new List<string>();
IEnumerable<object> objects = names; // Note:  Covariance allowed because string → object
```

Example of Contravariance with parameters:

```
Action<object> actObject = (obj) => Console.WriteLine(obj);
Action<string> actString = actObject; // Note:  Contravariance allowed because string is a derived type of object
```

Note: Covariance → "Output flexibility"
Note: Contravariance → "Input flexibility"

72. What is the difference between Task and Thread in C#?

Answer:

Feature	Task	Thread
Managed by	Task Scheduler	OS Thread Pool
Lightweight	Yes	No
Use case	For async operations	For manual thread management
Creation cost	Less	More

Example:

```
Task.Run(() => Console.WriteLine("Task Running"));
```

vs

```
Thread thread = new Thread(() => Console.WriteLine("Thread Running"));
```

```
thread.Start();
```

Note: Prefer Task for modern, scalable, async operations.

73. What is deadlock? How do you avoid it in C#?

Answer:

Note: Deadlock occurs when two or more threads are waiting for each other’s resources, causing an infinite wait.

Example of Deadlock:

```
object lock1 = new object();
object lock2 = new object();

Thread t1 = new Thread(() =>
{
    lock (lock1)
    {
        Thread.Sleep(1000);
        lock (lock2) { }
    }
});

Thread t2 = new Thread(() =>
{
    lock (lock2)
    {
        lock (lock1) { }
    }
});

t1.Start();
t2.Start();
```

Note: Avoid deadlocks by:

Always locking resources in the same order.

Using timeout patterns (e.g., Monitor.TryEnter()).

74. What is async and await in C#?

Answer:

async enables a method to be asynchronous (non-blocking).

await pauses method execution until awaited Task completes.

Example:

```
async Task<int> GetNumberAsync()
{
    await Task.Delay(1000);
    return 5;
}
```

Usage:

```
var result = await GetNumberAsync();
Console.WriteLine(result);
```

Note: Helps improve scalability without blocking threads.

75. Explain the difference between IEnumerable and IQueryable.

Answer:

Feature	IEnumerable	IQueryable
Where evaluated	In memory (client-side)	At database (server-side)
When evaluated	Deferred execution	Deferred execution
Suitable for	In-memory collections (List, Array)	Remote data sources (Database)

Example:

```
IEnumerable<int> localQuery = new List<int> {1,2,3,4}.Where(x => x > 2);
```

vs

```
IQueryable<int> dbQuery = dbContext.Employees.Where(e => e.Id > 100);
```

Note: Use IEnumerable for in-memory data,
Note: Use IQueryable for database-efficient queries.

76. What is a static class in C#? Can it have constructors?

Answer:
Note: A static class:

Cannot be instantiated.

Can only have static members.

Can have a static constructor (executed once when class is loaded).

Example:

```
static class MathHelper
{
    static MathHelper()
    {
        Console.WriteLine("Static Constructor Called");
    }

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

Usage:

```
Console.WriteLine(MathHelper.Add(3, 4));
```

Note: Static classes are used for utility/helper methods.

77. What is reflection in C#? How can you use it?

Answer:
Note: Reflection is the ability to inspect metadata (types, methods, properties) at runtime.

Example:

```
Type type = typeof(string);
Console.WriteLine("Type Name: " + type.Name);
foreach (var method in type.GetMethods())
{
    Console.WriteLine("Method: " + method.Name);
}
```

Note: Reflection is used for:

Dynamic loading,

Plugin architectures,

Inspecting attributes.

Be careful: Reflection is slow compared to normal access.

78. What is the difference between early binding and late binding using reflection?

Answer:

Feature	Early Binding	Late Binding (Reflection)
Method known	At compile time	Only at runtime
Speed	Fast	Slower
Flexibility	Rigid (fixed types)	Very flexible (dynamic types)

Example Early Binding:

```
string text = "Hello";
Console.WriteLine(text.ToUpper());
```

Example Late Binding:

```
object textObj = "Hello";
Type t = textObj.GetType();
MethodInfo method = t.GetMethod("ToUpper");
object result = method.Invoke(textObj, null);
Console.WriteLine(result);
```

Note: Reflection allows calling methods without knowing their name at compile time.

79. How do you create a custom attribute in C#?

Answer:
Note: Custom attributes allow you to attach metadata to classes, methods, properties.

Example:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class AuthorAttribute : Attribute
```

```
{
    public string Name { get; set; }
    public AuthorAttribute(string name)
    {
        Name = name;
    }
}
```

Usage:

```
[Author("Karthik")]
class MyClass
{
    public void DoWork() { }
}
```

Note: Attributes can later be retrieved via Reflection.

80. How do you read custom attributes using reflection?

Answer:
Note: After defining custom attributes, you can read them dynamically.

Example:

```
Type type = typeof(MyClass);
object[] attributes = type.GetCustomAttributes(typeof(AuthorAttribute), true);
foreach (AuthorAttribute attr in attributes)
{
    Console.WriteLine("Author: " + attr.Name);
}
```

Note: Reflection fetches metadata and enables dynamic behaviors.

81. What is a partial class in C#?

Answer:
Note: A partial class allows a class to be split across multiple files.
Note: At compile time, all parts are combined into a single class.

Example:

First file (Person1.cs):

```
public partial class Person
{
    public string FirstName;
}
```

Second file (Person2.cs):

```
public partial class Person
{
    public string LastName;
}
```

Usage:

```
Person p = new Person();
p.FirstName = "Karthik";
```

```
p.LastName = "Muthukrishnan";
```

Note: Partial classes are helpful for:

Large classes,

Auto-generated code (e.g., Designer files in WinForms).

82. What is a partial method in C#?

Answer:
Note: A partial method is a special method inside a partial class:

It can be optionally implemented.

If not implemented, the compiler removes its call (no error, no code).

Example:

In first file:

```
public partial class Demo
{
    partial void Log(string message);

    public void Run()
    {
        Log("Running Demo");
    }
}
```

In second file:

```
public partial class Demo
{
    partial void Log(string message)
    {
        Console.WriteLine("Log: " + message);
    }
}
```

Note: Partial methods are lightweight hooks for optional logic.

83. What is serialization in C#? What are types of serialization?

Answer:
Note: Serialization is the process of converting an object into a format that can be persisted (file, database) or transmitted (over network).

Types of Serialization:

1. Binary Serialization (compact, .bin file)

2. XML Serialization (human-readable)

3. JSON Serialization (modern, APIs)

4. SOAP Serialization (older web services)

Example JSON serialization:

```
using System.Text.Json;

Employee emp = new Employee { Id = 1, Name = "Karthik" };
string jsonString = JsonSerializer.Serialize(emp);
Console.WriteLine(jsonString);
```

Note: Serialization is important for data storage, API communication, and distributed systems.

84. What is the difference between shallow copy and deep copy in serialization?

Answer:

Feature	Shallow Copy	Deep Copy
What is copied	Only references are copied	Entire objects are copied
Effect	Changes affect both objects	Independent copies
How to achieve	MemberwiseClone()	Serialization/Manual copy

Example Deep Copy using JSON:

```
string json = JsonSerializer.Serialize(originalObject);
var deepCopy = JsonSerializer.Deserialize<Employee>(json);
```

Note: Deep copy ensures full independence between copies.

85. How do you create a custom exception in C#?

Answer:
Note: Create a class that inherits from Exception.

Example:

```
public class InvalidAgeException : Exception
{
    public InvalidAgeException(string message) : base(message)
    {
    }
}
```

Usage:

```
int age = 15;
if (age < 18)
{
    throw new InvalidAgeException("Age must be 18 or older.");
}
```

Note: Custom exceptions are useful for domain-specific error handling.

86. What is the difference between String and StringBuilder in C#?

Answer:

Feature	String	StringBuilder
Mutability	Immutable (modifications create new object)	Mutable (modifications happen in-place)
Performance	Poor for frequent modifications	High performance for modifications
Usage	Small and simple text	Large and dynamic text changes

Example of String:

```
string str = "Hello";
str += " World";
Console.WriteLine(str);
```

Example of StringBuilder:

```
StringBuilder sb = new StringBuilder("Hello");
sb.Append(" World");
Console.WriteLine(sb.ToString());
```

Note: Prefer StringBuilder for heavy text modifications like loops and concatenations.

87. What are extension methods in C#?

Answer:
Note: Extension methods allow you to "add" methods to existing types without modifying them.

Example:

```
public static class StringExtensions
{
    public static int WordCount(this string str)
    {
        return str.Split(' ').Length;
    }
}
```

Usage:

```
string text = "Hello from Karthik";
Console.WriteLine(text.WordCount());
```

Note: Extension methods make your code more readable and fluent.

88. What is the difference between IEnumerable and IEnumerator in C#?

Answer:

Feature	IEnumerable	IEnumerator
Purpose	Collection that can be iterated	Cursor that iterates the collection
Method	GetEnumerator()	MoveNext(), Current, Reset()
Used with	foreach loops	Inside foreach internals

Example:

```
class Numbers : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return 1;
        yield return 2;
        yield return 3;
    }
}
```

Usage:

```
Numbers numbers = new Numbers();
foreach (int n in numbers)
{
    Console.WriteLine(n);
}
```

Note: IEnumerable exposes enumeration behavior,
Note: IEnumerator controls the navigation logic.

89. What is Dependency Injection (DI) in C#?

Answer:
Note: Dependency Injection is a design pattern to inject an object's dependencies from outside instead of creating them inside the class.

Three common types:

1. Constructor Injection

2. Setter Injection

3. Method Injection

Example - Constructor Injection:

```
class Service
{
    public void Serve()
    {
        Console.WriteLine("Service Called");
    }
}

class Client
{
    private Service _service;

    public Client(Service service)
    {
        _service = service;
    }

    public void Start()
    {
        _service.Serve();
    }
}
```

Usage:

```
Service service = new Service();
Client client = new Client(service);
client.Start();
```

Note: DI improves testability, decoupling, and maintainability.

90. What is the difference between Singleton and Static Class?

Answer:

Feature	Singleton	Static Class
Object creation	Only one instance allowed	No instance allowed
Memory management	Created on demand	Always loaded in memory
Inheritance	Can implement interfaces	Cannot inherit interfaces

Example Singleton:

```
class Singleton
{
    private static Singleton _instance;
    private Singleton() {}

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
                _instance = new Singleton();
            return _instance;
        }
    }
}
```

Note: Singleton gives controlled object creation,
Note: Static class gives grouped static methods without object management.

91. What is the use of 'yield' keyword in C#?

Answer:
Note: The yield keyword allows you to iterate items one-by-one, maintaining the state between calls without creating a collection.

Example:

```
public static IEnumerable<int> GetNumbers()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
```

Usage:

```
foreach (var number in GetNumbers())
{
    Console.WriteLine(number);
}
```

Note: yield return simplifies creating custom iterators without manual IEnumerator implementation.

92. What is the 'lock' statement in C#?

Answer:
Note: The lock statement ensures that a critical section of code is executed by only one thread at a time.

Example:

```
private static readonly object _lockObject = new object();

public void CriticalMethod()
{
    lock (_lockObject)
    {
        // Only one thread can execute here at a time
        Console.WriteLine("Critical Section Accessed");
    }
}
```

Note: lock prevents race conditions in multi-threaded environments.

93. What is the difference between 'const', 'readonly', and 'static readonly'?

Answer:

Feature	const	readonly	static readonly
Value Set	At compile-time	At runtime (in constructor)	At runtime (only once in static constructor)
Modifiability	No	No	No
Context	Static by default	Instance-level or Static	Static

Example of const:

```
public const int MaxItems = 100;
```

Example of readonly:

```
public readonly int CreatedAt = DateTime.Now.Year;
```

Example of static readonly:

```
public static readonly int StaticValue;

static ClassName()
{
    StaticValue = 10;
}
```

Note: Use:

const for pure compile-time constants,

readonly for instance constants,

static readonly for class-level constants set at runtime.

94. What is method hiding in C#?

Answer:
Note: Method hiding happens when a derived class defines a new method with the same name as in base class, but does not override it.

Use new keyword to indicate method hiding.

Example:

```
class Base
{
    public void Display()
    {
        Console.WriteLine("Base Display");
    }
}

class Derived : Base
{
    public new void Display()
    {
        Console.WriteLine("Derived Display");
    }
}
```

Usage:

```
Base b = new Derived();
b.Display(); // Calls Base.Display (not Derived)
```

Note: Method hiding does not replace base method behavior unless explicitly casted.

95. What is a sealed class in C#?

Answer:
Note: A sealed class cannot be inherited.

Note: Use sealed keyword to prevent derivation.

Example:

```
sealed class FinalClass
{
    public void Show()
    {
        Console.WriteLine("Final Class Method");
    }
}
```

Note: Attempting to inherit from a sealed class will cause compile-time error.

Note: Sealed classes are used for:

Security (prevent overriding critical behavior),

Performance (JIT can optimize calls).

96. What is an abstract class and how is it different from an interface?

Answer:
Note: Abstract class:

Can have implemented and unimplemented methods.

Can have fields, properties, constructors.

Note: Interface:

Can only have method signatures (until C# 8.0, now can have default methods too).

Cannot have fields.

Example of Abstract Class:

```
abstract class Animal
{
    public abstract void Sound();

    public void Sleep()
    {
        Console.WriteLine("Sleeping...");
    }
}

class Dog : Animal
{
    public override void Sound()
    {
        Console.WriteLine("Bark");
    }
}
```

Note: Use abstract class when:

You want common functionality + force derived classes to implement some behavior.

97. What is a default interface method (C# 8.0 onwards)?

Answer:
Note: In C# 8.0+, interfaces can have default method implementations!

Example:

```
interface ILogger
{
    void Log(string message)
    {
        Console.WriteLine("Logging: " + message);
    }
}
```

Note: Now classes implementing ILogger are not forced to override Log() unless they want to.

Note: Default interface methods make interfaces more flexible without breaking old implementations.

98. What is a Tuple in C#?

Answer:
Note: A Tuple is a lightweight object for grouping multiple values together.

Example:

```
var employee = Tuple.Create(101, "Karthik", "Developer");
Console.WriteLine($"{employee.Item1} - {employee.Item2} - {employee.Item3}");
```

Note: From C# 7 onwards: You can use ValueTuple syntax:

```
(var id, var name, var role) = (101, "Karthik", "Developer");
Console.WriteLine($"{id} - {name} - {role}");
```

Note: Tuples are great for returning multiple values from a method easily.

99. What is the difference between Value Type and Reference Type in C#?

Answer:

Feature	Value Type	Reference Type
Stored in	Stack	Heap
Example types	int, double, struct	class, interface, array, string
Assignment	Copies value	Copies reference
Nullability	Cannot be null (except nullable types)	Can be null

Example of Value Type:

```
int x = 5;
int y = x;
y = 10;
Console.WriteLine(x); // Outputs 5
```

Example of Reference Type:

```
class Person
{
    public string Name;
}

Person p1 = new Person();
p1.Name = "Karthik";
Person p2 = p1;
p2.Name = "Rajesh";
Console.WriteLine(p1.Name); // Outputs "Rajesh"
```

Note: Value types store data,
Note: Reference types store memory address (reference).

100. What is Nullable type in C#?

Answer:
Note: Nullable types allow value types (int, double, etc.) to represent null values.

Example:

```
int? age = null;

if (age.HasValue)
    Console.WriteLine(age.Value);
else
    Console.WriteLine("Age is not set");
```

Note: Nullable types are crucial for:

Database operations,

Optional fields,

Handling missing values safely.

Shortcut syntax:

```
Nullable<int> age1 = 30; // same as int? age1 = 30;
```

Note: Avoids null reference exceptions by checking .HasValue before accessing .Value.