C#/.NET Learning Notes

Compiled for offline study and printing. Start with the Study Guide.

- C#/.NET Exam Preparation Notes

C#/.NET Exam Preparation Notes

Table of Contents

Visual Programming

Definition

Visual programming is a programming paradigm that uses graphical elements rather than text to create programs. It allows developers to manipulate program elements graphically rather than textually.

0

Visual Programming vs Text-Based Programming

Aspect	Visual Programming	Text-Based Programming
Interface	Drag-and-drop, visual components	Text editor, code writing
Learning Curve	Easier for beginners	Steeper learning curve
Flexibility	Limited by available components	Full control over code
Debugging	Visual debugging tools	Text-based debugging
Performance	May have overhead	Direct control over performance
Examples	Visual Studio Designer, Scratch	C#, Java, Python

Examples in .NET

- **Visual Studio Designer**: For WPF, WinForms
- **XAML Designer**: For WPF and UWP applications
- **Blazor Visual Designer**: For web components

```
// Text-based approach
Button myButton = new Button();
myButton.Text = "Click Me";
myButton.Width = 100;
myButton.Height = 30;
myButton.Click += MyButton_Click;

// Visual approach (generated code from designer)
// Designer generates similar code but through visual manipulation
```

Event-Driven Programming

Definition

Event-driven programming is a paradigm where program flow is determined by events such as user actions, sensor outputs, or message passing.

Key Concepts

1. **Events**: Notifications that something has happened
2. **Event Handlers**: Methods that respond to events
3. **Event Loop**: Continuously monitors for events

0

4. **Delegates**: Type-safe function pointers in C#

Example in C#

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        // Subscribe to button click event
        myButton.Click += MyButton_Click;
    }

    // Event handler method
    private void MyButton_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("Button was clicked!");
    }
}

// Custom event example
public class Publisher
{
    // Declare event using delegate
    public event Action<string> OnMessagePublished;

    public void PublishMessage(string message)
    {
        // Raise the event
        OnMessagePublished?.Invoke(message);
    }
}

public class Subscriber
{
    public void Subscribe(Publisher pub)
    {
        // Subscribe to event
0       pub.OnMessagePublished += HandleMessage;
```

```
    }

    private void HandleMessage(string message)
    {
        Console.WriteLine($"Received: {message}");
    }
}
```

.NET Framework Architecture

Key Components

1. Common Language Runtime (CLR)

- **Memory Management**: Automatic garbage collection
- **Type Safety**: Ensures type safety at runtime
- **Exception Handling**: Unified exception handling
- **Thread Management**: Manages application threads
- **Security**: Code access security

2. Base Class Library (BCL)

- **System.Object**: Root of all .NET types
- **Collections**: Generic and non-generic collections
- **I/O**: File and stream operations
- **Networking**: Network communication classes

3. .NET Framework vs .NET Core vs .NET 5+

Feature	.NET Framework	.NET Core	.NET 5+
Platform	Windows only	Cross-platform	Cross-platform
Open Source	No	Yes	Yes
Performance	Good	Better	Best
Deployment	Framework dependent	Self-contained options	Self-contained options

Architecture Diagram

0

```
+-------------------------------------+
|        Your Application             |
+-------------------------------------+
|    Base Class Library (BCL)         |
+-------------------------------------+
|  Common Language Runtime (CLR)      |
+-------------------------------------+
|       Operating System              |
+-------------------------------------+
```

Layer Description:

- **Your Application**: C# code you write (WPF, Console, Web, etc.)
- **Base Class Library**: Built-in .NET classes (System.*, Collections, etc.)
- **Common Language Runtime**: Memory management, type safety, execution
- **Operating System**: Windows, Linux, macOS platform services

RAD Tools

Rapid Application Development (RAD)

RAD is a software development methodology that prioritizes rapid prototyping and quick feedback over long planning cycles.

RAD Tools in .NET Ecosystem

1. Visual Studio

- **IntelliSense**: Code completion and suggestions
- **Debugging Tools**: Breakpoints, watch windows
- **Designer Support**: Visual designers for UI
- **Project Templates**: Pre-built project structures

2. Visual Studio Code

- **Extensions**: Rich ecosystem of extensions
- **Integrated Terminal**: Built-in command line
- **Git Integration**: Version control support

0 **3. JetBrains Rider**

- **Advanced Refactoring**: Powerful code transformation tools
- **Unit Testing**: Integrated test runner
- **Database Tools**: Built-in database support

Example: Quick WPF Application

```
// Program.cs - Entry point
using System;
using System.Windows;

namespace QuickApp
{
    public partial class App : Application
    {
        [STAThread]
        public static void Main()
        {
            App app = new App();
            app.Run(new MainWindow());
        }
    }
}

// MainWindow.xaml
<Window x:Class="QuickApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="Quick RAD App" Height="200" Width="300">
    <StackPanel Margin="10">
        <TextBox x:Name="InputText" Margin="5"/>
        <Button Content="Process" Click="ProcessButton_Click"
Margin="5"/>
        <TextBlock x:Name="OutputText" Margin="5"/>
    </StackPanel>
</Window>

// MainWindow.xaml.cs
public partial class MainWindow : Window
{
    public MainWindow()
    {
```

```
        InitializeComponent();
    }

    private void ProcessButton_Click(object sender, RoutedEventArgs e)
    {
        OutputText.Text = $"Processed: {InputText.Text.ToUpper()}";
    }
}
```

Type Conversion

Types of Conversion

1. Implicit Conversion (Automatic)

```
int intValue = 42;
long longValue = intValue;    // Implicit conversion (safe)
double doubleValue = intValue; // Implicit conversion (safe)

// Implicit conversions for numeric types
byte b = 100;
short s = b;     // byte to short
int i = s;       // short to int
long l = i;      // int to long
float f = l;     // long to float
double d = f;    // float to double
```

2. Explicit Conversion (Manual)

```
double doubleValue = 42.7;
int intValue = (int)doubleValue;  // Explicit cast (data loss possible)

// Explicit conversions
long longValue = 123456789;
int intValue2 = (int)longValue;   // Potential overflow

// Using Convert class
string numberString = "123";
```

0

```
int converted = Convert.ToInt32(numberString);
double convertedDouble = Convert.ToDouble("123.45");
```

3. Boxing and Unboxing

```
// Boxing: Value type to reference type
int value = 42;
object boxed = value;         // Boxing

// Unboxing: Reference type to value type
object boxedValue = 42;
int unboxed = (int)boxedValue; // Unboxing

// Performance consideration
List<object> mixedList = new List<object>();
mixedList.Add(42);           // Boxing occurs
mixedList.Add("Hello");      // No boxing (already reference type)
```

4. Parse and TryParse Methods

```
// Parse (throws exception on failure)
string input = "123";
int parsed = int.Parse(input);

// TryParse (returns false on failure)
string userInput = "abc";
if (int.TryParse(userInput, out int result))
{
    Console.WriteLine($"Parsed: {result}");
}
else
{
    Console.WriteLine("Invalid input");
}

// DateTime parsing
string dateString = "2023-12-25";
if (DateTime.TryParse(dateString, out DateTime parsedDate))
{
```

```
    Console.WriteLine($"Date: {parsedDate:yyyy-MM-dd}");
}
```

Structures vs Enumerations

Structures (struct)

Definition

Structures are value types that can contain data members and function members.

Key Characteristics

- **Value Type**: Stored on stack
- **No Inheritance**: Cannot inherit from other types
- **Immutable Recommended**: Should be immutable for best practices
- **Default Constructor**: Always available

```
public struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public double DistanceFromOrigin()
    {
        return Math.Sqrt(X * X + Y * Y);
    }

    public override string ToString()
    {
        return $"({X}, {Y})";
    }
}
```

0

```
// Usage
Point p1 = new Point(3, 4);
Point p2 = new Point(); // Default constructor (0, 0)
Console.WriteLine(p1.DistanceFromOrigin()); // 5
```

Enumerations (enum)

Definition

Enumerations define a set of named constants of the underlying integral numeric type.

Key Characteristics

- **Named Constants**: Improve code readability
- **Type Safe**: Prevents invalid values
- **Underlying Type**: Default is int, can be changed
- **Flags Support**: Can be combined using bitwise operations

```
// Basic enumeration
public enum OrderStatus
{
    Pending,      // 0
    Processing,   // 1
    Shipped,      // 2
    Delivered,    // 3
    Cancelled     // 4
}

// Custom underlying type and values
public enum Priority : byte
{
    Low = 1,
    Medium = 5,
    High = 10,
    Critical = 20
}

// Flags enumeration
[Flags]
```

```
public enum FilePermissions
{
    None = 0,
    Read = 1,
    Write = 2,
    Execute = 4,
    ReadWrite = Read | Write,
    All = Read | Write | Execute
}

// Usage examples
OrderStatus status = OrderStatus.Processing;

// Enum methods
string statusName = status.ToString();              // "Processing"
OrderStatus parsed = Enum.Parse<OrderStatus>("Shipped");
bool isValid = Enum.IsDefined(typeof(OrderStatus), 3); // true

// Flags usage
FilePermissions permissions = FilePermissions.Read |
FilePermissions.Write;
bool canRead = permissions.HasFlag(FilePermissions.Read);    // true
bool canExecute = permissions.HasFlag(FilePermissions.Execute); // false
```

Comparison Table

Feature	struct	enum
Purpose	Data containers	Named constants
Type	Value type	Value type (integral)
Memory	Stack	Constant value
Inheritance	No inheritance	No inheritance
Methods	Can have methods	Only predefined methods
Mutability	Should be immutable	Immutable

Collections

Generic Collections (Recommended)

0

1. List

```
// Dynamic array implementation
List<string> names = new List<string>();
names.Add("Alice");
names.Add("Bob");
names.AddRange(new[] { "Charlie", "David" });

// Access and modification
names[0] = "Alice Smith";
names.Insert(1, "Betty");
names.Remove("Bob");
names.RemoveAt(2);

// Iteration
foreach (string name in names)
{
    Console.WriteLine(name);
}

// LINQ operations
var longNames = names.Where(n => n.Length > 5).ToList();
```

2. Dictionary<TKey, TValue>

```
Dictionary<string, int> ages = new Dictionary<string, int>
{
    ["Alice"] = 30,
    ["Bob"] = 25
};

// Adding and accessing
ages.Add("Charlie", 35);
ages["David"] = 28; // Add or update

// Safe access
if (ages.TryGetValue("Alice", out int aliceAge))
{
    Console.WriteLine($"Alice is {aliceAge} years old");
```

0

```
}

// Iteration
foreach (KeyValuePair<string, int> kvp in ages)
{
    Console.WriteLine($"{kvp.Key}: {kvp.Value}");
}
```

3. HashSet

```
HashSet<string> uniqueNames = new HashSet<string>();
uniqueNames.Add("Alice");
uniqueNames.Add("Bob");
uniqueNames.Add("Alice"); // Duplicate ignored

// Set operations
HashSet<string> otherNames = new HashSet<string> { "Bob", "Charlie" };
uniqueNames.UnionWith(otherNames);        // Union
uniqueNames.IntersectWith(otherNames);    // Intersection
bool contains = uniqueNames.Contains("Alice");
```

4. Queue and Stack

```
// Queue (FIFO - First In, First Out)
Queue<string> taskQueue = new Queue<string>();
taskQueue.Enqueue("Task 1");
taskQueue.Enqueue("Task 2");
string nextTask = taskQueue.Dequeue(); // "Task 1"

// Stack (LIFO - Last In, First Out)
Stack<string> undoStack = new Stack<string>();
undoStack.Push("Action 1");
undoStack.Push("Action 2");
string lastAction = undoStack.Pop(); // "Action 2"
```

Non-Generic Collections (Legacy)

ArrayList

0

```
// Non-generic - boxing/unboxing occurs
ArrayList list = new ArrayList();
list.Add(42);        // Boxing
list.Add("Hello");   // Reference type
int value = (int)list[0]; // Unboxing + casting required

// Problems:
// 1. No compile-time type checking
// 2. Boxing/unboxing performance overhead
// 3. Runtime errors possible
```

Hashtable

```
Hashtable table = new Hashtable();
table["key1"] = "value1";
table[42] = "number key";  // Any type as key

// Type casting required
string value = (string)table["key1"];
```

Collection Interfaces

```
// IEnumerable<T> - Basic iteration
public void ProcessItems(IEnumerable<string> items)
{
    foreach (string item in items)
    {
        Console.WriteLine(item);
    }
}

// ICollection<T> - Add/Remove operations
public void ModifyCollection(ICollection<string> items)
{
    items.Add("New Item");
    items.Remove("Old Item");
    Console.WriteLine($"Count: {items.Count}");
}
```

0

```
// IList<T> - Indexed access
public void AccessByIndex(IList<string> items)
{
    items[0] = "First Item";
    items.Insert(1, "Second Item");
}
```

Regular Expressions

Definition

Regular expressions (regex) are patterns used to match character combinations in strings. They provide a powerful way to search, replace, and validate text.

Basic Syntax

Metacharacters

- `.` - Any character except newline
- `*` - Zero or more occurrences
- `+` - One or more occurrences
- `?` - Zero or one occurrence
- `^` - Start of string
- `$` - End of string
- `|` - OR operator
- `[]` - Character class
- `()` - Grouping

C# Regex Implementation

```
using System.Text.RegularExpressions;

// Basic pattern matching
string text = "The quick brown fox jumps over the lazy dog";
string pattern = @"quick|lazy";
Regex regex = new Regex(pattern);
```

0

```
// Check if pattern exists
bool hasMatch = regex.IsMatch(text); // true

// Find all matches
MatchCollection matches = regex.Matches(text);
foreach (Match match in matches)
{
    Console.WriteLine($"Found: '{match.Value}' at position
{match.Index}");
}

// Replace text
string replaced = regex.Replace(text, "REPLACED");
Console.WriteLine(replaced);
```

Common Patterns

Email Validation

```
string emailPattern = @"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]
{2,}$";
Regex emailRegex = new Regex(emailPattern);

string[] emails = { "user@example.com", "invalid.email",
"test@domain.co.uk" };
foreach (string email in emails)
{
    bool isValid = emailRegex.IsMatch(email);
    Console.WriteLine($"{email}: {(isValid ? "Valid" : "Invalid")}");
}
```

Phone Number Extraction

```
string phonePattern = @"\b\d{3}-\d{3}-\d{4}\b";
string text = "Call me at 555-123-4567 or 555-987-6543";

MatchCollection phoneMatches = Regex.Matches(text, phonePattern);
foreach (Match match in phoneMatches)
```

0

```
{
    Console.WriteLine($"Phone: {match.Value}");
}
```

Groups and Capturing

```
string namePattern = @"(\w+)\s+(\w+)"; // First name, Last name
string input = "John Doe, Jane Smith, Bob Johnson";

MatchCollection nameMatches = Regex.Matches(input, namePattern);
foreach (Match match in nameMatches)
{
    string firstName = match.Groups[1].Value;
    string lastName = match.Groups[2].Value;
    Console.WriteLine($"Name: {firstName} {lastName}");
}
```

Advanced Features

Regex Options

```
// Case-insensitive matching
Regex regex1 = new Regex(@"hello", RegexOptions.IgnoreCase);

// Multiline mode
Regex regex2 = new Regex(@"^Start", RegexOptions.Multiline);

// Compiled regex for performance
Regex regex3 = new Regex(@"\d+", RegexOptions.Compiled);
```

Named Groups

```
string pattern = @"(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})";
string dateText = "Today is 2023-12-25";

Match match = Regex.Match(dateText, pattern);
if (match.Success)
{
```

0

```
    string year = match.Groups["year"].Value;
    string month = match.Groups["month"].Value;
    string day = match.Groups["day"].Value;
    Console.WriteLine($"Date: {year}/{month}/{day}");
}
```

Polymorphism

Definition

Polymorphism allows objects of different types to be treated as objects of a common base type, while maintaining their specific behavior.

Types of Polymorphism

1. Method Overloading (Compile-time Polymorphism)

```
public class Calculator
{
    // Same method name, different parameters
    public int Add(int a, int b)
    {
        return a + b;
    }

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

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

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

```
}

// Usage
Calculator calc = new Calculator();
int result1 = calc.Add(5, 3);           // Calls int version
double result2 = calc.Add(5.5, 3.2);    // Calls double version
int result3 = calc.Add(1, 2, 3);        // Calls three-parameter version
string result4 = calc.Add("Hello", " World"); // Calls string version
```

2. Method Overriding (Runtime Polymorphism)

```
// Base class
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("The animal makes a sound");
    }

    public virtual void Move()
    {
        Console.WriteLine("The animal moves");
    }
}

// Derived classes
public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("The dog barks: Woof!");
    }

    public override void Move()
    {
        Console.WriteLine("The dog runs");
    }
}
```

0

```
public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("The cat meows: Meow!");
    }

    public override void Move()
    {
        Console.WriteLine("The cat prowls");
    }
}

// Polymorphic usage
Animal[] animals = { new Dog(), new Cat(), new Animal() };
foreach (Animal animal in animals)
{
    animal.MakeSound(); // Calls appropriate overridden method
    animal.Move();      // Runtime determines which method to call
}
```

Abstract Methods and Classes

```
public abstract class Shape
{
    // Abstract method - must be implemented by derived classes
    public abstract double CalculateArea();

    // Virtual method - can be overridden
    public virtual void Display()
    {
        Console.WriteLine($"Area: {CalculateArea():F2}");
    }

    // Regular method - inherited as-is
    public void PrintInfo()
    {
        Console.WriteLine("This is a shape");
    }
```

0

```
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public override double CalculateArea()
    {
        return Width * Height;
    }

    public override void Display()
    {
        Console.WriteLine($"Rectangle - Width: {Width}, Height: {Height},
Area: {CalculateArea():F2}");
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

0

Interface Polymorphism

```
public interface IDrawable
{
    void Draw();
    void Resize(double factor);
}

public interface IColorable
{
    string Color { get; set; }
    void ChangeColor(string newColor);
}

public class Button : IDrawable, IColorable
{
    public string Color { get; set; } = "Gray";
    public string Text { get; set; }

    public void Draw()
    {
        Console.WriteLine($"Drawing {Color} button with text: {Text}");
    }

    public void Resize(double factor)
    {
        Console.WriteLine($"Resizing button by factor: {factor}");
    }

    public void ChangeColor(string newColor)
    {
        Color = newColor;
        Console.WriteLine($"Button color changed to: {newColor}");
    }
}

// Polymorphic usage with interfaces
IDrawable[] drawables = { new Button { Text = "OK" }, new Button { Text =
"Cancel" } };
```

0

```
foreach (IDrawable drawable in drawables)
{
    drawable.Draw();
    drawable.Resize(1.5);
}
```

Abstract Classes vs Interfaces

Abstract Classes

Definition

An abstract class is a class that cannot be instantiated and may contain both abstract and concrete members.

Key Characteristics

- Cannot be instantiated directly
- Can contain both abstract and concrete methods
- Can have constructors
- Can have fields and properties
- Supports single inheritance only
- Can have access modifiers for members

```
public abstract class Vehicle
{
    // Fields
    protected string brand;
    protected int year;

    // Constructor
    public Vehicle(string brand, int year)
    {
        this.brand = brand;
        this.year = year;
    }

    // Abstract method - must be implemented
    public abstract void Start();
```

0

```
    public abstract double CalculateFuelEfficiency();

    // Concrete method - inherited as-is
    public void DisplayInfo()
    {
        Console.WriteLine($"{brand} {year}");
    }

    // Virtual method - can be overridden
    public virtual void Stop()
    {
        Console.WriteLine("Vehicle stopped");
    }

    // Properties
    public string Brand => brand;
    public int Year => year;
}

public class Car : Vehicle
{
    private double engineSize;

    public Car(string brand, int year, double engineSize)
        : base(brand, year)
    {
        this.engineSize = engineSize;
    }

    public override void Start()
    {
        Console.WriteLine($"Car {brand} started with {engineSize}L
engine");
    }

    public override double CalculateFuelEfficiency()
    {
        return 25.0 - (engineSize * 2); // Simplified calculation
    }
```

0

```
    public override void Stop()
    {
        Console.WriteLine("Car stopped with brake pedal");
    }
}
```

Interfaces

Definition

An interface defines a contract that implementing classes must follow. It contains only declarations.

Key Characteristics

- Cannot be instantiated
- Contains only method signatures, properties, events, indexers
- No implementation (except default interface methods in C# 8+)
- No fields or constructors
- Supports multiple inheritance
- All members are implicitly public

```
public interface IVehicle
{
    // Properties
    string Brand { get; }
    int Year { get; }

    // Methods
    void Start();
    void Stop();
    double CalculateFuelEfficiency();
}

public interface IElectric
{
    int BatteryCapacity { get; }
    void Charge();
    double GetRemainingCharge();
}
```

0

```
public interface IGasoline
{
    double FuelTankCapacity { get; }
    void Refuel();
    double GetRemainingFuel();
}

// Class implementing multiple interfaces
public class HybridCar : IVehicle, IElectric, IGasoline
{
    public string Brand { get; private set; }
    public int Year { get; private set; }
    public int BatteryCapacity { get; private set; }
    public double FuelTankCapacity { get; private set; }

    private double currentCharge;
    private double currentFuel;

    public HybridCar(string brand, int year, int batteryCapacity, double
fuelCapacity)
    {
        Brand = brand;
        Year = year;
        BatteryCapacity = batteryCapacity;
        FuelTankCapacity = fuelCapacity;
        currentCharge = batteryCapacity;
        currentFuel = fuelCapacity;
    }

    public void Start()
    {
        Console.WriteLine($"Hybrid {Brand} started (using battery
first)");
    }

    public void Stop()
    {
        Console.WriteLine("Hybrid car stopped");
    }
```

0

```
    public double CalculateFuelEfficiency()
    {
        return 50.0; // High efficiency due to hybrid nature
    }

    public void Charge()
    {
        currentCharge = BatteryCapacity;
        Console.WriteLine("Battery fully charged");
    }

    public double GetRemainingCharge()
    {
        return currentCharge;
    }

    public void Refuel()
    {
        currentFuel = FuelTankCapacity;
        Console.WriteLine("Fuel tank refilled");
    }

    public double GetRemainingFuel()
    {
        return currentFuel;
    }
}
```

Comparison Table

Feature	Abstract Class	Interface
Instantiation	Cannot be instantiated	Cannot be instantiated
Implementation	Can have both abstract and concrete methods	Only method signatures (except default methods)
Fields	Can have fields	Cannot have fields
Constructors	Can have constructors	Cannot have constructors
Access Modifiers	Can use various access modifiers	Members are implicitly public

0

Feature	Abstract Class	Interface
Inheritance	Single inheritance	Multiple inheritance
When to Use	When classes share common implementation	When classes share common behavior contract

When to Use Which?

Use Abstract Class When:

- You want to share code among several closely related classes
- You expect classes that extend your abstract class to have many common methods or fields
- You want to declare non-public members
- You need to provide a common constructor

Use Interface When:

- You expect unrelated classes to implement your interface
- You want to specify the behavior of a particular data type, but not concerned about who implements it
- You want to support multiple inheritance of type
- You want to provide a contract for classes to follow

```
// Example: When to use both
public abstract class DatabaseConnection
{
    protected string connectionString;

    protected DatabaseConnection(string connectionString)
    {
        this.connectionString = connectionString;
    }

    public abstract void Connect();
    public abstract void Disconnect();

    // Common implementation
    public void LogOperation(string operation)
    {
        Console.WriteLine($"[{DateTime.Now}] {operation}");
```

0

```
    }
}

public interface IQueryable
{
    IEnumerable<T> Query<T>(string sql);
    void Execute(string sql);
}

public class SqlServerConnection : DatabaseConnection, IQueryable
{
    public SqlServerConnection(string connectionString)
        : base(connectionString) { }

    public override void Connect()
    {
        LogOperation("Connecting to SQL Server");
    }

    public override void Disconnect()
    {
        LogOperation("Disconnecting from SQL Server");
    }

    public IEnumerable<T> Query<T>(string sql)
    {
        LogOperation($"Executing query: {sql}");
        // Implementation here
        return new List<T>();
    }

    public void Execute(string sql)
    {
        LogOperation($"Executing command: {sql}");
        // Implementation here
    }
}
```

Inheritance and Encapsulation

0

Inheritance

Definition

Inheritance allows a class to inherit properties and methods from another class, promoting code reuse and establishing an "is-a" relationship.

Types of Inheritance in C#

```
// Base class (Parent)
public class Person
{
    protected string name;
    protected int age;

    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public virtual void Introduce()
    {
        Console.WriteLine($"Hi, I'm {name} and I'm {age} years old.");
    }

    public void Sleep()
    {
        Console.WriteLine($"{name} is sleeping.");
    }
}

// Derived class (Child)
public class Student : Person
{
    private string studentId;
    private List<string> courses;

    public Student(string name, int age, string studentId)
        : base(name, age) // Call parent constructor
```

```
    {
        this.studentId = studentId;
        this.courses = new List<string>();
    }

    // Override parent method
    public override void Introduce()
    {
        Console.WriteLine($"Hi, I'm {name}, a student with ID
{studentId}.");
    }

    // New method specific to Student
    public void EnrollInCourse(string course)
    {
        courses.Add(course);
        Console.WriteLine($"{name} enrolled in {course}");
    }

    public void Study()
    {
        Console.WriteLine($"{name} is studying.");
    }
}

// Further inheritance
public class GraduateStudent : Student
{
    private string researchTopic;

    public GraduateStudent(string name, int age, string studentId, string
researchTopic)
        : base(name, age, studentId)
    {
        this.researchTopic = researchTopic;
    }

    public override void Introduce()
    {
        Console.WriteLine($"Hi, I'm {name}, a graduate student
```

```
researching {researchTopic}.");
    }

    public void Conduct_Research()
    {
        Console.WriteLine($"{name} is conducting research on
{researchTopic}.");
    }
}
```

Method Hiding vs Overriding

```
public class BaseClass
{
    public virtual void VirtualMethod()
    {
        Console.WriteLine("Base virtual method");
    }

    public void RegularMethod()
    {
        Console.WriteLine("Base regular method");
    }
}

public class DerivedClass : BaseClass
{
    // Method overriding (runtime polymorphism)
    public override void VirtualMethod()
    {
        Console.WriteLine("Derived overridden method");
    }

    // Method hiding (compile-time)
    public new void RegularMethod()
    {
        Console.WriteLine("Derived hidden method");
    }
}
```

0

```
// Usage demonstration
BaseClass baseRef = new DerivedClass();
baseRef.VirtualMethod();  // "Derived overridden method" (polymorphism)
baseRef.RegularMethod();  // "Base regular method" (no polymorphism)

DerivedClass derivedRef = new DerivedClass();
derivedRef.VirtualMethod(); // "Derived overridden method"
derivedRef.RegularMethod(); // "Derived hidden method"
```

Encapsulation

Definition

Encapsulation is the bundling of data and methods that operate on that data within a single unit, while restricting access to some components.

Access Modifiers

```
public class BankAccount
{
    // Private fields (encapsulated data)
    private string accountNumber;
    private decimal balance;
    private string ownerName;

    // Public constructor
    public BankAccount(string accountNumber, string ownerName, decimal
initialBalance = 0)
    {
        this.accountNumber = accountNumber;
        this.ownerName = ownerName;
        this.balance = initialBalance >= 0 ? initialBalance : 0;
    }

    // Public properties (controlled access)
    public string AccountNumber
    {
        get { return accountNumber; }
```

0

```
        // No setter - read-only
    }

    public string OwnerName
    {
        get { return ownerName; }
        set
        {
            if (!string.IsNullOrWhiteSpace(value))
                ownerName = value;
        }
    }

    public decimal Balance
    {
        get { return balance; }
        // No public setter - controlled through methods
    }

    // Public methods (controlled operations)
    public bool Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
            LogTransaction($"Deposited {amount:C}");
            return true;
        }
        return false;
    }

    public bool Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= balance)
        {
            balance -= amount;
            LogTransaction($"Withdrew {amount:C}");
            return true;
        }
        return false;
```

0

```
    }

    // Private helper method (internal implementation)
    private void LogTransaction(string transaction)
    {
        Console.WriteLine($"[{DateTime.Now}] {AccountNumber}:
{transaction}. Balance: {balance:C}");
    }

    // Protected method (accessible to derived classes)
    protected virtual bool ValidateTransaction(decimal amount)
    {
        return amount > 0 && amount <= balance;
    }
}

// Inheritance with encapsulation
public class SavingsAccount : BankAccount
{
    private decimal interestRate;
    private DateTime lastInterestDate;

    public SavingsAccount(string accountNumber, string ownerName, decimal
interestRate, decimal initialBalance = 0)
        : base(accountNumber, ownerName, initialBalance)
    {
        this.interestRate = interestRate;
        this.lastInterestDate = DateTime.Now;
    }

    public decimal InterestRate
    {
        get { return interestRate; }
        set { interestRate = value > 0 ? value : interestRate; }
    }

    public void ApplyInterest()
    {
        if (DateTime.Now.Month != lastInterestDate.Month)
        {
0
```

```
            decimal interest = Balance * (interestRate / 100 / 12);
            Deposit(interest); // Using inherited method
            lastInterestDate = DateTime.Now;
        }
    }

    // Override inherited behavior
    protected override bool ValidateTransaction(decimal amount)
    {
        // Savings account might have different validation rules
        return base.ValidateTransaction(amount) && amount <= 1000; //
Daily limit
    }
}
```

Properties and Auto-Properties

```
public class Product
{
    // Full property with backing field
    private decimal price;
    public decimal Price
    {
        get { return price; }
        set
        {
            if (value >= 0)
                price = value;
            else
                throw new ArgumentException("Price cannot be negative");
        }
    }

    // Auto-property (compiler generates backing field)
    public string Name { get; set; }

    // Auto-property with private setter
    public DateTime CreatedDate { get; private set; }
```

0

```
    // Auto-property with default value
    public bool IsActive { get; set; } = true;

    // Read-only auto-property
    public int Id { get; }

    // Constructor
    public Product(int id, string name, decimal price)
    {
        Id = id; // Can only be set in constructor
        Name = name;
        Price = price;
        CreatedDate = DateTime.Now;
    }

    // Computed property
    public string DisplayName => $"{Name} (${Price:F2})";
}
```

Exception Handling

Definition

Exception handling is a programming construct that allows programs to respond to exceptional circumstances during execution.

Try-Catch-Finally Structure

```
public class ExceptionHandlingExamples
{
    public static void BasicExceptionHandling()
    {
        try
        {
            // Code that might throw an exception
            Console.Write("Enter a number: ");
            string input = Console.ReadLine();
            int number = int.Parse(input);
            int result = 100 / number;
```

```
            Console.WriteLine($"Result: {result}");
        }
        catch (FormatException ex)
        {
            // Handle specific exception type
            Console.WriteLine($"Invalid format: {ex.Message}");
        }
        catch (DivideByZeroException ex)
        {
            // Handle specific exception type
            Console.WriteLine($"Division by zero: {ex.Message}");
        }
        catch (Exception ex)
        {
            // Handle any other exception
            Console.WriteLine($"An error occurred: {ex.Message}");
        }
        finally
        {
            // Always executes (cleanup code)
            Console.WriteLine("Operation completed.");
        }
    }
}
```

Multiple Catch Blocks

```
public class FileProcessor
{
    public void ProcessFile(string filePath)
    {
        FileStream fileStream = null;
        StreamReader reader = null;

        try
        {
            fileStream = new FileStream(filePath, FileMode.Open);
            reader = new StreamReader(fileStream);

0
```

```
            string content = reader.ReadToEnd();
            int lineCount = content.Split('\n').Length;

            Console.WriteLine($"File processed. Line count:
{lineCount}");
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine($"File not found: {ex.FileName}");
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.WriteLine($"Access denied: {ex.Message}");
        }
        catch (IOException ex)
        {
            Console.WriteLine($"I/O error: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Unexpected error: {ex.Message}");
            // Log the full exception for debugging
            Console.WriteLine($"Stack trace: {ex.StackTrace}");
        }
        finally
        {
            // Cleanup resources
            reader?.Dispose();
            fileStream?.Dispose();
            Console.WriteLine("Resources cleaned up.");
        }
    }
}
```

Custom Exceptions

```
// Custom exception class
public class InsufficientFundsException : Exception
{
```

0

```
    public decimal RequiredAmount { get; }
    public decimal AvailableAmount { get; }

    public InsufficientFundsException() : base() { }

    public InsufficientFundsException(string message) : base(message) { }

    public InsufficientFundsException(string message, Exception
innerException)
        : base(message, innerException) { }

    public InsufficientFundsException(decimal required, decimal
available)
        : base($"Insufficient funds. Required: {required:C}, Available:
{available:C}")
    {
        RequiredAmount = required;
        AvailableAmount = available;
    }
}

// Usage of custom exception
public class BankAccountWithExceptions
{
    private decimal balance;

    public decimal Balance => balance;

    public void Withdraw(decimal amount)
    {
        if (amount <= 0)
            throw new ArgumentException("Amount must be positive",
nameof(amount));

        if (amount > balance)
            throw new InsufficientFundsException(amount, balance);

        balance -= amount;
    }
```

0

```
    public void ProcessWithdrawal(decimal amount)
    {
        try
        {
            Withdraw(amount);
            Console.WriteLine($"Withdrawal successful. New balance:
{balance:C}");
        }
        catch (InsufficientFundsException ex)
        {
            Console.WriteLine($"Withdrawal failed: {ex.Message}");
            Console.WriteLine($"You need {ex.RequiredAmount -
ex.AvailableAmount:C} more.");
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Invalid amount: {ex.Message}");
        }
    }
}
```

Exception Propagation and Rethrowing

```
public class ExceptionPropagation
{
    public void MethodA()
    {
        try
        {
            MethodB();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Caught in MethodA: {ex.Message}");
            // Log and rethrow
            throw; // Preserves original stack trace
        }
    }
```

0

```
    public void MethodB()
    {
        try
        {
            MethodC();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Caught in MethodB: {ex.Message}");
            // Wrap and throw new exception
            throw new InvalidOperationException("Error in MethodB", ex);
        }
    }

    public void MethodC()
    {
        throw new ArgumentException("Something went wrong in MethodC");
    }
}
```

Using Statement for Resource Management

```
public class ResourceManagement
{
    // Using statement automatically calls Dispose()
    public void ReadFileWithUsing(string filePath)
    {
        try
        {
            using (var fileStream = new FileStream(filePath,
FileMode.Open))
            using (var reader = new StreamReader(fileStream))
            {
                string content = reader.ReadToEnd();
                Console.WriteLine($"File content length:
{content.Length}");
                // fileStream and reader are automatically disposed
            }
        }
0
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"Error reading file: {ex.Message}");
        }
    }

    // Multiple using statements (C# 8+ syntax)
    public void ReadFileWithMultipleUsing(string filePath)
    {
        try
        {
            using var fileStream = new FileStream(filePath,
FileMode.Open);
            using var reader = new StreamReader(fileStream);

            string content = reader.ReadToEnd();
            Console.WriteLine($"File content length: {content.Length}");
            // Automatic disposal at end of method
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error reading file: {ex.Message}");
        }
    }
}
```

Best Practices

```
public class ExceptionBestPractices
{
    // DON'T: Catch and ignore exceptions
    public void BadExample1()
    {
        try
        {
            // Some operation
            int result = int.Parse("abc");
        }
        catch
```

0

```
        {
            // Silently ignoring exception - BAD!
        }
    }

    // DON'T: Catch Exception when you should catch specific types
    public void BadExample2()
    {
        try
        {
            // Some operation
        }
        catch (Exception ex)
        {
            // Too broad - might catch unexpected exceptions
            Console.WriteLine("Something went wrong");
        }
    }

    // DO: Catch specific exceptions and handle appropriately
    public bool TryParseNumber(string input, out int result)
    {
        result = 0;
        try
        {
            result = int.Parse(input);
            return true;
        }
        catch (FormatException)
        {
            return false;
        }
        catch (OverflowException)
        {
            return false;
        }
    }

    // DO: Use Try* methods when available
    public bool SafeParseNumber(string input, out int result)
```

```
    {
        return int.TryParse(input, out result); // No exception throwing
    }

    // DO: Provide meaningful error messages
    public void ValidateAge(int age)
    {
        if (age < 0)
            throw new ArgumentOutOfRangeException(nameof(age), age, "Age
cannot be negative");

        if (age > 150)
            throw new ArgumentOutOfRangeException(nameof(age), age, "Age
cannot exceed 150 years");
    }
}
```

Parallel Programming

Definition

Parallel programming involves executing multiple computations simultaneously to improve performance on multi-core processors.

Task Parallel Library (TPL)

Basic Task Usage

```
using System.Threading.Tasks;

public class TaskExamples
{
    public static async Task BasicTaskExample()
    {
        // Creating and starting a task
        Task task1 = Task.Run(() =>
        {
            Console.WriteLine($"Task 1 running on thread
{Thread.CurrentThread.ManagedThreadId}");
```

```
            Thread.Sleep(2000);
            Console.WriteLine("Task 1 completed");
        });

        // Task with return value
        Task<int> task2 = Task.Run(() =>
        {
            Console.WriteLine($"Task 2 running on thread
{Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(1000);
            return 42;
        });

        // Wait for tasks to complete
        await task1;
        int result = await task2;
        Console.WriteLine($"Task 2 result: {result}");

        // Alternative: Wait for all tasks
        await Task.WhenAll(task1, task2);
    }

    public static void TaskWithContinuation()
    {
        Task<string> downloadTask = Task.Run(() =>
        {
            Thread.Sleep(2000);
            return "Downloaded data";
        });

        // Continuation task
        Task processTask = downloadTask.ContinueWith(antecedent =>
        {
            string data = antecedent.Result;
            Console.WriteLine($"Processing: {data}");
        });

        processTask.Wait();


0
```

```
    }
}
```

Parallel Loops

```
public class ParallelLoopExamples
{
    public static void ParallelForExample()
    {
        // Sequential version
        Console.WriteLine("Sequential processing:");
        var stopwatch = Stopwatch.StartNew();
        for (int i = 0; i < 10; i++)
        {
            ProcessNumber(i);
        }
        stopwatch.Stop();
        Console.WriteLine($"Sequential time:
{stopwatch.ElapsedMilliseconds}ms");

        // Parallel version
        Console.WriteLine("\nParallel processing:");
        stopwatch.Restart();
        Parallel.For(0, 10, i =>
        {
            ProcessNumber(i);
        });
        stopwatch.Stop();
        Console.WriteLine($"Parallel time:
{stopwatch.ElapsedMilliseconds}ms");
    }

    public static void ParallelForEachExample()
    {
        var numbers = Enumerable.Range(1, 1000).ToList();
        var results = new ConcurrentBag<int>();

        Parallel.ForEach(numbers, number =>
        {
```

0

```
            int result = ExpensiveCalculation(number);
            results.Add(result);
        });

        Console.WriteLine($"Processed {results.Count} numbers");
    }

    private static void ProcessNumber(int number)
    {
        Thread.Sleep(100); // Simulate work
        Console.WriteLine($"Processed {number} on thread
{Thread.CurrentThread.ManagedThreadId}");
    }

    private static int ExpensiveCalculation(int number)
    {
        // Simulate expensive calculation
        Thread.Sleep(10);
        return number * number;
    }
}
```

PLINQ (Parallel LINQ)

```
public class PLinqExamples
{
    public static void BasicPLinqExample()
    {
        var numbers = Enumerable.Range(1, 1000000).ToArray();

        // Sequential LINQ
        var sequentialStopwatch = Stopwatch.StartNew();
        var sequentialResult = numbers
            .Where(n => n % 2 == 0)
            .Select(n => n * n)
            .Sum();
        sequentialStopwatch.Stop();

        // Parallel LINQ
```

0

```
        var parallelStopwatch = Stopwatch.StartNew();
        var parallelResult = numbers
            .AsParallel()
            .Where(n => n % 2 == 0)
            .Select(n => n * n)
            .Sum();
        parallelStopwatch.Stop();

        Console.WriteLine($"Sequential result: {sequentialResult}, Time:
{sequentialStopwatch.ElapsedMilliseconds}ms");
        Console.WriteLine($"Parallel result: {parallelResult}, Time:
{parallelStopwatch.ElapsedMilliseconds}ms");
    }

    public static void PLinqWithOptions()
    {
        var data = Enumerable.Range(1, 1000).ToArray();

        var result = data
            .AsParallel()
            .WithDegreeOfParallelism(4) // Limit to 4 threads
            .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
            .Where(n => ExpensiveFilter(n))
            .Select(n => ExpensiveTransform(n))
            .OrderBy(n => n) // This forces sequential execution
            .ToArray();

        Console.WriteLine($"Processed {result.Length} items");
    }

    private static bool ExpensiveFilter(int number)
    {
        Thread.Sleep(1); // Simulate work
        return number % 3 == 0;
    }

    private static int ExpensiveTransform(int number)
    {
        Thread.Sleep(1); // Simulate work
        return number * 2;
```

0

```
    }
}
```

Thread-Safe Collections

```
public class ThreadSafeCollectionsExample
{
    public static void ConcurrentCollectionsExample()
    {
        // ConcurrentBag - Thread-safe collection of objects
        var bag = new ConcurrentBag<int>();

        Parallel.For(0, 100, i =>
        {
            bag.Add(i);
        });

        Console.WriteLine($"Bag contains {bag.Count} items");

        // ConcurrentDictionary - Thread-safe dictionary
        var dictionary = new ConcurrentDictionary<string, int>();

        Parallel.For(0, 100, i =>
        {
            dictionary.TryAdd($"key{i}", i);
        });

        // Safe operations
        dictionary.AddOrUpdate("key1", 1, (key, oldValue) => oldValue +
1);
        int value = dictionary.GetOrAdd("newKey", 999);

        Console.WriteLine($"Dictionary contains {dictionary.Count}
items");

        // ConcurrentQueue - Thread-safe FIFO collection
        var queue = new ConcurrentQueue<string>();

        Task producer = Task.Run(() =>
```

```
        {
            for (int i = 0; i < 10; i++)
            {
                queue.Enqueue($"Item {i}");
                Thread.Sleep(100);
            }
        });

        Task consumer = Task.Run(() =>
        {
            while (!producer.IsCompleted || !queue.IsEmpty)
            {
                if (queue.TryDequeue(out string item))
                {
                    Console.WriteLine($"Consumed: {item}");
                }
                Thread.Sleep(50);
            }
        });

        Task.WaitAll(producer, consumer);
    }
}
```

Async/Await Pattern

```
public class AsyncAwaitExamples
{
    public static async Task AsyncMethodExample()
    {
        Console.WriteLine("Starting async operations...");

        // Start multiple async operations
        Task<string> download1 =
DownloadDataAsync("https://api1.example.com");
        Task<string> download2 =
DownloadDataAsync("https://api2.example.com");
        Task<string> download3 =
DownloadDataAsync("https://api3.example.com");
```

0

```
        // Wait for all to complete
        string[] results = await Task.WhenAll(download1, download2,
download3);

        foreach (string result in results)
        {
            Console.WriteLine($"Downloaded: {result}");
        }
    }

    private static async Task<string> DownloadDataAsync(string url)
    {
        using (var client = new HttpClient())
        {
            // Simulate network delay
            await Task.Delay(Random.Shared.Next(1000, 3000));
            return $"Data from {url}";
        }
    }

    public static async Task CancellationExample()
    {
        using var cts = new CancellationTokenSource();

        // Cancel after 5 seconds
        cts.CancelAfter(TimeSpan.FromSeconds(5));

        try
        {
            await LongRunningOperationAsync(cts.Token);
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Operation was cancelled");
        }
    }

    private static async Task LongRunningOperationAsync(CancellationToken
cancellationToken)
```

0

```
    {
        for (int i = 0; i < 100; i++)
        {
            cancellationToken.ThrowIfCancellationRequested();

            // Simulate work
            await Task.Delay(100, cancellationToken);
            Console.WriteLine($"Step {i + 1}/100");
        }
    }
}
```

Synchronization Primitives

```
public class SynchronizationExamples
{
    private static readonly object lockObject = new object();
    private static int sharedCounter = 0;

    public static void LockExample()
    {
        Task[] tasks = new Task[10];

        for (int i = 0; i < 10; i++)
        {
            tasks[i] = Task.Run(() =>
            {
                for (int j = 0; j < 1000; j++)
                {
                    lock (lockObject)
                    {
                        sharedCounter++;
                    }
                }
            });
        }

        Task.WaitAll(tasks);
        Console.WriteLine($"Final counter value: {sharedCounter}"); //
0
```

```
Should be 10000
    }

    private static readonly SemaphoreSlim semaphore = new
SemaphoreSlim(3); // Allow 3 concurrent operations

    public static async Task SemaphoreExample()
    {
        Task[] tasks = new Task[10];

        for (int i = 0; i < 10; i++)
        {
            int taskId = i;
            tasks[i] = AccessResourceAsync(taskId);
        }

        await Task.WhenAll(tasks);
    }

    private static async Task AccessResourceAsync(int id)
    {
        await semaphore.WaitAsync();
        try
        {
            Console.WriteLine($"Task {id} accessing resource at
{DateTime.Now:HH:mm:ss.fff}");
            await Task.Delay(2000); // Simulate work
            Console.WriteLine($"Task {id} finished at
{DateTime.Now:HH:mm:ss.fff}");
        }
        finally
        {
            semaphore.Release();
        }
    }
}
```

ADO.NET

0

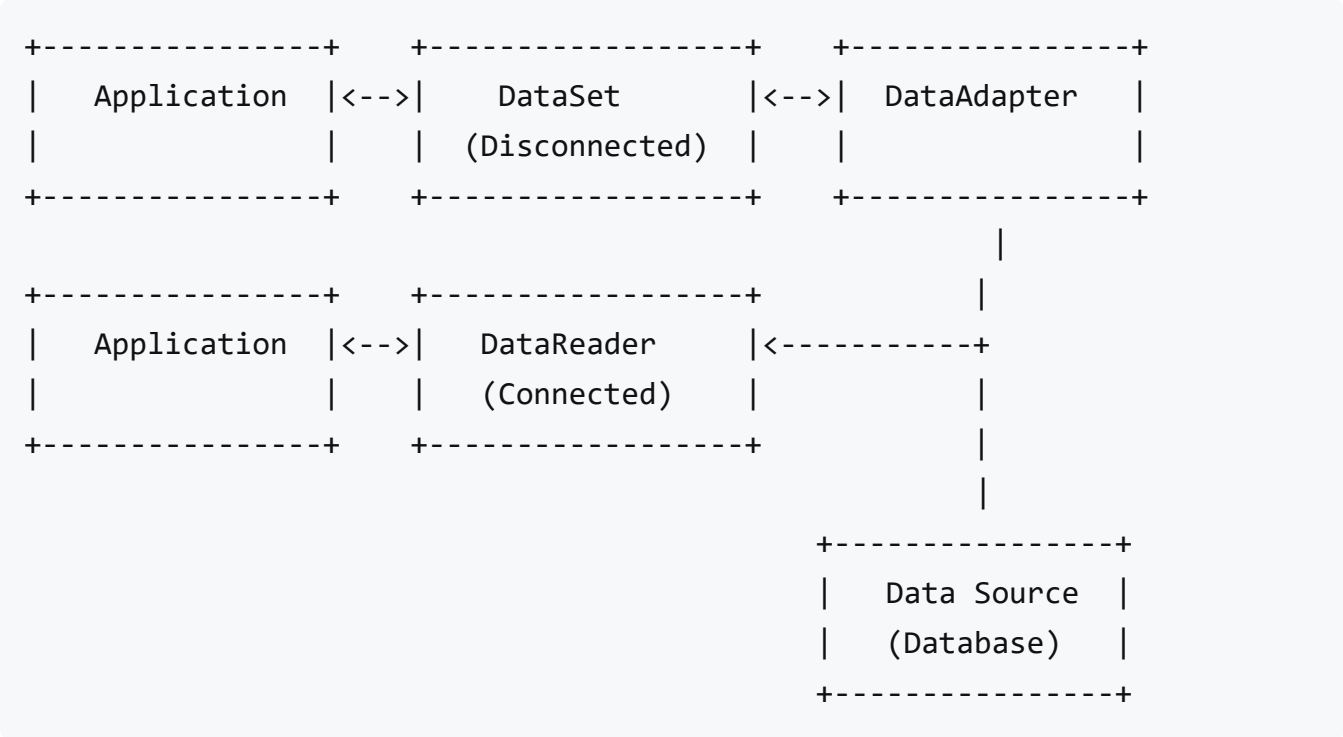
Definition

ADO.NET is a set of classes that expose data access services for .NET Framework programmers, providing access to relational databases, XML, and application data.

ADO.NET Architecture

Core Components

1. **Data Provider** - Connects to database
2. **DataSet** - In-memory representation of data
3. **DataAdapter** - Bridge between DataSet and data source
4. **DataReader** - Forward-only, read-only data stream



Connection Types:

- **DataSet**: Disconnected - loads data into memory, works offline
- **DataReader**: Connected - requires active connection, reads forward-only

DataReader vs DataSet

DataReader (Connected Architecture)

```
public class DataReaderExample
{
```

```
    private string connectionString =
"Server=localhost;Database=TestDB;Integrated Security=true;";

    public void ReadDataWithDataReader()
    {
        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            string sql = "SELECT EmployeeId, FirstName, LastName, Salary
FROM Employees";
            using (SqlCommand command = new SqlCommand(sql, connection))
            using (SqlDataReader reader = command.ExecuteReader())
            {
                Console.WriteLine("Employee Data:");
                Console.WriteLine("ID\tFirst Name\tLast Name\tSalary");
                Console.WriteLine("".PadRight(50, '-'));

                while (reader.Read())
                {
                    int id = reader.GetInt32("EmployeeId");
                    string firstName = reader.GetString("FirstName");
                    string lastName = reader.GetString("LastName");
                    decimal salary = reader.GetDecimal("Salary");

                    Console.WriteLine($"
{id}\t{firstName}\t\t{lastName}\t\t{salary:C}");
                }
            }
            // Connection automatically closed due to using statement
        }
    }

    public async Task ReadDataAsync()
    {
        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            await connection.OpenAsync();
```

```
            string sql = "SELECT * FROM Products WHERE CategoryId =
@categoryId";
            using (SqlCommand command = new SqlCommand(sql, connection))
            {
                command.Parameters.AddWithValue("@categoryId", 1);

                using (SqlDataReader reader = await
command.ExecuteReaderAsync())
                {
                    while (await reader.ReadAsync())
                    {
                        string productName =
reader["ProductName"].ToString();
                        decimal price =
Convert.ToDecimal(reader["Price"]);
                        Console.WriteLine($"{productName}: {price:C}");
                    }
                }
            }
        }
    }
}
```

DataSet (Disconnected Architecture)

```
public class DataSetExample
{
    private string connectionString =
"Server=localhost;Database=TestDB;Integrated Security=true;";

    public void WorkWithDataSet()
    {
        // Create DataSet and DataAdapter
        DataSet dataSet = new DataSet("EmployeeDataSet");

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
```

0

```
            // Create DataAdapter
            string sql = "SELECT EmployeeId, FirstName, LastName,
DepartmentId FROM Employees";
            SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);

            // Fill DataSet (connection opened and closed automatically)
            adapter.Fill(dataSet, "Employees");

            // Work with data offline
            DataTable employeeTable = dataSet.Tables["Employees"];

            // Display data
            foreach (DataRow row in employeeTable.Rows)
            {
                Console.WriteLine($"{row["EmployeeId"]}:
{row["FirstName"]} {row["LastName"]}");
            }

            // Modify data
            DataRow newRow = employeeTable.NewRow();
            newRow["FirstName"] = "John";
            newRow["LastName"] = "Doe";
            newRow["DepartmentId"] = 1;
            employeeTable.Rows.Add(newRow);

            // Update database
            SqlCommandBuilder commandBuilder = new
SqlCommandBuilder(adapter);
            adapter.Update(dataSet, "Employees");
        }
    }

    public void DataSetWithRelations()
    {
        DataSet dataSet = new DataSet();

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            // Fill Departments table
0
```

```
            SqlDataAdapter deptAdapter = new SqlDataAdapter("SELECT *
FROM Departments", connection);
            deptAdapter.Fill(dataSet, "Departments");

            // Fill Employees table
            SqlDataAdapter empAdapter = new SqlDataAdapter("SELECT * FROM
Employees", connection);
            empAdapter.Fill(dataSet, "Employees");

            // Create relationship
            DataRelation relation = new DataRelation("DeptEmployees",
                dataSet.Tables["Departments"].Columns["DepartmentId"],
                dataSet.Tables["Employees"].Columns["DepartmentId"]);
            dataSet.Relations.Add(relation);

            // Navigate relationship
            foreach (DataRow deptRow in
dataSet.Tables["Departments"].Rows)
            {
                Console.WriteLine($"Department:
{deptRow["DepartmentName"]}");

                DataRow[] employees = deptRow.GetChildRows(relation);
                foreach (DataRow empRow in employees)
                {
                    Console.WriteLine($"  Employee: {empRow["FirstName"]}
{empRow["LastName"]}");
                }
            }
        }
    }
}
```

Steps to Connect to SQL Database

```
public class DatabaseConnection
{
    // Step 1: Define connection string
    private readonly string connectionString =
0
```

```
        "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";

    // Alternative with integrated security
    private readonly string integratedConnectionString =
        "Server=myServerAddress;Database=myDataBase;Integrated
Security=true;";

    public void ConnectToDatabase()
    {
        // Step 2: Create SqlConnection object
        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            try
            {
                // Step 3: Open the connection
                connection.Open();
                Console.WriteLine("Connection opened successfully!");
                Console.WriteLine($"Database: {connection.Database}");
                Console.WriteLine($"Server Version:
{connection.ServerVersion}");

                // Step 4: Create and execute command
                string sql = "SELECT COUNT(*) FROM Employees";
                using (SqlCommand command = new SqlCommand(sql,
connection))
                {
                    // Step 5: Execute command and get result
                    int employeeCount = (int)command.ExecuteScalar();
                    Console.WriteLine($"Total employees:
{employeeCount}");
                }

                // Step 6: Connection automatically closed by using
statement
            }
            catch (SqlException ex)
            {
                Console.WriteLine($"SQL Error: {ex.Message}");
0
```

```
            }
            catch (Exception ex)
            {
                Console.WriteLine($"General Error: {ex.Message}");
            }
        }
    }

    // CRUD Operations Example
    public class EmployeeCRUD
    {
        private readonly string connectionString;

        public EmployeeCRUD(string connectionString)
        {
            this.connectionString = connectionString;
        }

        // CREATE
        public int CreateEmployee(string firstName, string lastName,
decimal salary, int departmentId)
        {
            string sql = @"INSERT INTO Employees (FirstName, LastName,
Salary, DepartmentId)
                           VALUES (@firstName, @lastName, @salary,
@departmentId);
                           SELECT SCOPE_IDENTITY();";

            using (SqlConnection connection = new
SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand command = new SqlCommand(sql,
connection))
                {
                    command.Parameters.AddWithValue("@firstName",
firstName);
                    command.Parameters.AddWithValue("@lastName",
lastName);
                    command.Parameters.AddWithValue("@salary", salary);
```

```
                    command.Parameters.AddWithValue("@departmentId",
departmentId);

                    return Convert.ToInt32(command.ExecuteScalar());
                }
            }
        }

        // READ
        public Employee GetEmployee(int employeeId)
        {
            string sql = "SELECT * FROM Employees WHERE EmployeeId =
@employeeId";

            using (SqlConnection connection = new
SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand command = new SqlCommand(sql,
connection))
                {
                    command.Parameters.AddWithValue("@employeeId",
employeeId);

                    using (SqlDataReader reader =
command.ExecuteReader())
                    {
                        if (reader.Read())
                        {
                            return new Employee
                            {
                                EmployeeId =
reader.GetInt32("EmployeeId"),
                                FirstName =
reader.GetString("FirstName"),
                                LastName = reader.GetString("LastName"),
                                Salary = reader.GetDecimal("Salary"),
                                DepartmentId =
reader.GetInt32("DepartmentId")
                            };
```

0

```
                        }
                    }
                }
            }
            return null;
        }

        // UPDATE
        public bool UpdateEmployee(Employee employee)
        {
            string sql = @"UPDATE Employees
                           SET FirstName = @firstName, LastName =
@lastName,
                               Salary = @salary, DepartmentId =
@departmentId
                           WHERE EmployeeId = @employeeId";

            using (SqlConnection connection = new
SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand command = new SqlCommand(sql,
connection))
                {
                    command.Parameters.AddWithValue("@firstName",
employee.FirstName);
                    command.Parameters.AddWithValue("@lastName",
employee.LastName);
                    command.Parameters.AddWithValue("@salary",
employee.Salary);
                    command.Parameters.AddWithValue("@departmentId",
employee.DepartmentId);
                    command.Parameters.AddWithValue("@employeeId",
employee.EmployeeId);

                    return command.ExecuteNonQuery() > 0;
                }
            }
        }
```

0

```
        // DELETE
        public bool DeleteEmployee(int employeeId)
        {
            string sql = "DELETE FROM Employees WHERE EmployeeId =
@employeeId";

            using (SqlConnection connection = new
SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand command = new SqlCommand(sql,
connection))
                {
                    command.Parameters.AddWithValue("@employeeId",
employeeId);
                    return command.ExecuteNonQuery() > 0;
                }
            }
        }
    }

    public class Employee
    {
        public int EmployeeId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public decimal Salary { get; set; }
        public int DepartmentId { get; set; }
    }
}
```

LINQ to SQL

```
// Entity classes
[Table(Name = "Employees")]
public class Employee
{
    [Column(IsPrimaryKey = true, IsDbGenerated = true)]
    public int EmployeeId { get; set; }
```

0

```
    [Column]
    public string FirstName { get; set; }

    [Column]
    public string LastName { get; set; }

    [Column]
    public decimal Salary { get; set; }

    [Column]
    public int DepartmentId { get; set; }
}

// DataContext
public class CompanyDataContext : DataContext
{
    public CompanyDataContext(string connectionString) :
base(connectionString) { }

    public Table<Employee> Employees => GetTable<Employee>();
}

// Usage
public class LinqToSqlExample
{
    private string connectionString =
"Server=localhost;Database=Company;Integrated Security=true;";

    public void QueryWithLinqToSql()
    {
        using (var context = new CompanyDataContext(connectionString))
        {
            // Query syntax
            var highEarners = from emp in context.Employees
                              where emp.Salary > 50000
                              orderby emp.Salary descending
                              select emp;

            Console.WriteLine("High earners:");
```

0

```
            foreach (var employee in highEarners)
            {
                Console.WriteLine($"{employee.FirstName}
{employee.LastName}: {employee.Salary:C}");
            }

            // Method syntax
            var topPerformers = context.Employees
                .Where(e => e.Salary > 75000)
                .OrderByDescending(e => e.Salary)
                .Take(5)
                .ToList();

            // Insert new employee
            var newEmployee = new Employee
            {
                FirstName = "Jane",
                LastName = "Smith",
                Salary = 60000,
                DepartmentId = 1
            };

            context.Employees.InsertOnSubmit(newEmployee);
            context.SubmitChanges();
        }
    }
}
```

WPF

Definition

Windows Presentation Foundation (WPF) is Microsoft's latest approach to a GUI framework, used with the .NET Framework and .NET Core/5+.

XAML Basics

Basic Window Structure

0

```
<!-- MainWindow.xaml -->
<Window x:Class="WpfApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="My WPF Application"
        Height="450"
        Width="800"
        WindowStartupLocation="CenterScreen">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <!-- Header -->
        <TextBlock Grid.Row="0"
                   Text="Welcome to WPF"
                   FontSize="24"
                   FontWeight="Bold"
                   HorizontalAlignment="Center"
                   Margin="10"/>

        <!-- Content Area -->
        <StackPanel Grid.Row="1"
                    Orientation="Vertical"
                    Margin="20">

            <Label Content="Enter your name:"
                   FontWeight="Bold"/>

            <TextBox x:Name="NameTextBox"
                     Width="200"
                     HorizontalAlignment="Left"
                     Margin="0,5,0,10"/>

            <Button x:Name="GreetButton"
                    Content="Greet Me"
```

0

```
                    Width="100"
                    HorizontalAlignment="Left"
                    Click="GreetButton_Click"/>

            <TextBlock x:Name="ResultTextBlock"
                       FontSize="16"
                       Margin="0,10,0,0"
                       Foreground="Blue"/>

        </StackPanel>

        <!-- Status Bar -->
        <StatusBar Grid.Row="2">
            <StatusBarItem Content="Ready"/>
        </StatusBar>

    </Grid>
</Window>
```

Code-Behind

```
// MainWindow.xaml.cs
using System.Windows;

namespace WpfApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void GreetButton_Click(object sender, RoutedEventArgs e)
        {
            string name = NameTextBox.Text;
            if (string.IsNullOrWhiteSpace(name))
            {
                MessageBox.Show("Please enter your name!", "Warning",
```

```
                    MessageBoxButton.OK, MessageBoxImage.Warning);
                return;
            }

            ResultTextBlock.Text = $"Hello, {name}! Welcome to WPF.";
        }
    }
}
```

WPF Data Binding

Simple Data Binding

```
// Person model
public class Person : INotifyPropertyChanged
{
    private string firstName;
    private string lastName;
    private int age;

    public string FirstName
    {
        get { return firstName; }
        set
        {
            firstName = value;
            OnPropertyChanged();
            OnPropertyChanged(nameof(FullName)); // Update computed
property
        }
    }

    public string LastName
    {
        get { return lastName; }
        set
        {
            lastName = value;
            OnPropertyChanged();
```

0

```
            OnPropertyChanged(nameof(FullName));
        }
    }

    public int Age
    {
        get { return age; }
        set
        {
            age = value;
            OnPropertyChanged();
        }
    }

    public string FullName => $"{FirstName} {LastName}";

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

```
<!-- Data Binding XAML -->
<Window x:Class="WpfApp.DataBindingWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel Margin="20">

        <!-- Two-way binding -->
        <Label Content="First Name:"/>
        <TextBox Text="{Binding FirstName,
UpdateSourceTrigger=PropertyChanged}"
                 Margin="0,0,0,10"/>
```

0

```
        <Label Content="Last Name:"/>
        <TextBox Text="{Binding LastName,
UpdateSourceTrigger=PropertyChanged}"
                 Margin="0,0,0,10"/>

        <Label Content="Age:"/>
        <TextBox Text="{Binding Age,
UpdateSourceTrigger=PropertyChanged}"
                 Margin="0,0,0,10"/>

        <!-- One-way binding (computed property) -->
        <Label Content="Full Name:"/>
        <TextBlock Text="{Binding FullName}"
                   FontWeight="Bold"
                   FontSize="16"
                   Margin="0,0,0,10"/>

        <!-- Binding with conversion -->
        <TextBlock Margin="0,10,0,0">
            <TextBlock.Text>
                <MultiBinding StringFormat="Person: {0}, Age: {1}">
                    <Binding Path="FullName"/>
                    <Binding Path="Age"/>
                </MultiBinding>
            </TextBlock.Text>
        </TextBlock>

    </StackPanel>
</Window>
```

```
// Code-behind for data binding
public partial class DataBindingWindow : Window
{
    public DataBindingWindow()
    {
        InitializeComponent();

        // Set DataContext
        DataContext = new Person
```

0

```
        {
            FirstName = "John",
            LastName = "Doe",
            Age = 30
        };
    }
}
```

Collection Binding with ListBox

```
<Window x:Class="WpfApp.EmployeeListWindow">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="300"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <!-- Employee List -->
        <ListBox Grid.Column="0"
                 ItemsSource="{Binding Employees}"
                 SelectedItem="{Binding SelectedEmployee}"
                 DisplayMemberPath="FullName"
                 Margin="10"/>

        <!-- Employee Details -->
        <StackPanel Grid.Column="1"
                    DataContext="{Binding SelectedEmployee}"
                    Margin="10">

            <Label Content="Employee Details"
                   FontWeight="Bold"
                   FontSize="16"/>

            <Label Content="First Name:"/>
            <TextBox Text="{Binding FirstName,
UpdateSourceTrigger=PropertyChanged}"/>

            <Label Content="Last Name:"/>
            <TextBox Text="{Binding LastName,
```

0

```
UpdateSourceTrigger=PropertyChanged}"/>

            <Label Content="Department:"/>
            <ComboBox ItemsSource="{Binding DataContext.Departments,
                                    RelativeSource={RelativeSource
AncestorType=Window}}"
                      SelectedItem="{Binding Department}"
                      DisplayMemberPath="Name"/>

            <Label Content="Salary:"/>
            <TextBox Text="{Binding Salary, StringFormat=C,
UpdateSourceTrigger=PropertyChanged}"/>

        </StackPanel>
    </Grid>
</Window>
```

Commands and MVVM Pattern

RelayCommand Implementation

```
public class RelayCommand : ICommand
{
    private readonly Action<object> execute;
    private readonly Func<object, bool> canExecute;

    public RelayCommand(Action<object> execute, Func<object, bool>
canExecute = null)
    {
        this.execute = execute ?? throw new
ArgumentNullException(nameof(execute));
        this.canExecute = canExecute;
    }

    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
```

0

```
    public bool CanExecute(object parameter)
    {
        return canExecute == null || canExecute(parameter);
    }

    public void Execute(object parameter)
    {
        execute(parameter);
    }
}
```

ViewModel Example

```
public class MainViewModel : INotifyPropertyChanged
{
    private ObservableCollection<Employee> employees;
    private Employee selectedEmployee;
    private string searchText;

    public MainViewModel()
    {
        Employees = new ObservableCollection<Employee>();
        LoadEmployees();

        // Initialize commands
        AddEmployeeCommand = new RelayCommand(AddEmployee);
        DeleteEmployeeCommand = new RelayCommand(DeleteEmployee,
CanDeleteEmployee);
        SearchCommand = new RelayCommand(Search);
    }

    public ObservableCollection<Employee> Employees
    {
        get { return employees; }
        set
        {
            employees = value;
            OnPropertyChanged();
```

0

```
        }
    }

    public Employee SelectedEmployee
    {
        get { return selectedEmployee; }
        set
        {
            selectedEmployee = value;
            OnPropertyChanged();
            CommandManager.InvalidateRequerySuggested(); // Update
command states
        }
    }

    public string SearchText
    {
        get { return searchText; }
        set
        {
            searchText = value;
            OnPropertyChanged();
        }
    }

    // Commands
    public ICommand AddEmployeeCommand { get; }
    public ICommand DeleteEmployeeCommand { get; }
    public ICommand SearchCommand { get; }

    private void AddEmployee(object parameter)
    {
        var newEmployee = new Employee
        {
            FirstName = "New",
            LastName = "Employee",
            Salary = 50000
        };
        Employees.Add(newEmployee);
        SelectedEmployee = newEmployee;
```

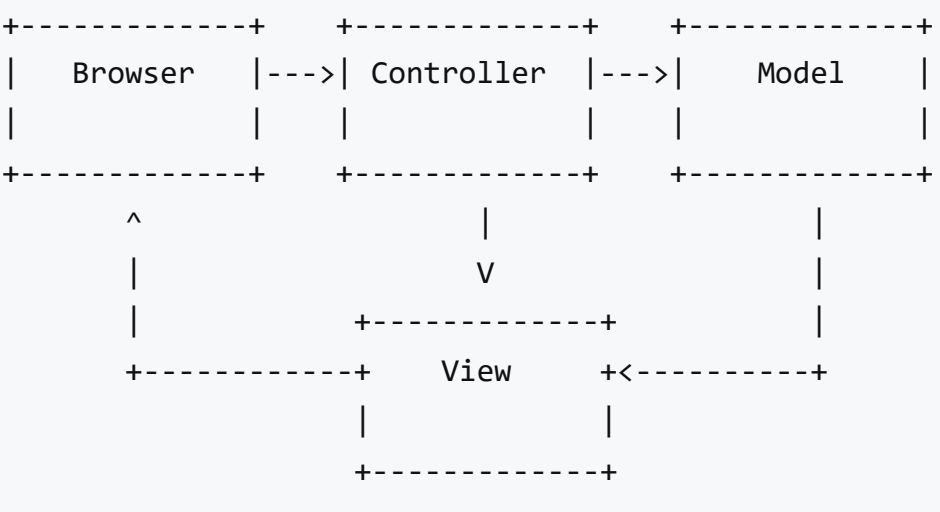
```
    }

    private void DeleteEmployee(object parameter)
    {
        if (SelectedEmployee != null)
        {
            Employees.Remove(SelectedEmployee);
            SelectedEmployee = null;
        }
    }

    private bool CanDeleteEmployee(object parameter)
    {
        return SelectedEmployee != null;
    }

    private void Search(object parameter)
    {
        // Implement search logic
        if (string.IsNullOrWhiteSpace(SearchText))
        {
            LoadEmployees(); // Show all
        }
        else
        {
            var filtered = employees.Where(e =>
                e.FirstName.Contains(SearchText,
StringComparison.OrdinalIgnoreCase) ||
                e.LastName.Contains(SearchText,
StringComparison.OrdinalIgnoreCase)).ToList();

            Employees.Clear();
            foreach (var emp in filtered)
            {
                Employees.Add(emp);
            }
        }
    }

    private void LoadEmployees()
0
```

```
    {
        // Load from database or service
        Employees.Clear();
        // Add sample data
        Employees.Add(new Employee { FirstName = "John", LastName =
"Doe", Salary = 60000 });
        Employees.Add(new Employee { FirstName = "Jane", LastName =
"Smith", Salary = 65000 });
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

ASP.NET & ASP.NET Core

ASP.NET Core MVC Architecture

Model-View-Controller Pattern



MVC Flow:

0

1. **Browser** sends request to **Controller**
2. **Controller** processes request, interacts with **Model**
3. **Model** returns data to **Controller**
4. **Controller** passes data to **View** for rendering
5. **View** sends rendered HTML back to **Browser**

Model Example

```
public class Product
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Product name is required")]
    [StringLength(100, ErrorMessage = "Product name cannot exceed 100
characters")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Description is required")]
    public string Description { get; set; }

    [Required(ErrorMessage = "Price is required")]
    [Range(0.01, double.MaxValue, ErrorMessage = "Price must be greater
than 0")]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [Required(ErrorMessage = "Category is required")]
    public int CategoryId { get; set; }

    public Category Category { get; set; }

    [DataType(DataType.Date)]
    public DateTime CreatedDate { get; set; } = DateTime.Now;

    public bool IsActive { get; set; } = true;
}

public class Category
{
    public int Id { get; set; }
```

0

```
    public string Name { get; set; }
    public List<Product> Products { get; set; } = new List<Product>();
}
```

Controller Example

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    private readonly IProductService productService;
    private readonly ILogger<ProductsController> logger;

    public ProductsController(IProductService productService,
ILogger<ProductsController> logger)
    {
        this.productService = productService;
        this.logger = logger;
    }

    // GET: api/products
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
    {
        try
        {
            var products = await productService.GetAllProductsAsync();
            return Ok(products);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error retrieving products");
            return StatusCode(500, "Internal server error");
        }
    }

    // GET: api/products/5
    [HttpGet("{id}")]
    public async Task<ActionResult<Product>> GetProduct(int id)
0
```

```
    {
        var product = await productService.GetProductByIdAsync(id);

        if (product == null)
        {
            return NotFound($"Product with ID {id} not found");
        }

        return Ok(product);
    }

    // POST: api/products
    [HttpPost]
    public async Task<ActionResult<Product>> CreateProduct(Product
product)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        try
        {
            var createdProduct = await
productService.CreateProductAsync(product);
            return CreatedAtAction(nameof(GetProduct),
                new { id = createdProduct.Id }, createdProduct);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error creating product");
            return StatusCode(500, "Internal server error");
        }
    }

    // PUT: api/products/5
    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateProduct(int id, Product
product)
    {
0
```

```
        if (id != product.Id)
        {
            return BadRequest("Product ID mismatch");
        }

        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        try
        {
            await productService.UpdateProductAsync(product);
            return NoContent();
        }
        catch (NotFoundException)
        {
            return NotFound($"Product with ID {id} not found");
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error updating product");
            return StatusCode(500, "Internal server error");
        }
    }

    // DELETE: api/products/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteProduct(int id)
    {
        try
        {
            await productService.DeleteProductAsync(id);
            return NoContent();
        }
        catch (NotFoundException)
        {
            return NotFound($"Product with ID {id} not found");
        }
        catch (Exception ex)
```

0

```
        {
            logger.LogError(ex, "Error deleting product");
            return StatusCode(500, "Internal server error");
        }
    }
}
```

Middleware in ASP.NET Core

Definition

Middleware is software that's assembled into an app pipeline to handle requests and responses.

Custom Middleware

```
// Custom middleware class
public class RequestLoggingMiddleware
{
    private readonly RequestDelegate next;
    private readonly ILogger<RequestLoggingMiddleware> logger;

    public RequestLoggingMiddleware(RequestDelegate next,
ILogger<RequestLoggingMiddleware> logger)
    {
        this.next = next;
        this.logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var startTime = DateTime.UtcNow;
        var requestId = Guid.NewGuid().ToString();

        // Log request
        logger.LogInformation($"[{requestId}] Request started:
{context.Request.Method} {context.Request.Path}");

        try
0
```

```
        {
            // Call the next middleware in the pipeline
            await next(context);
        }
        finally
        {
            var duration = DateTime.UtcNow - startTime;
            logger.LogInformation($"[{requestId}] Request completed:
{context.Response.StatusCode} in {duration.TotalMilliseconds:F2}ms");
        }
    }
}

// Extension method for easier registration
public static class RequestLoggingMiddlewareExtensions
{
    public static IApplicationBuilder UseRequestLogging(this
IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestLoggingMiddleware>();
    }
}
```

Startup/Program Configuration

```
// Program.cs (ASP.NET Core 6+)
var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConn

// Add custom services
builder.Services.AddScoped<IProductService, ProductService>();

// Add authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
```

0

```
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
        };
    });

var app = builder.Build();

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

// Custom middleware
app.UseRequestLogging();

app.UseAuthentication();
app.UseAuthorization();
```

0

```
app.MapControllers();

app.Run();
```

CRUD Operations using Razor Pages

Page Model

```
// Pages/Products/Index.cshtml.cs
public class IndexModel : PageModel
{
    private readonly IProductService productService;

    public IndexModel(IProductService productService)
    {
        this.productService = productService;
    }

    public IList<Product> Products { get; set; }

    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }

    public async Task OnGetAsync()
    {
        Products = await productService.GetProductsAsync(SearchString);
    }
}
```

```
<!-- Pages/Products/Index.cshtml -->
@page
@model IndexModel
@{
    ViewData["Title"] = "Products";
}

<h1>Products</h1>
```

0

```
<div class="row">
    <div class="col-md-6">
        <a asp-page="Create" class="btn btn-primary">Create New
Product</a>
    </div>
    <div class="col-md-6">
        <form method="get">
            <div class="input-group">
                <input asp-for="SearchString" class="form-control"
placeholder="Search products..."/>
                <div class="input-group-append">
                    <button class="btn btn-outline-secondary"
type="submit">Search</button>
                </div>
            </div>
        </form>
    </div>
</div>

<table class="table table-striped mt-3">
    <thead>
        <tr>
            <th>Name</th>
            <th>Description</th>
            <th>Price</th>
            <th>Category</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var product in Model.Products)
        {
            <tr>
                <td>@product.Name</td>
                <td>@product.Description</td>
                <td>@product.Price.ToString("C")</td>
                <td>@product.Category?.Name</td>
                <td>
                    <a asp-page="Details" asp-route-id="@product.Id"
class="btn btn-sm btn-info">Details</a>
```

0

```
                    <a asp-page="Edit" asp-route-id="@product.Id"
class="btn btn-sm btn-warning">Edit</a>
                    <a asp-page="Delete" asp-route-id="@product.Id"
class="btn btn-sm btn-danger">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Create Page

```
// Pages/Products/Create.cshtml.cs
public class CreateModel : PageModel
{
    private readonly IProductService productService;

    public CreateModel(IProductService productService)
    {
        this.productService = productService;
    }

    [BindProperty]
    public Product Product { get; set; }

    public SelectList Categories { get; set; }

    public async Task<IActionResult> OnGetAsync()
    {
        await LoadCategoriesAsync();
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            await LoadCategoriesAsync();
            return Page();
```

0

```
        }

        await productService.CreateProductAsync(Product);

        TempData["SuccessMessage"] = "Product created successfully!";
        return RedirectToPage("./Index");
    }

    private async Task LoadCategoriesAsync()
    {
        var categories = await productService.GetCategoriesAsync();
        Categories = new SelectList(categories, "Id", "Name");
    }
}
```

Authentication and Authorization

JWT Authentication

```
// Services/AuthService.cs
public class AuthService : IAuthService
{
    private readonly IConfiguration configuration;
    private readonly IUserService userService;

    public AuthService(IConfiguration configuration, IUserService
userService)
    {
        this.configuration = configuration;
        this.userService = userService;
    }

    public async Task<AuthResult> LoginAsync(LoginModel model)
    {
        var user = await userService.ValidateUserAsync(model.Email,
model.Password);
        if (user == null)
        {
            return new AuthResult { Success = false, Message = "Invalid
```

```
credentials" };
        }

        var token = GenerateJwtToken(user);
        return new AuthResult
        {
            Success = true,
            Token = token,
            User = user
        };
    }

    private string GenerateJwtToken(User user)
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes(configuration["Jwt:Key"]);

        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
            new Claim(ClaimTypes.Name, user.Email),
            new Claim(ClaimTypes.Email, user.Email)
        };

        // Add role claims
        foreach (var role in user.Roles)
        {
            claims.Add(new Claim(ClaimTypes.Role, role));
        }

        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(claims),
            Expires = DateTime.UtcNow.AddHours(24),
            Issuer = configuration["Jwt:Issuer"],
            Audience = configuration["Jwt:Audience"],
            SigningCredentials = new SigningCredentials(
                new SymmetricSecurityKey(key),
                SecurityAlgorithms.HmacSha256Signature)
        };
```

0

```
        var token = tokenHandler.CreateToken(tokenDescriptor);
        return tokenHandler.WriteToken(token);
    }
}
```

Authorization Policies

```
// Program.cs - Authorization setup
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy =>
        policy.RequireRole("Admin"));

    options.AddPolicy("ManageProducts", policy =>
        policy.RequireClaim("Permission", "ManageProducts"));

    options.AddPolicy("MinimumAge", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(18)));
});

// Custom authorization requirement
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}

public class MinimumAgeHandler :
AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        MinimumAgeRequirement requirement)
    {
```

0

```
        var birthDate =
context.User.FindFirst(ClaimTypes.DateOfBirth)?.Value;

        if (DateTime.TryParse(birthDate, out DateTime dateOfBirth))
        {
            var age = DateTime.Today.Year - dateOfBirth.Year;
            if (dateOfBirth.Date > DateTime.Today.AddYears(-age))
                age--;

            if (age >= requirement.MinimumAge)
            {
                context.Succeed(requirement);
            }
        }

        return Task.CompletedTask;
    }
}
```

Deployment Methods

Docker Deployment

```
# Dockerfile
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["MyApp/MyApp.csproj", "MyApp/"]
RUN dotnet restore "MyApp/MyApp.csproj"
COPY . .
WORKDIR "/src/MyApp"
RUN dotnet build "MyApp.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "MyApp.csproj" -c Release -o /app/publish
```

0

```
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "MyApp.dll"]
```

```
# docker-compose.yml
version: '3.8'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:80"
    environment:
      -
ConnectionStrings__DefaultConnection=Server=db;Database=MyAppDb;User=sa;Pas
      - ASPNETCORE_ENVIRONMENT=Production
    depends_on:
      - db

  db:
    image: mcr.microsoft.com/mssql/server:2019-latest
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=YourPassword123
    ports:
      - "1433:1433"
    volumes:
      - sqldata:/var/opt/mssql

volumes:
  sqldata:
```

Blazor

Definition

0

Blazor is a free and open-source web framework that enables developers to create web apps using C# and HTML, running either on the server or in the browser via WebAssembly.

Server-side vs Client-side Blazor

Feature	Blazor Server	Blazor WebAssembly
Execution	Runs on server	Runs in browser
Connection	Requires SignalR connection	Works offline
Performance	Fast startup, server processing	Slow startup, client processing
Scalability	Limited by server resources	Scales with client devices
Security	Code stays on server	Code downloaded to client
Network	Requires constant connection	Works with intermittent connection

Blazor Components

Basic Component Structure

```
@* Components/Counter.razor *@
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

0

Component with Parameters

```
@* Components/ProductCard.razor *@
<div class="card" style="width: 18rem;">
    <div class="card-body">
        <h5 class="card-title">@Product.Name</h5>
        <p class="card-text">@Product.Description</p>
        <p class="card-text">
            <strong>Price: @Product.Price.ToString("C")</strong>
        </p>
        <button class="btn btn-primary" @onclick="OnPurchaseClick">
            Buy Now
        </button>
    </div>
</div>

@code {
    [Parameter] public Product Product { get; set; }
    [Parameter] public EventCallback<Product> OnPurchase { get; set; }

    private async Task OnPurchaseClick()
    {
        await OnPurchase.InvokeAsync(Product);
    }
}
```

Component Creating Process

```
// 1. Define the component class
@inherits ComponentBase
@implements IDisposable

// 2. Add parameters and properties
@code {
    [Parameter] public string Title { get; set; }
    [Parameter] public RenderFragment ChildContent { get; set; }
    [Parameter] public EventCallback<string> OnValueChanged { get; set; }

    [Inject] public IJSRuntime JSRuntime { get; set; }
```

0

```
    [Inject] public IProductService ProductService { get; set; }

    private string inputValue = "";
    private List<Product> products = new();
    private Timer timer;

    // 3. Lifecycle methods
    protected override async Task OnInitializedAsync()
    {
        products = await ProductService.GetProductsAsync();
        timer = new Timer(UpdateTime, null, TimeSpan.Zero,
TimeSpan.FromSeconds(1));
    }

    protected override async Task OnParametersSetAsync()
    {
        // Called when parameters change
        if (!string.IsNullOrEmpty(Title))
        {
            await JSRuntime.InvokeVoidAsync("console.log", $"Title
changed to: {Title}");
        }
    }

    protected override bool ShouldRender()
    {
        // Control when component re-renders
        return !string.IsNullOrEmpty(Title);
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            await JSRuntime.InvokeVoidAsync("initializeComponent");
        }
    }

    // 4. Event handlers
    private async Task OnInputChange(ChangeEventArgs e)
0
```

```
    {
        inputValue = e.Value?.ToString() ?? "";
        await OnValueChanged.InvokeAsync(inputValue);
    }

    private void UpdateTime(object state)
    {
        InvokeAsync(StateHasChanged); // Re-render component
    }

    // 5. Cleanup
    public void Dispose()
    {
        timer?.Dispose();
    }
}
```

Data Binding in Blazor

One-way and Two-way Binding

```
@* Pages/DataBindingDemo.razor *@
@page "/databinding"

<h3>Data Binding Demo</h3>

<div class="row">
    <div class="col-md-6">
        <!-- One-way binding -->
        <h4>One-way Binding</h4>
        <p>Current time: @DateTime.Now.ToString("HH:mm:ss")</p>
        <p>User name: @user.Name</p>
        <p>Is active: @user.IsActive</p>

        <!-- Two-way binding -->
        <h4>Two-way Binding</h4>
        <div class="form-group">
            <label>Name:</label>
            <input @bind="user.Name" class="form-control" />
```

```
        </div>

        <div class="form-group">
            <label>Email:</label>
            <input @bind="user.Email" @bind:event="oninput" class="form-
control" />
        </div>

        <div class="form-group">
            <label>Age:</label>
            <input @bind="user.Age" type="number" class="form-control" />
        </div>

        <div class="form-group">
            <label>
                <input type="checkbox" @bind="user.IsActive" />
                Is Active
            </label>
        </div>

        <div class="form-group">
            <label>Country:</label>
            <select @bind="user.Country" class="form-control">
                <option value="">Select Country</option>
                <option value="US">United States</option>
                <option value="CA">Canada</option>
                <option value="UK">United Kingdom</option>
            </select>
        </div>
    </div>

    <div class="col-md-6">
        <h4>Live Preview</h4>
        <div class="card">
            <div class="card-body">
                <h5>@user.Name</h5>
                <p>Email: @user.Email</p>
                <p>Age: @user.Age</p>
                <p>Country: @user.Country</p>
                <p>Status: @(user.IsActive ? "Active" : "Inactive")</p>
0
```

```
            </div>
        </div>
    </div>
</div>

@code {
    private User user = new User { Name = "John Doe", Age = 30, IsActive
= true };

    public class User
    {
        public string Name { get; set; } = "";
        public string Email { get; set; } = "";
        public int Age { get; set; }
        public bool IsActive { get; set; }
        public string Country { get; set; } = "";
    }
}
```

Form Validation

```
@* Pages/UserForm.razor *@
@page "/userform"
@using System.ComponentModel.DataAnnotations

<EditForm Model="user" OnValidSubmit="HandleValidSubmit"
OnInvalidSubmit="HandleInvalidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div class="form-group">
        <label>First Name:</label>
        <InputText @bind-Value="user.FirstName" class="form-control" />
        <ValidationMessage For="@(() => user.FirstName)" />
    </div>

    <div class="form-group">
        <label>Email:</label>
        <InputText @bind-Value="user.Email" class="form-control" />
```

0

```
        <ValidationMessage For="@(() => user.Email)" />
    </div>

    <div class="form-group">
        <label>Birth Date:</label>
        <InputDate @bind-Value="user.BirthDate" class="form-control" />
        <ValidationMessage For="@(() => user.BirthDate)" />
    </div>

    <div class="form-group">
        <label>Salary:</label>
        <InputNumber @bind-Value="user.Salary" class="form-control" />
        <ValidationMessage For="@(() => user.Salary)" />
    </div>

    <button type="submit" class="btn btn-primary">Submit</button>
</EditForm>

@if (isSubmitted)
{
    <div class="alert alert-success mt-3">
        Form submitted successfully!
    </div>
}

@code {
    private UserModel user = new UserModel();
    private bool isSubmitted = false;

    private void HandleValidSubmit()
    {
        isSubmitted = true;
        // Process valid form
    }

    private void HandleInvalidSubmit()
    {
        isSubmitted = false;
        // Handle invalid form
    }
```

0

```
    public class UserModel
    {
        [Required(ErrorMessage = "First name is required")]
        [StringLength(50, ErrorMessage = "First name cannot exceed 50
characters")]
        public string FirstName { get; set; } = "";

        [Required(ErrorMessage = "Email is required")]
        [EmailAddress(ErrorMessage = "Invalid email format")]
        public string Email { get; set; } = "";

        [Required(ErrorMessage = "Birth date is required")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; } =
DateTime.Today.AddYears(-18);

        [Range(0, double.MaxValue, ErrorMessage = "Salary must be
positive")]
        public decimal Salary { get; set; }
    }
}
```

JavaScript Interop

```
@* Components/JSInteropDemo.razor *@
@inject IJSRuntime JSRuntime

<h3>JavaScript Interop Demo</h3>

<button @onclick="ShowAlert">Show Alert</button>
<button @onclick="CallJSFunction">Call JS Function</button>
<button @onclick="GetUserLocation">Get Location</button>

<div id="map" style="height: 300px; width: 100%; margin-top: 20px;">
</div>

@code {
    private async Task ShowAlert()
```

```
    {
        await JSRuntime.InvokeVoidAsync("alert", "Hello from Blazor!");
    }

    private async Task CallJSFunction()
    {
        var result = await JSRuntime.InvokeAsync<string>("prompt", "Enter
your name:");
        if (!string.IsNullOrEmpty(result))
        {
            await JSRuntime.InvokeVoidAsync("console.log", $"User
entered: {result}");
        }
    }

    private async Task GetUserLocation()
    {
        try
        {
            var location = await JSRuntime.InvokeAsync<LocationData>
("getUserLocation");
            await JSRuntime.InvokeVoidAsync("initializeMap",
location.Latitude, location.Longitude);
        }
        catch (Exception ex)
        {
            await JSRuntime.InvokeVoidAsync("console.error", $"Error
getting location: {ex.Message}");
        }
    }

    public class LocationData
    {
        public double Latitude { get; set; }
        public double Longitude { get; set; }
    }
}
```

0

```
// wwwroot/js/site.js
window.getUserLocation = () => {
    return new Promise((resolve, reject) => {
        if (navigator.geolocation) {
            navigator.geolocation.getCurrentPosition(
                position => {
                    resolve({
                        latitude: position.coords.latitude,
                        longitude: position.coords.longitude
                    });
                },
                error => reject(error)
            );
        } else {
            reject(new Error("Geolocation is not supported"));
        }
    });
};

window.initializeMap = (lat, lng) => {
    // Initialize map with given coordinates
    console.log(`Initializing map at ${lat}, ${lng}`);
};
```

Razor Class Library

Creating a Razor Class Library

```
<!-- MyBlazorLibrary.csproj -->
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <SupportedPlatform Include="browser" />
```

```
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components.Web"
Version="6.0.0" />
  </ItemGroup>

</Project>
```

Shared Components in Library

```
@* Components/SharedButton.razor *@
@namespace MyBlazorLibrary.Components

<button class="btn @CssClass" @onclick="OnClick">
    @if (!string.IsNullOrEmpty(Icon))
    {
        <i class="@Icon"></i>
    }
    @Text
</button>

@code {
    [Parameter] public string Text { get; set; } = "Button";
    [Parameter] public string Icon { get; set; } = "";
    [Parameter] public string CssClass { get; set; } = "btn-primary";
    [Parameter] public EventCallback OnClick { get; set; }
}
```

Using Library in Main Project

```
@* Add to _Imports.razor *@
@using MyBlazorLibrary.Components

@* Use in components *@
<SharedButton Text="Save"
              Icon="fas fa-save"


0
```

```
                CssClass="btn-success"
                OnClick="SaveData" />
```

State Management

```
// Services/AppStateService.cs
public class AppStateService
{
    private string currentUser = "";
    private Dictionary<string, object> state = new();

    public event Action OnChange;

    public string CurrentUser
    {
        get => currentUser;
        set
        {
            currentUser = value;
            NotifyStateChanged();
        }
    }

    public T GetState<T>(string key, T defaultValue = default)
    {
        if (state.TryGetValue(key, out var value) && value is T)
        {
            return (T)value;
        }
        return defaultValue;
    }

    public void SetState<T>(string key, T value)
    {
        state[key] = value;
        NotifyStateChanged();
    }

    private void NotifyStateChanged() => OnChange?.Invoke();
```

0

```
}

// Program.cs
builder.Services.AddScoped<AppStateService>();

// Component usage
@inject AppStateService AppState
@implements IDisposable

@code {
    protected override void OnInitialized()
    {
        AppState.OnChange += StateHasChanged;
    }

    public void Dispose()
    {
        AppState.OnChange -= StateHasChanged;
    }
}
```

Xamarin

Xamarin vs Xamarin.Forms

Feature	Xamarin.iOS/Android	Xamarin.Forms
UI	Platform-specific UI	Shared UI across platforms
Performance	Native performance	Near-native performance
Code Sharing	Business logic only	UI + Business logic
Learning Curve	Requires platform knowledge	Easier for beginners
Customization	Full platform control	Limited to Forms controls
Use Case	Complex, platform-specific apps	Cross-platform business apps

MVC and MVVM Design Patterns

MVC Pattern in Xamarin

0

```
// Model
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}

// Controller (in Xamarin, this is often the Page code-behind)
public partial class ProductListPage : ContentPage
{
    private readonly ProductService productService;
    private List<Product> products;

    public ProductListPage()
    {
        InitializeComponent();
        productService = new ProductService();
        LoadProducts();
    }

    private async void LoadProducts()
    {
        try
        {
            products = await productService.GetProductsAsync();
            ProductListView.ItemsSource = products;
        }
        catch (Exception ex)
        {
            await DisplayAlert("Error", $"Failed to load products:
{ex.Message}", "OK");
        }
    }

    private async void OnProductSelected(object sender,
SelectedItemChangedEventArgs e)
    {
```

0

```
        if (e.SelectedItem is Product product)
        {
            await Navigation.PushAsync(new ProductDetailPage(product));
        }
    }
}
```

MVVM Pattern in Xamarin.Forms

```
// ViewModel
public class ProductListViewModel : INotifyPropertyChanged
{
    private readonly ProductService productService;
    private ObservableCollection<Product> products;
    private Product selectedProduct;
    private bool isLoading;
    private string searchText;

    public ProductListViewModel()
    {
        productService = DependencyService.Get<ProductService>();
        Products = new ObservableCollection<Product>();

        LoadProductsCommand = new Command(async () => await
LoadProducts());
        SearchCommand = new Command<string>(async (searchText) => await
SearchProducts(searchText));
        SelectProductCommand = new Command<Product>(async (product) =>
await SelectProduct(product));

        LoadProductsCommand.Execute(null);
    }

    public ObservableCollection<Product> Products
    {
        get => products;
        set
        {
            products = value;
```

0

```
            OnPropertyChanged();
        }
    }

    public Product SelectedProduct
    {
        get => selectedProduct;
        set
        {
            selectedProduct = value;
            OnPropertyChanged();
        }
    }

    public bool IsLoading
    {
        get => isLoading;
        set
        {
            isLoading = value;
            OnPropertyChanged();
        }
    }

    public string SearchText
    {
        get => searchText;
        set
        {
            searchText = value;
            OnPropertyChanged();
            SearchCommand.Execute(value);
        }
    }

    public ICommand LoadProductsCommand { get; }
    public ICommand SearchCommand { get; }
    public ICommand SelectProductCommand { get; }

    private async Task LoadProducts()
```

0

```
    {
        IsLoading = true;
        try
        {
            var productList = await productService.GetProductsAsync();
            Products.Clear();
            foreach (var product in productList)
            {
                Products.Add(product);
            }
        }
        catch (Exception ex)
        {
            // Handle error
            await Application.Current.MainPage.DisplayAlert("Error",
ex.Message, "OK");
        }
        finally
        {
            IsLoading = false;
        }
    }

    private async Task SearchProducts(string searchText)
    {
        if (string.IsNullOrWhiteSpace(searchText))
        {
            await LoadProducts();
            return;
        }

        var filtered = await
productService.SearchProductsAsync(searchText);
        Products.Clear();
        foreach (var product in filtered)
        {
            Products.Add(product);
        }
    }
```

0

```
    private async Task SelectProduct(Product product)
    {
        if (product != null)
        {
            await Shell.Current.GoToAsync($"productdetail?productId=
{product.Id}");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

XAML and Key Elements

Basic XAML Structure

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MyApp.Views.ProductListPage"
             Title="Products">

    <ContentPage.BindingContext>
        <vm:ProductListViewModel />
    </ContentPage.BindingContext>

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
                     IconImageSource="add_icon.png"
                     Command="{Binding AddProductCommand}" />
    </ContentPage.ToolbarItems>

0
```

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <!-- Search Bar -->
        <SearchBar Grid.Row="0"
                   Placeholder="Search products..."
                   Text="{Binding SearchText}"
                   SearchCommand="{Binding SearchCommand}"
                   Margin="10" />

        <!-- Product List -->
        <CollectionView Grid.Row="1"
                        ItemsSource="{Binding Products}"
                        SelectedItem="{Binding SelectedProduct}"
                        SelectionMode="Single">

            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <Grid Padding="15">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="80" />
                            <ColumnDefinition Width="*" />
                            <ColumnDefinition Width="Auto" />
                        </Grid.ColumnDefinitions>

                        <Image Grid.Column="0"
                               Source="{Binding ImageUrl}"
                               Aspect="AspectFill"
                               HeightRequest="60"
                               WidthRequest="60" />

                        <StackLayout Grid.Column="1"
                                     Spacing="5"
                                     Margin="10,0">
                            <Label Text="{Binding Name}"
                                   FontSize="16"
                                   FontAttributes="Bold" />
```

```
                            <Label Text="{Binding Description}"
                                   FontSize="14"
                                   TextColor="Gray" />
                        </StackLayout>

                        <Label Grid.Column="2"
                               Text="{Binding Price,
StringFormat='{0:C}'}"
                               FontSize="16"
                               FontAttributes="Bold"
                               VerticalOptions="Center" />
                    </Grid>
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>

        <!-- Loading Indicator -->
        <ActivityIndicator Grid.RowSpan="2"
                           IsVisible="{Binding IsLoading}"
                           IsRunning="{Binding IsLoading}"
                           Color="Blue"
                           HorizontalOptions="Center"
                           VerticalOptions="Center" />
    </Grid>
</ContentPage>
```

SQLite.NET in Xamarin Apps

Database Model

```
[Table("Products")]
public class Product
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }

    [MaxLength(100), NotNull]
    public string Name { get; set; }
```

0

```
    public string Description { get; set; }

    [NotNull]
    public decimal Price { get; set; }

    [MaxLength(50)]
    public string Category { get; set; }

    public DateTime CreatedDate { get; set; } = DateTime.Now;

    public bool IsActive { get; set; } = true;

    // Foreign key
    public int CategoryId { get; set; }
}

[Table("Categories")]
public class Category
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }

    [MaxLength(50), NotNull]
    public string Name { get; set; }

    public string Description { get; set; }
}
```

Database Service

```
public class DatabaseService
{
    private SQLiteAsyncConnection database;
    private readonly string databasePath;

    public DatabaseService()
    {
        databasePath =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalAppli
```

0

```
"Products.db3");
    }

    private async Task InitializeDatabaseAsync()
    {
        if (database is not null)
            return;

        database = new SQLiteAsyncConnection(databasePath);

        await database.CreateTableAsync<Product>();
        await database.CreateTableAsync<Category>();

        // Seed data if needed
        await SeedDataAsync();
    }

    // CRUD Operations for Products
    public async Task<List<Product>> GetProductsAsync()
    {
        await InitializeDatabaseAsync();
        return await database.Table<Product>()
            .Where(p => p.IsActive)
            .OrderBy(p => p.Name)
            .ToListAsync();
    }

    public async Task<Product> GetProductByIdAsync(int id)
    {
        await InitializeDatabaseAsync();
        return await database.Table<Product>()
            .Where(p => p.Id == id)
            .FirstOrDefaultAsync();
    }

    public async Task<int> SaveProductAsync(Product product)
    {
        await InitializeDatabaseAsync();

        if (product.Id != 0)
```

```
        {
            return await database.UpdateAsync(product);
        }
        else
        {
            return await database.InsertAsync(product);
        }
    }

    public async Task<int> DeleteProductAsync(Product product)
    {
        await InitializeDatabaseAsync();
        return await database.DeleteAsync(product);
    }

    public async Task<List<Product>> SearchProductsAsync(string
searchText)
    {
        await InitializeDatabaseAsync();
        return await database.Table<Product>()
            .Where(p => p.IsActive &&
                   (p.Name.Contains(searchText) ||
p.Description.Contains(searchText)))
            .ToListAsync();
    }

    // Category operations
    public async Task<List<Category>> GetCategoriesAsync()
    {
        await InitializeDatabaseAsync();
        return await database.Table<Category>().ToListAsync();
    }

    private async Task SeedDataAsync()
    {
        var categoryCount = await database.Table<Category>
().CountAsync();
        if (categoryCount == 0)
        {
            var categories = new List<Category>
```

```
            {
                new Category { Name = "Electronics", Description =
"Electronic devices" },
                new Category { Name = "Clothing", Description = "Apparel
and accessories" },
                new Category { Name = "Books", Description = "Books and
literature" }
            };

            await database.InsertAllAsync(categories);
        }
    }
}
```

Navigation Patterns in Xamarin.Forms

Shell Navigation

```
// AppShell.xaml
<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:MyApp.Views"
       x:Class="MyApp.AppShell">

    <TabBar>
        <ShellContent Title="Products"
                      Icon="products_icon.png"
                      ContentTemplate="{DataTemplate
views:ProductListPage}" />

        <ShellContent Title="Categories"
                      Icon="categories_icon.png"
                      ContentTemplate="{DataTemplate
views:CategoriesPage}" />

        <ShellContent Title="Profile"
                      Icon="profile_icon.png"
                      ContentTemplate="{DataTemplate views:ProfilePage}"
```

```
/>
    </TabBar>

</Shell>
```

```
// Registering routes
public partial class AppShell : Shell
{
    public AppShell()
    {
        InitializeComponent();

        // Register routes for navigation
        Routing.RegisterRoute("productdetail",
typeof(ProductDetailPage));
        Routing.RegisterRoute("editproduct", typeof(EditProductPage));
        Routing.RegisterRoute("addproduct", typeof(AddProductPage));
    }
}

// Navigation in ViewModels
public class ProductListViewModel : BaseViewModel
{
    private async Task NavigateToProductDetail(Product product)
    {
        var route = $"productdetail?productId={product.Id}";
        await Shell.Current.GoToAsync(route);
    }

    private async Task NavigateToAddProduct()
    {
        await Shell.Current.GoToAsync("addproduct");
    }

    private async Task NavigateBack()
    {
        await Shell.Current.GoToAsync("..");
    }
}
```

0

```
// Receiving parameters
[QueryProperty(nameof(ProductId), "productId")]
public partial class ProductDetailPage : ContentPage
{
    public string ProductId
    {
        set
        {
            if (int.TryParse(value, out int id))
            {
                ((ProductDetailViewModel)BindingContext).LoadProduct(id);
            }
        }
    }
}
```

Traditional Navigation

```
public class NavigationService : INavigationService
{
    public async Task NavigateToAsync(string pageName, object parameter =
null)
    {
        Page page = pageName switch
        {
            "ProductDetail" => new ProductDetailPage(),
            "EditProduct" => new EditProductPage(),
            "AddProduct" => new AddProductPage(),
            _ => throw new ArgumentException($"Unknown page: {pageName}")
        };

        if (page.BindingContext is BaseViewModel viewModel && parameter
!= null)
        {
            await viewModel.InitializeAsync(parameter);
        }

        await Application.Current.MainPage.Navigation.PushAsync(page);
```

0

```
    }

    public async Task NavigateBackAsync()
    {
        await Application.Current.MainPage.Navigation.PopAsync();
    }

    public async Task NavigateToModalAsync(string pageName, object
parameter = null)
    {
        Page page = CreatePage(pageName);

        if (page.BindingContext is BaseViewModel viewModel && parameter
!= null)
        {
            await viewModel.InitializeAsync(parameter);
        }

        await Application.Current.MainPage.Navigation.PushModalAsync(new
NavigationPage(page));
    }
}
```

Summary and Quick Reference

Key Concepts Checklist

Programming Fundamentals

- âœ“ **Visual vs Text Programming**: Visual uses drag-drop; Text uses code
- âœ“ **Event-Driven Programming**: Program flow controlled by events (clicks, input)
- âœ“ **.NET Architecture**: CLR + BCL + Your App
- âœ“ **RAD Tools**: Visual Studio, IntelliSense, designers for rapid development

C# Language Features

- âœ“ **Type Conversion**: Implicit (safe), Explicit (cast), Boxing/Unboxing
- âœ“ **Structures**: Value types, immutable, no inheritance
- âœ“ **Enumerations**: Named constants, type-safe, can use flags

0

- âœ **Collections**: Generic (List, Dictionary<K,V>) vs Non-generic (ArrayList)
- âœ **Regex**: Pattern matching with Regex class

OOP Principles

- âœ **Polymorphism**: Overloading (compile-time) vs Overriding (runtime)
- âœ **Abstract vs Interface**: Abstract has implementation; Interface is contract
- âœ **Inheritance**: IS-A relationship, virtual/override for polymorphism
- âœ **Encapsulation**: Private fields, public properties, controlled access

Advanced Topics

- âœ **Exception Handling**: try-catch-finally, custom exceptions, proper cleanup
- âœ **Parallel Programming**: Tasks, Parallel.For, PLINQ, async/await
- âœ **ADO.NET**: DataReader (connected) vs DataSet (disconnected)
- âœ **LINQ to SQL**: Object-relational mapping with strongly-typed queries

UI Technologies

- âœ **WPF**: XAML + C#, data binding, MVVM pattern, rich desktop apps
- âœ **ASP.NET Core**: MVC pattern, middleware pipeline, Razor Pages
- âœ **Blazor**: C# in browser, Server vs WebAssembly, component-based
- âœ **Xamarin**: Cross-platform mobile, XAML UI, SQLite data, Shell navigation

Exam Questions and Answers

Question 1: Explain the value type and reference type in C# with example.

Value Types

Value types store data directly in memory (usually on the stack). When you assign a value type to another variable, a copy of the data is created.

Characteristics:

- Stored on stack (for local variables)
- Direct memory allocation
- Assignment creates a copy
- Cannot be null (except nullable types)
- Examples: int, float, double, bool, char, struct, enum

0

Example:

```
// Value type example
int a = 10;
int b = a;    // Copy of 'a' is created
a = 20;       // Only 'a' changes, 'b' remains 10

Console.WriteLine($"a = {a}"); // Output: a = 20
Console.WriteLine($"b = {b}"); // Output: b = 10

// Struct example (value type)
public struct Point
{
    public int X, Y;
    public Point(int x, int y) { X = x; Y = y; }
}

Point p1 = new Point(5, 10);
Point p2 = p1;    // Copy created
p1.X = 15;        // Only p1 changes
Console.WriteLine($"p1.X = {p1.X}, p2.X = {p2.X}"); // Output: p1.X = 15,
p2.X = 5
```

Reference Types

Reference types store a reference (address) to the actual data location in memory (heap). When you assign a reference type, both variables point to the same memory location.

Characteristics:

- Stored on heap
- Assignment copies the reference, not the data
- Multiple variables can reference the same object
- Can be null
- Examples: string, object, class, interface, delegate, array

Example:

```
// Reference type example
class Person
0 {
```

```
    public string Name { get; set; }
    public int Age { get; set; }
}

Person person1 = new Person { Name = "John", Age = 25 };
Person person2 = person1;    // Both variables reference same object

person1.Age = 30;            // Changes affect both references
Console.WriteLine($"person1.Age = {person1.Age}"); // Output: person1.Age
= 30
Console.WriteLine($"person2.Age = {person2.Age}"); // Output: person2.Age
= 30

// Array example (reference type)
int[] array1 = { 1, 2, 3 };
int[] array2 = array1;       // Both reference same array
array1[0] = 10;              // Changes affect both
Console.WriteLine($"array2[0] = {array2[0]}"); // Output: array2[0] = 10
```

Memory Allocation Comparison

```
public void CompareTypes()
{
    // Value type - stored on stack
    int valueType = 42;

    // Reference type - reference on stack, object on heap
    string referenceType = "Hello";

    // Nullable value type
    int? nullableInt = null;  // Can be null

    // Boxing: value type -> reference type
    object boxed = valueType;  // valueType copied to heap

    // Unboxing: reference type -> value type
    int unboxed = (int)boxed;  // Copy from heap to stack
}
```

0

Question 2: What is an event in C#? Explain how to implement event using delegate in C#.NET?

What is an Event?

An event in C# is a special kind of multicast delegate that provides notifications when something of interest happens. Events enable a class to notify other classes when something occurs, following the publisher-subscriber pattern.

Key Characteristics:

- Based on delegates
- Encapsulated (cannot be directly invoked from outside the class)
- Supports multiple subscribers
- Provides loose coupling between publisher and subscriber

Event Implementation Using Delegates

Step 1: Define Delegate and Event

```
// Define delegate for event handler
public delegate void NotificationEventHandler(string message);

public class Publisher
{
    // Declare event based on delegate
    public event NotificationEventHandler OnNotification;

    // Method to raise the event
    protected virtual void RaiseNotification(string message)
    {
        // Check if there are subscribers before raising event
        OnNotification?.Invoke(message);
    }

    // Method that triggers the event
    public void DoSomething()
    {
        Console.WriteLine("Publisher: Doing some work...");

        // Trigger the event
```

0

```
        RaiseNotification("Work completed successfully!");
    }
}
```

Step 2: Create Subscribers

```
public class Subscriber1
{
    public void Subscribe(Publisher pub)
    {
        // Subscribe to the event
        pub.OnNotification += HandleNotification;
    }

    public void Unsubscribe(Publisher pub)
    {
        // Unsubscribe from the event
        pub.OnNotification -= HandleNotification;
    }

    private void HandleNotification(string message)
    {
        Console.WriteLine($"Subscriber1 received: {message}");
    }
}

public class Subscriber2
{
    public void Subscribe(Publisher pub)
    {
        pub.OnNotification += HandleNotification;
    }

    private void HandleNotification(string message)
    {
        Console.WriteLine($"Subscriber2 received: {message}");
    }
}
```

0 **Step 3: Usage Example**

```
public class EventDemo
{
    public static void Main()
    {
        // Create publisher and subscribers
        Publisher publisher = new Publisher();
        Subscriber1 sub1 = new Subscriber1();
        Subscriber2 sub2 = new Subscriber2();

        // Subscribe to events
        sub1.Subscribe(publisher);
        sub2.Subscribe(publisher);

        // Trigger event
        publisher.DoSomething();

        // Output:
        // Publisher: Doing some work...
        // Subscriber1 received: Work completed successfully!
        // Subscriber2 received: Work completed successfully!

        // Unsubscribe one subscriber
        sub1.Unsubscribe(publisher);

        // Trigger event again
        publisher.DoSomething();

        // Output:
        // Publisher: Doing some work...
        // Subscriber2 received: Work completed successfully!
    }
}
```

Advanced Event Example with EventArgs

```
// Custom EventArgs class
public class OrderEventArgs : EventArgs
{
    public string OrderId { get; set; }
```

0

```
    public decimal Amount { get; set; }
    public DateTime OrderDate { get; set; }
}

// Publisher class
public class OrderProcessor
{
    // Event using EventHandler<T> generic delegate
    public event EventHandler<OrderEventArgs> OrderProcessed;

    public void ProcessOrder(string orderId, decimal amount)
    {
        Console.WriteLine($"Processing order {orderId}...");

        // Simulate processing
        Thread.Sleep(1000);

        // Raise event with custom data
        OnOrderProcessed(new OrderEventArgs
        {
            OrderId = orderId,
            Amount = amount,
            OrderDate = DateTime.Now
        });
    }

    protected virtual void OnOrderProcessed(OrderEventArgs e)
    {
        OrderProcessed?.Invoke(this, e);
    }
}

// Subscriber classes
public class EmailNotificationService
{
    public void Subscribe(OrderProcessor processor)
    {
        processor.OrderProcessed += OnOrderProcessed;
    }
```

0

```
    private void OnOrderProcessed(object sender, OrderEventArgs e)
    {
        Console.WriteLine($"Email: Order {e.OrderId} for ${e.Amount}
processed at {e.OrderDate}");
    }
}

public class InventoryService
{
    public void Subscribe(OrderProcessor processor)
    {
        processor.OrderProcessed += OnOrderProcessed;
    }

    private void OnOrderProcessed(object sender, OrderEventArgs e)
    {
        Console.WriteLine($"Inventory: Updating stock for order
{e.OrderId}");
    }
}
```

Question 3: What is deferred evaluation? Discuss different standard LINQ operators.

Deferred Evaluation

Deferred evaluation (also called lazy evaluation) means that the execution of a LINQ query is delayed until the results are actually needed. The query is not executed when it's defined, but when it's enumerated.

Key Points:

- Query execution is postponed until results are accessed
- Allows for query optimization
- Enables composition of multiple queries
- Results reflect current data state when enumerated

Example:

0

```
public void DeferredEvaluationExample()
{
    List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

    // Query is defined but NOT executed yet
    var evenNumbers = numbers.Where(n => n % 2 == 0);

    Console.WriteLine("Before adding new numbers:");
    foreach (int num in evenNumbers)  // Query executes HERE
    {
        Console.WriteLine(num);  // Output: 2, 4
    }

    // Add more numbers
    numbers.AddRange(new[] { 6, 7, 8 });

    Console.WriteLine("After adding new numbers:");
    foreach (int num in evenNumbers)  // Query executes AGAIN with new
data
    {
        Console.WriteLine(num);  // Output: 2, 4, 6, 8
    }
}
```

Standard LINQ Operators

1. Filtering Operators

```
var numbers = Enumerable.Range(1, 20);

// Where - filters based on predicate
var evenNumbers = numbers.Where(n => n % 2 == 0);

// OfType - filters by type
object[] mixed = { 1, "hello", 3.14, "world", 42 };
var strings = mixed.OfType<string>();
```

2. Projection Operators

0

```
var people = new[]
{
    new { Name = "John", Age = 25 },
    new { Name = "Jane", Age = 30 },
    new { Name = "Bob", Age = 35 }
};

// Select - transforms each element
var names = people.Select(p => p.Name);
var upperNames = people.Select(p => p.Name.ToUpper());

// SelectMany - flattens nested collections
var sentences = new[] { "Hello world", "LINQ is powerful" };
var words = sentences.SelectMany(s => s.Split(' '));
```

3. Ordering Operators

```
var products = new[]
{
    new { Name = "Laptop", Price = 1200 },
    new { Name = "Mouse", Price = 25 },
    new { Name = "Keyboard", Price = 80 }
};

// OrderBy - ascending order
var byPrice = products.OrderBy(p => p.Price);

// OrderByDescending - descending order
var byPriceDesc = products.OrderByDescending(p => p.Price);

// ThenBy - secondary sorting
var sorted = products.OrderBy(p => p.Name).ThenBy(p => p.Price);

// Reverse - reverses order
var reversed = products.Reverse();
```

4. Grouping Operators

0

```
var students = new[]
{
    new { Name = "John", Grade = "A", Subject = "Math" },
    new { Name = "Jane", Grade = "B", Subject = "Math" },
    new { Name = "Bob", Grade = "A", Subject = "Science" }
};

// GroupBy - groups elements by key
var byGrade = students.GroupBy(s => s.Grade);
foreach (var group in byGrade)
{
    Console.WriteLine($"Grade {group.Key}:");
    foreach (var student in group)
    {
        Console.WriteLine($"  {student.Name}");
    }
}
```

5. Aggregation Operators

```
var scores = new[] { 85, 92, 78, 96, 88 };

// Count - number of elements
int totalCount = scores.Count();
int highScores = scores.Count(s => s > 90);

// Sum, Average, Min, Max
int total = scores.Sum();
double average = scores.Average();
int minimum = scores.Min();
int maximum = scores.Max();

// Aggregate - custom aggregation
int product = scores.Aggregate((a, b) => a * b);
```

6. Set Operators

```
var list1 = new[] { 1, 2, 3, 4, 5 };
var list2 = new[] { 4, 5, 6, 7, 8 };
```

0

```
// Distinct - removes duplicates
var duplicates = new[] { 1, 2, 2, 3, 3, 4 };
var unique = duplicates.Distinct();

// Union - combines and removes duplicates
var union = list1.Union(list2);

// Intersect - common elements
var common = list1.Intersect(list2);

// Except - elements in first but not in second
var difference = list1.Except(list2);
```

7. Element Operators

```
var numbers = new[] { 1, 2, 3, 4, 5 };

// First, FirstOrDefault
int first = numbers.First();
int firstEven = numbers.First(n => n % 2 == 0);
int firstOrDefault = numbers.FirstOrDefault(n => n > 10); // returns 0

// Last, LastOrDefault
int last = numbers.Last();

// Single, SingleOrDefault - exactly one element
var singleNumbers = new[] { 42 };
int single = singleNumbers.Single();

// ElementAt
int thirdElement = numbers.ElementAt(2); // zero-based index
```

8. Quantifier Operators

```
var numbers = new[] { 2, 4, 6, 8, 10 };

// All - checks if all elements satisfy condition
bool allEven = numbers.All(n => n % 2 == 0); // true
```

0

```
// Any - checks if any element satisfies condition
bool hasLarge = numbers.Any(n => n > 5); // true

// Contains - checks if collection contains element
bool containsFive = numbers.Contains(5); // false
```

9. Join Operators

```
var customers = new[]
{
    new { Id = 1, Name = "John" },
    new { Id = 2, Name = "Jane" }
};

var orders = new[]
{
    new { CustomerId = 1, Product = "Laptop" },
    new { CustomerId = 1, Product = "Mouse" },
    new { CustomerId = 2, Product = "Keyboard" }
};

// Join - inner join
var customerOrders = customers.Join(
    orders,
    customer => customer.Id,
    order => order.CustomerId,
    (customer, order) => new { customer.Name, order.Product }
);

// GroupJoin - left outer join
var customerOrderGroups = customers.GroupJoin(
    orders,
    customer => customer.Id,
    order => order.CustomerId,
    (customer, orderGroup) => new { customer.Name, Orders = orderGroup }
);
```

0

Question 4: Why serialization is required? Write a program to write user input into file and display file content using stream.

Why Serialization is Required

Serialization is the process of converting an object into a format that can be stored or transmitted, and later reconstructed. It's required for:

1. **Data Persistence**: Store object state to disk, database, or other storage
2. **Network Communication**: Send objects between applications over network
3. **Caching**: Store objects in memory or external cache systems
4. **Deep Copying**: Create exact copies of complex objects
5. **Interoperability**: Exchange data between different platforms/languages
6. **State Management**: Save and restore application state

Types of Serialization:

- **Binary Serialization**: Compact, fast, .NET specific
- **XML Serialization**: Human-readable, cross-platform
- **JSON Serialization**: Lightweight, web-friendly, cross-platform

File I/O Program with Streams

```
using System;
using System.IO;
using System.Text;
using System.Text.Json;

// Class to demonstrate serialization
[Serializable]
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
    public decimal Salary { get; set; }
    public DateTime JoiningDate { get; set; }

    public override string ToString()
    {
        return $"ID: {Id}, Name: {Name}, Department: {Department},
```

```
Salary: {Salary:C}, Joining: {JoiningDate:yyyy-MM-dd}";
    }
}

public class FileOperationsProgram
{
    private const string TextFileName = "employee_data.txt";
    private const string JsonFileName = "employee_data.json";

    public static void Main()
    {
        Console.WriteLine("=== File Operations with Streams Demo ===\n");

        // Get user input
        Employee employee = GetEmployeeFromUser();

        // Demonstrate different file operations
        WriteToTextFile(employee);
        ReadFromTextFile();

        WriteToJsonFile(employee);
        ReadFromJsonFile();

        // Stream operations demonstration
        DemonstrateStreamOperations();

        Console.WriteLine("\nPress any key to exit...");
        Console.ReadKey();
    }

    // Get employee data from user input
    private static Employee GetEmployeeFromUser()
    {
        Console.WriteLine("Enter Employee Details:");

        Console.Write("ID: ");
        int id = int.Parse(Console.ReadLine());

        Console.Write("Name: ");
        string name = Console.ReadLine();
```

0

```
        Console.Write("Department: ");
        string department = Console.ReadLine();

        Console.Write("Salary: ");
        decimal salary = decimal.Parse(Console.ReadLine());

        Console.Write("Joining Date (yyyy-mm-dd): ");
        DateTime joiningDate = DateTime.Parse(Console.ReadLine());

        return new Employee
        {
            Id = id,
            Name = name,
            Department = department,
            Salary = salary,
            JoiningDate = joiningDate
        };
    }

    // Write to text file using StreamWriter
    private static void WriteToTextFile(Employee employee)
    {
        try
        {
            using (FileStream fileStream = new FileStream(TextFileName,
FileMode.Create))
            using (StreamWriter writer = new StreamWriter(fileStream,
Encoding.UTF8))
            {
                writer.WriteLine("=== Employee Information ===");
                writer.WriteLine($"ID: {employee.Id}");
                writer.WriteLine($"Name: {employee.Name}");
                writer.WriteLine($"Department: {employee.Department}");
                writer.WriteLine($"Salary: {employee.Salary:C}");
                writer.WriteLine($"Joining Date:
{employee.JoiningDate:yyyy-MM-dd}");
                writer.WriteLine($"File Created: {DateTime.Now}");
            }
```

0

```
            Console.WriteLine($"\nâœ“ Employee data written to
{TextFileName}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error writing to file: {ex.Message}");
        }
    }

    // Read from text file using StreamReader
    private static void ReadFromTextFile()
    {
        try
        {
            using (FileStream fileStream = new FileStream(TextFileName,
FileMode.Open))
            using (StreamReader reader = new StreamReader(fileStream,
Encoding.UTF8))
            {
                Console.WriteLine($"\n=== Contents of {TextFileName}
===");
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    Console.WriteLine(line);
                }
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine($"File {TextFileName} not found.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error reading file: {ex.Message}");
        }
    }

    // Write to JSON file (serialization)
    private static void WriteToJsonFile(Employee employee)
0
```

```
    {
        try
        {
            using (FileStream fileStream = new FileStream(JsonFileName,
FileMode.Create))
            {
                JsonSerializer.Serialize(fileStream, employee, new
JsonSerializerOptions
                {
                    WriteIndented = true // Pretty formatting
                });
            }

            Console.WriteLine($"\nâœ“ Employee data serialized to
{JsonFileName}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error serializing to JSON:
{ex.Message}");
        }
    }

    // Read from JSON file (deserialization)
    private static void ReadFromJsonFile()
    {
        try
        {
            using (FileStream fileStream = new FileStream(JsonFileName,
FileMode.Open))
            {
                Employee deserializedEmployee =
JsonSerializer.Deserialize<Employee>(fileStream);

                Console.WriteLine($"\n=== Deserialized from
{JsonFileName} ===");
                Console.WriteLine(deserializedEmployee.ToString());
            }
        }
        catch (FileNotFoundException)
0
```

```
        {
            Console.WriteLine($"File {JsonFileName} not found.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error deserializing JSON: {ex.Message}");
        }
    }

    // Demonstrate various stream operations
    private static void DemonstrateStreamOperations()
    {
        Console.WriteLine("\n=== Stream Operations Demo ===");

        string demoFileName = "stream_demo.txt";

        try
        {
            // Write using different methods
            using (FileStream fs = new FileStream(demoFileName,
FileMode.Create))
            {
                // Write bytes directly
                byte[] data = Encoding.UTF8.GetBytes("Hello from
FileStream!\n");
                fs.Write(data, 0, data.Length);

                // Write using StreamWriter
                using (StreamWriter sw = new StreamWriter(fs,
Encoding.UTF8, 1024, true))
                {
                    sw.WriteLine("Hello from StreamWriter!");
                    sw.WriteLine($"Current time: {DateTime.Now}");
                    sw.Flush(); // Ensure data is written
                }
            }

            // Read using different methods
            using (FileStream fs = new FileStream(demoFileName,
FileMode.Open))
0
```

```
            {
                Console.WriteLine("File size: " + fs.Length + " bytes");

                // Read all bytes
                byte[] buffer = new byte[fs.Length];
                fs.Read(buffer, 0, buffer.Length);
                string content = Encoding.UTF8.GetString(buffer);

                Console.WriteLine("File contents:");
                Console.WriteLine(content);
            }

            // Memory stream example
            using (MemoryStream ms = new MemoryStream())
            {
                string text = "Hello Memory Stream!";
                byte[] bytes = Encoding.UTF8.GetBytes(text);
                ms.Write(bytes, 0, bytes.Length);

                ms.Position = 0; // Reset position to beginning

                using (StreamReader sr = new StreamReader(ms))
                {
                    string result = sr.ReadToEnd();
                    Console.WriteLine($"From MemoryStream: {result}");
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Stream operation error: {ex.Message}");
        }
        finally
        {
            // Cleanup
            if (File.Exists(demoFileName))
                File.Delete(demoFileName);
        }
```

0

```
    }
}
```

Advanced Stream Example with Custom Serialization

```
public class CustomSerializationExample
{
    public static void BinarySerializationDemo()
    {
        var employees = new List<Employee>
        {
            new Employee { Id = 1, Name = "John Doe", Department = "IT",
Salary = 50000, JoiningDate = DateTime.Now.AddYears(-2) },
            new Employee { Id = 2, Name = "Jane Smith", Department =
"HR", Salary = 45000, JoiningDate = DateTime.Now.AddYears(-1) }
        };

        string fileName = "employees.dat";

        // Write binary data
        using (FileStream fs = new FileStream(fileName, FileMode.Create))
        using (BinaryWriter writer = new BinaryWriter(fs))
        {
            writer.Write(employees.Count);

            foreach (var emp in employees)
            {
                writer.Write(emp.Id);
                writer.Write(emp.Name);
                writer.Write(emp.Department);
                writer.Write(emp.Salary);
                writer.Write(emp.JoiningDate.ToBinary());
            }
        }

        // Read binary data
        using (FileStream fs = new FileStream(fileName, FileMode.Open))
        using (BinaryReader reader = new BinaryReader(fs))
        {
0
```

```
            int count = reader.ReadInt32();

            for (int i = 0; i < count; i++)
            {
                var emp = new Employee
                {
                    Id = reader.ReadInt32(),
                    Name = reader.ReadString(),
                    Department = reader.ReadString(),
                    Salary = reader.ReadDecimal(),
                    JoiningDate = DateTime.FromBinary(reader.ReadInt64())
                };

                Console.WriteLine(emp.ToString());
            }
        }
    }
}
```

Question 5: Differentiate connected architecture of ADO.NET from disconnected architecture of ADO.NET. Write a C# program to connect to database and insert five employee records and delete employee records whose salary is less than Rs 10000.

Connected vs Disconnected Architecture

Aspect	Connected Architecture	Disconnected Architecture
Connection	Maintains active connection	Works with disconnected data
Performance	Faster for single operations	Better for multiple operations
Memory Usage	Less memory usage	More memory usage
Concurrency	Limited concurrent users	Supports many concurrent users
Network Traffic	Continuous network usage	Minimal network usage
Offline Support	No offline capability	Full offline support
Data Modification	Direct database updates	Batch updates possible
Primary Classes	SqlConnection, SqlCommand, SqlDataReader	DataSet, DataTable, SqlDataAdapter

0

Aspect	Connected Architecture	Disconnected Architecture
Use Case	Simple read operations, real-time data	Complex operations, data manipulation

Database Program Implementation

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string JobTitle { get; set; }
    public string Address { get; set; }
    public float Salary { get; set; }
    public DateTime JoiningDate { get; set; }
}

public class EmployeeDataAccess
{
    // Connection string - update with your SQL Server details
    private string connectionString =
@"Server=localhost;Database=Empinfo;Integrated
Security=true;TrustServerCertificate=true;";

    // Alternative connection string for SQL Server Authentication
    // private string connectionString =
@"Server=localhost;Database=Empinfo;User
Id=sa;Password=yourpassword;TrustServerCertificate=true;";

    public static void Main()
    {
        var dataAccess = new EmployeeDataAccess();

        try
        {
```

0

```
            // Create database and table if they don't exist
            dataAccess.CreateDatabaseAndTable();

            // Insert five employee records
            dataAccess.InsertEmployeeRecords();

            // Display all employees before deletion
            Console.WriteLine("=== All Employees Before Deletion ===");
            dataAccess.DisplayAllEmployees();

            // Delete employees with salary less than 10000
            int deletedCount = dataAccess.DeleteLowSalaryEmployees();
            Console.WriteLine($"\n{deletedCount} employee(s) deleted with
salary less than Rs. 10000");

            // Display remaining employees
            Console.WriteLine("\n=== Remaining Employees After Deletion
===");
            dataAccess.DisplayAllEmployees();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        Console.WriteLine("\nPress any key to exit...");
        Console.ReadKey();
    }

    // Create database and table
    private void CreateDatabaseAndTable()
    {
        string masterConnectionString =
connectionString.Replace("Database=Empinfo", "Database=master");

        // Create database
        using (SqlConnection connection = new
SqlConnection(masterConnectionString))
        {
            connection.Open();
```

```
            string createDbQuery = @"
                IF NOT EXISTS (SELECT name FROM sys.databases WHERE name
= 'Empinfo')
                CREATE DATABASE Empinfo";

            using (SqlCommand command = new SqlCommand(createDbQuery,
connection))
            {
                command.ExecuteNonQuery();
                Console.WriteLine("Database 'Empinfo' created or already
exists.");
            }
        }

        // Create table
        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            string createTableQuery = @"
                IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id
= OBJECT_ID(N'[dbo].[Employee]') AND type in (N'U'))
                CREATE TABLE Employee (
                    Id INT PRIMARY KEY IDENTITY(1,1),
                    Name NVARCHAR(50) NOT NULL,
                    JobTitle NVARCHAR(50) NOT NULL,
                    Address NVARCHAR(50) NOT NULL,
                    Salary FLOAT NOT NULL,
                    JoiningDate DATETIME NOT NULL
                )";

            using (SqlCommand command = new SqlCommand(createTableQuery,
connection))
            {
                command.ExecuteNonQuery();
                Console.WriteLine("Table 'Employee' created or already
exists.");
            }
0
```

```
        }
    }

    // Connected Architecture - Insert Employee Records
    private void InsertEmployeeRecords()
    {
        var employees = new[]
        {
            new Employee { Name = "John Doe", JobTitle = "Software
Developer", Address = "123 Main St, City A", Salary = 55000, JoiningDate
= DateTime.Now.AddYears(-2) },
            new Employee { Name = "Jane Smith", JobTitle = "Project
Manager", Address = "456 Oak Ave, City B", Salary = 75000, JoiningDate =
DateTime.Now.AddYears(-3) },
            new Employee { Name = "Mike Johnson", JobTitle = "Junior
Developer", Address = "789 Pine Rd, City C", Salary = 8000, JoiningDate =
DateTime.Now.AddMonths(-6) },
            new Employee { Name = "Sarah Wilson", JobTitle = "Senior
Developer", Address = "321 Elm St, City D", Salary = 85000, JoiningDate =
DateTime.Now.AddYears(-4) },
            new Employee { Name = "Tom Brown", JobTitle = "Intern",
Address = "654 Maple Dr, City E", Salary = 5000, JoiningDate =
DateTime.Now.AddMonths(-3) }
        };

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            // Clear existing data for demo
            string clearQuery = "DELETE FROM Employee";
            using (SqlCommand clearCommand = new SqlCommand(clearQuery,
connection))
            {
                clearCommand.ExecuteNonQuery();
            }

            // Insert new records
            string insertQuery = @"
```

0

```
                INSERT INTO Employee (Name, JobTitle, Address, Salary,
JoiningDate)
                VALUES (@Name, @JobTitle, @Address, @Salary,
@JoiningDate)";

            foreach (var employee in employees)
            {
                using (SqlCommand command = new SqlCommand(insertQuery,
connection))
                {
                    // Using parameters to prevent SQL injection
                    command.Parameters.AddWithValue("@Name",
employee.Name);
                    command.Parameters.AddWithValue("@JobTitle",
employee.JobTitle);
                    command.Parameters.AddWithValue("@Address",
employee.Address);
                    command.Parameters.AddWithValue("@Salary",
employee.Salary);
                    command.Parameters.AddWithValue("@JoiningDate",
employee.JoiningDate);

                    int rowsAffected = command.ExecuteNonQuery();
                    if (rowsAffected > 0)
                    {
                        Console.WriteLine($"Inserted: {employee.Name}");
                    }
                }
            }

            Console.WriteLine("\nAll employee records inserted
successfully!");
        }
    }

    // Connected Architecture - Display All Employees using DataReader
    private void DisplayAllEmployees()
    {
        using (SqlConnection connection = new
SqlConnection(connectionString))
0
```

```
        {
            connection.Open();

            string selectQuery = "SELECT Id, Name, JobTitle, Address,
Salary, JoiningDate FROM Employee ORDER BY Id";

            using (SqlCommand command = new SqlCommand(selectQuery,
connection))
            using (SqlDataReader reader = command.ExecuteReader())
            {
                Console.WriteLine($"{"ID",-5} {"Name",-15} {"Job
Title",-20} {"Address",-25} {"Salary",-10} {"Joining Date",-12}");
                Console.WriteLine(new string('-', 95));

                while (reader.Read())
                {
                    Console.WriteLine($"{reader["Id"],-5} " +
                                      $"{reader["Name"],-15} " +
                                      $"{reader["JobTitle"],-20} " +
                                      $"{reader["Address"],-25} " +
                                      $"{reader["Salary"],-10:F0} " +
                                      $"
{((DateTime)reader["JoiningDate"]).ToString("yyyy-MM-dd"),-12}");
                }
            }
        }
    }

    // Connected Architecture - Delete Low Salary Employees
    private int DeleteLowSalaryEmployees()
    {
        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            string deleteQuery = "DELETE FROM Employee WHERE Salary <
@MinSalary";

            using (SqlCommand command = new SqlCommand(deleteQuery,
```

```
connection))
            {
                command.Parameters.AddWithValue("@MinSalary", 10000);

                int rowsDeleted = command.ExecuteNonQuery();
                return rowsDeleted;
            }
        }
    }
}

// Disconnected Architecture Example
public class DisconnectedEmployeeDataAccess
{
    private string connectionString =
@"Server=localhost;Database=Empinfo;Integrated
Security=true;TrustServerCertificate=true;";

    // Disconnected Architecture - Using DataSet and DataAdapter
    public void DisconnectedOperationsDemo()
    {
        DataSet employeeDataSet = new DataSet("EmployeeData");

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            // Create DataAdapter
            string selectQuery = "SELECT Id, Name, JobTitle, Address,
Salary, JoiningDate FROM Employee";
            SqlDataAdapter adapter = new SqlDataAdapter(selectQuery,
connection);

            // Configure command builders for automatic INSERT, UPDATE,
DELETE commands
            SqlCommandBuilder commandBuilder = new
SqlCommandBuilder(adapter);

            // Fill DataSet (connection opens and closes automatically)
            adapter.Fill(employeeDataSet, "Employees");
```

0

```
            // Work with data offline
            DataTable employeeTable =
employeeDataSet.Tables["Employees"];

            Console.WriteLine("=== Working with Disconnected Data ===");
            Console.WriteLine($"Loaded {employeeTable.Rows.Count}
employees into DataSet");

            // Modify data in memory
            foreach (DataRow row in employeeTable.Rows)
            {
                if ((double)row["Salary"] < 50000)
                {
                    row["Salary"] = (double)row["Salary"] * 1.1; // 10%
raise
                    Console.WriteLine($"Increased salary for
{row["Name"]}");
                }
            }

            // Add new employee
            DataRow newRow = employeeTable.NewRow();
            newRow["Name"] = "Alice Cooper";
            newRow["JobTitle"] = "Data Analyst";
            newRow["Address"] = "999 Data St, City F";
            newRow["Salary"] = 60000;
            newRow["JoiningDate"] = DateTime.Now;
            employeeTable.Rows.Add(newRow);

            // Update database with all changes in one batch
            try
            {
                int updatedRows = adapter.Update(employeeDataSet,
"Employees");
                Console.WriteLine($"Updated {updatedRows} rows in
database");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Update failed: {ex.Message}");
0
```

```
            }
        }
    }
}
```

Question 6: Write a short note on the following

a. Sealed Class and Sealed Method

Sealed Class: A sealed class is a class that cannot be inherited by other classes. It prevents further derivation and is used when you want to restrict the inheritance hierarchy.

Characteristics:

- Cannot be used as a base class
- All members are implicitly sealed
- Can inherit from other classes but cannot be inherited
- Provides performance benefits (no virtual method lookups)

Example:

```
// Sealed class example
public sealed class MathUtility
{
    public static double CalculateCircleArea(double radius)
    {
        return Math.PI * radius * radius;
    }

    public static double CalculateRectangleArea(double length, double
width)
    {
        return length * width;
    }
}

// This would cause compilation error:
// public class ExtendedMath : MathUtility { } // Error: Cannot inherit
from sealed class

0 // Real-world example: String class is sealed
```

```
public void StringExample()
{
    string text = "Hello"; // String is sealed, cannot be inherited
}
```

Sealed Method: A sealed method is used in derived classes to prevent further overriding of a virtual method in subsequent derived classes.

Characteristics:

- Must be used with override keyword
- Prevents further overriding in derived classes
- Can only be applied to virtual or abstract methods
- Maintains the inheritance chain but stops further overriding

Example:

```
// Base class with virtual method
public class Vehicle
{
    public virtual void Start()
    {
        Console.WriteLine("Vehicle starting...");
    }
}

// Derived class that seals the method
public class Car : Vehicle
{
    public sealed override void Start()
    {
        Console.WriteLine("Car engine starting...");
    }
}

// Further derived class cannot override the sealed method
public class SportsCar : Car
{
    // This would cause compilation error:
    // public override void Start() { } // Error: Cannot override sealed
method
```

0

```
    // But can create new method with same name (method hiding)
    public new void Start()
    {
        Console.WriteLine("Sports car turbo engine starting...");
    }
}

// Usage example
public void SealedMethodDemo()
{
    Vehicle vehicle = new SportsCar();
    vehicle.Start(); // Calls Car's sealed Start method

    SportsCar sports = new SportsCar();
    sports.Start(); // Calls SportsCar's new Start method (hiding)
}
```

Practical Use Cases:

```
// Security-sensitive class
public sealed class CryptoHelper
{
    public static string EncryptPassword(string password)
    {
        // Secure encryption logic
        return
Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(password));
    }
}

// Performance-critical class
public sealed class FastCalculator
{
    public double Calculate(double x, double y)
    {
        // No virtual method overhead
        return x * y + Math.Sin(x);
```

0

```
    }
}
```

b. AJAX (Asynchronous JavaScript and XML)

AJAX is a web development technique that allows web pages to update content dynamically without requiring a full page reload. It enables asynchronous communication between the client and server.

Key Characteristics:

- **Asynchronous**: Operations don't block the user interface
- **Partial Updates**: Only specific parts of the page are updated
- **Better User Experience**: Faster, more responsive web applications
- **Reduced Server Load**: Less data transfer and processing

Core Technologies:

1. **JavaScript**: Controls the AJAX behavior
2. **XMLHttpRequest**: Browser API for making HTTP requests
3. **DOM**: Manipulates page content dynamically
4. **CSS**: Styles the dynamic content
5. **Server-side**: Processes requests and returns data (often JSON)

AJAX in ASP.NET Core Example:

Controller (C#):

```
[ApiController]
[Route("api/[controller]")]
public class EmployeeController : ControllerBase
{
    [HttpGet]
    public IActionResult GetEmployees()
    {
        var employees = new[]
        {
            new { Id = 1, Name = "John Doe", Department = "IT" },
            new { Id = 2, Name = "Jane Smith", Department = "HR" },
            new { Id = 3, Name = "Mike Johnson", Department = "Finance" }
        };
```

0

```
        return Ok(employees);
    }

    [HttpPost]
    public IActionResult AddEmployee([FromBody] Employee employee)
    {
        // Add employee logic here
        return Ok(new { Message = "Employee added successfully", Id =
employee.Id });
    }

    [HttpDelete("{id}")]
    public IActionResult DeleteEmployee(int id)
    {
        // Delete employee logic here
        return Ok(new { Message = $"Employee {id} deleted successfully"
});
    }
}
```

JavaScript (Client-side):

```
// Fetch employees using AJAX
function loadEmployees() {
    fetch('/api/employee')
        .then(response => response.json())
        .then(data => {
            const tableBody =
document.getElementById('employeeTableBody');
            tableBody.innerHTML = '';

            data.forEach(employee => {
                const row = `
                    <tr>
                        <td>${employee.id}</td>
                        <td>${employee.name}</td>
                        <td>${employee.department}</td>
                        <td>
                            <button
onclick="deleteEmployee(${employee.id})">Delete</button>
```

```
                        </td>
                    </tr>
                `;
                tableBody.innerHTML += row;
            });
        })
        .catch(error => {
            console.error('Error loading employees:', error);
            alert('Failed to load employees');
        });
}

// Add employee using AJAX
function addEmployee() {
    const employeeData = {
        name: document.getElementById('employeeName').value,
        department: document.getElementById('employeeDepartment').value
    };

    fetch('/api/employee', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(employeeData)
    })
    .then(response => response.json())
    .then(data => {
        alert(data.message);
        loadEmployees(); // Refresh the list
        clearForm();
    })
    .catch(error => {
        console.error('Error adding employee:', error);
        alert('Failed to add employee');
    });
}

// Delete employee using AJAX
function deleteEmployee(id) {
```

0

```
    if (confirm('Are you sure you want to delete this employee?')) {
        fetch(`/api/employee/${id}`, {
            method: 'DELETE'
        })
        .then(response => response.json())
        .then(data => {
            alert(data.message);
            loadEmployees(); // Refresh the list
        })
        .catch(error => {
            console.error('Error deleting employee:', error);
            alert('Failed to delete employee');
        });
    }
}

// Load employees when page loads
document.addEventListener('DOMContentLoaded', function() {
    loadEmployees();
});
```

HTML:

```
<!DOCTYPE html>
<html>
<head>
    <title>AJAX Employee Management</title>
    <style>
        table { border-collapse: collapse; width: 100%; }
        th, td { border: 1px solid #ddd; padding: 8px; text-align: left;
}
        th { background-color: #f2f2f2; }
        .form-container { margin-bottom: 20px; }
        .form-container input, .form-container button { margin: 5px;
padding: 5px; }
    </style>
</head>
<body>
    <h1>Employee Management with AJAX</h1>
```

0

```
    <div class="form-container">
        <h3>Add New Employee</h3>
        <input type="text" id="employeeName" placeholder="Employee Name"
/>
        <input type="text" id="employeeDepartment"
placeholder="Department" />
        <button onclick="addEmployee()">Add Employee</button>
    </div>

    <h3>Employee List</h3>
    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
                <th>Department</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody id="employeeTableBody">
            <!-- Employee data will be loaded here via AJAX -->
        </tbody>
    </table>

    <script src="employee-ajax.js"></script>
</body>
</html>
```

jQuery AJAX Alternative:

```
// Using jQuery for AJAX operations
$(document).ready(function() {
    loadEmployees();
});

function loadEmployees() {
    $.ajax({
        url: '/api/employee',
        type: 'GET',
        dataType: 'json',
```

```
        success: function(data) {
            var html = '';
            $.each(data, function(index, employee) {
                html += '<tr>';
                html += '<td>' + employee.id + '</td>';
                html += '<td>' + employee.name + '</td>';
                html += '<td>' + employee.department + '</td>';
                html += '<td><button onclick="deleteEmployee(' +
employee.id + ')">Delete</button></td>';
                html += '</tr>';
            });
            $('#employeeTableBody').html(html);
        },
        error: function() {
            alert('Failed to load employees');
        }
    });
}

function addEmployee() {
    var employeeData = {
        name: $('#employeeName').val(),
        department: $('#employeeDepartment').val()
    };

    $.ajax({
        url: '/api/employee',
        type: 'POST',
        contentType: 'application/json',
        data: JSON.stringify(employeeData),
        success: function(data) {
            alert(data.message);
            loadEmployees();
            $('#employeeName').val('');
            $('#employeeDepartment').val('');
        },
        error: function() {
            alert('Failed to add employee');
        }
```

0

```
    });
}
```

Benefits of AJAX:

- **Improved User Experience**: No page reloads, faster interactions
- **Reduced Bandwidth**: Only necessary data is transferred
- **Better Performance**: Less server processing and faster response
- **Dynamic Content**: Real-time updates without page refresh
- **Asynchronous Processing**: Non-blocking operations

Common Use Cases:

- Form validation and submission
- Auto-complete and search suggestions
- Loading data dynamically (infinite scroll)
- Real-time notifications and updates
- Single Page Applications (SPAs)

Additional Exam Questions and Answers

Question 1: What is the difference between 'for loop' and 'foreach loop'? Explain with Example in C#.

Differences:

Aspect	for loop	foreach loop
Purpose	General-purpose iteration with index control	Iterating through collections
Index Access	Provides index access	No direct index access
Performance	Slightly faster for arrays	Optimized for collections
Flexibility	Can modify iteration pattern	Fixed iteration pattern
Collection Modification	Can modify collection during iteration	Cannot modify collection during iteration

Examples:

```
public class LoopComparison
{
    public static void ForLoopExample()
    {
        int[] numbers = { 1, 2, 3, 4, 5 };

        // for loop - provides index access
        Console.WriteLine("For loop:");
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine($"Index {i}: {numbers[i]}");
            // Can modify array elements
            numbers[i] *= 2;
        }

        // Can iterate backwards
        Console.WriteLine("\nBackward iteration:");
        for (int i = numbers.Length - 1; i >= 0; i--)
        {
            Console.WriteLine($"Index {i}: {numbers[i]}");
        }

        // Can skip elements
        Console.WriteLine("\nSkip every other element:");
        for (int i = 0; i < numbers.Length; i += 2)
        {
            Console.WriteLine($"Index {i}: {numbers[i]}");
        }
    }

    public static void ForEachLoopExample()
    {
        int[] numbers = { 10, 20, 30, 40, 50 };
        List<string> names = new List<string> { "Alice", "Bob", "Charlie"
};

        // foreach loop - simpler syntax for iteration
        Console.WriteLine("Foreach loop with array:");
        foreach (int number in numbers)
```

0

```
        {
            Console.WriteLine($"Value: {number}");
            // Cannot modify 'number' variable to change array
        }

        // Works with any IEnumerable
        Console.WriteLine("\nForeach loop with List:");
        foreach (string name in names)
        {
            Console.WriteLine($"Name: {name}");
        }

        // Works with Dictionary
        Dictionary<string, int> ages = new Dictionary<string, int>
        {
            ["Alice"] = 25,
            ["Bob"] = 30
        };

        Console.WriteLine("\nForeach loop with Dictionary:");
        foreach (KeyValuePair<string, int> kvp in ages)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }
    }

    public static void PerformanceComparison()
    {
        int[] largeArray = new int[1000000];
        for (int i = 0; i < largeArray.Length; i++)
        {
            largeArray[i] = i;
        }

        Stopwatch sw = new Stopwatch();

        // for loop timing
        sw.Start();
        long sum1 = 0;
        for (int i = 0; i < largeArray.Length; i++)
0
```

```
        {
            sum1 += largeArray[i];
        }
        sw.Stop();
        long forTime = sw.ElapsedMilliseconds;

        // foreach loop timing
        sw.Restart();
        long sum2 = 0;
        foreach (int number in largeArray)
        {
            sum2 += number;
        }
        sw.Stop();
        long foreachTime = sw.ElapsedMilliseconds;

        Console.WriteLine($"For loop time: {forTime}ms");
        Console.WriteLine($"Foreach loop time: {foreachTime}ms");
    }
}
```

Question 2: What is LINQ? What are the advantages of LINQ? Explain with Example.

What is LINQ?

LINQ (Language Integrated Query) is a set of features that extends powerful query capabilities to the language syntax of C# and Visual Basic. It allows you to write queries directly in your programming language.

Advantages of LINQ:

1. **Type Safety** - Compile-time checking
2. **IntelliSense Support** - Auto-completion and syntax checking
3. **Unified Syntax** - Same syntax for different data sources
4. **Deferred Execution** - Queries execute when enumerated
5. **Readable Code** - More intuitive than traditional loops

Examples:

0

```
public class LinqExamples
{
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public string Course { get; set; }
        public double Grade { get; set; }
    }

    public static void LinqBasics()
    {
        List<Student> students = new List<Student>
        {
            new Student { Id = 1, Name = "Alice", Age = 20, Course =
"Computer Science", Grade = 85.5 },
            new Student { Id = 2, Name = "Bob", Age = 22, Course =
"Mathematics", Grade = 92.0 },
            new Student { Id = 3, Name = "Charlie", Age = 19, Course =
"Computer Science", Grade = 78.5 },
            new Student { Id = 4, Name = "Diana", Age = 21, Course =
"Physics", Grade = 88.0 },
            new Student { Id = 5, Name = "Eve", Age = 20, Course =
"Mathematics", Grade = 95.5 }
        };

        // Method Syntax
        Console.WriteLine("Students with grade > 85 (Method Syntax):");
        var highGradeStudents = students
            .Where(s => s.Grade > 85)
            .OrderBy(s => s.Name)
            .Select(s => new { s.Name, s.Grade, s.Course });

        foreach (var student in highGradeStudents)
        {
            Console.WriteLine($"{student.Name}: {student.Grade}
({student.Course})");
        }
```

0

```
        // Query Syntax
        Console.WriteLine("\nComputer Science students (Query Syntax):");
        var csStudents = from s in students
                         where s.Course == "Computer Science"
                         orderby s.Grade descending
                         select s;

        foreach (var student in csStudents)
        {
            Console.WriteLine($"{student.Name}: {student.Grade}");
        }
    }

    public static void AdvancedLinqOperations()
    {
        int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // Aggregation operations
        int sum = numbers.Sum();
        double average = numbers.Average();
        int max = numbers.Max();
        int min = numbers.Min();
        int count = numbers.Count(n => n % 2 == 0);

        Console.WriteLine($"Sum: {sum}, Average: {average:F2}");
        Console.WriteLine($"Max: {max}, Min: {min}, Even count:
{count}");

        // Grouping
        List<Student> students = GetStudents();
        var groupedByCourse = students
            .GroupBy(s => s.Course)
            .Select(g => new
            {
                Course = g.Key,
                Count = g.Count(),
                AverageGrade = g.Average(s => s.Grade)
            });
```

0

```
        Console.WriteLine("\nStudents grouped by course:");
        foreach (var group in groupedByCourse)
        {
            Console.WriteLine($"{group.Course}: {group.Count} students,
Avg grade: {group.AverageGrade:F2}");
        }

        // Joining
        List<Course> courses = new List<Course>
        {
            new Course { Name = "Computer Science", Credits = 4 },
            new Course { Name = "Mathematics", Credits = 3 },
            new Course { Name = "Physics", Credits = 4 }
        };

        var studentCourseInfo = from s in students
                                join c in courses on s.Course equals
c.Name
                                select new
                                {
                                    StudentName = s.Name,
                                    CourseName = c.Name,
                                    Credits = c.Credits,
                                    Grade = s.Grade
                                };

        Console.WriteLine("\nStudent-Course Information:");
        foreach (var info in studentCourseInfo)
        {
            Console.WriteLine($"{info.StudentName} - {info.CourseName}
({info.Credits} credits): {info.Grade}");
        }
    }

    public class Course
    {
        public string Name { get; set; }
        public int Credits { get; set; }
    }
```

0

```
    private static List<Student> GetStudents()
    {
        return new List<Student>
        {
            new Student { Id = 1, Name = "Alice", Age = 20, Course =
"Computer Science", Grade = 85.5 },
            new Student { Id = 2, Name = "Bob", Age = 22, Course =
"Mathematics", Grade = 92.0 },
            new Student { Id = 3, Name = "Charlie", Age = 19, Course =
"Computer Science", Grade = 78.5 },
            new Student { Id = 4, Name = "Diana", Age = 21, Course =
"Physics", Grade = 88.0 }
        };
    }
}
```

Question 3: Write a C# program to connect to the database, insert data into Database and display the data using ADO.NET.

```
using System;
using System.Data;
using System.Data.SqlClient;

public class DatabaseOperations
{
    private static string connectionString =
"Server=localhost;Database=StudentDB;Integrated Security=true;";

    public static void Main()
    {
        try
        {
            CreateDatabase();
            InsertStudentData();
            DisplayStudentData();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
```

0

```
        }
    }

    public static void CreateDatabase()
    {
        string createTableQuery = @"
            IF NOT EXISTS (SELECT * FROM sysobjects WHERE name='Students'
AND xtype='U')
            CREATE TABLE Students (
                Id INT PRIMARY KEY IDENTITY(1,1),
                Name NVARCHAR(50) NOT NULL,
                Age INT NOT NULL,
                Course NVARCHAR(100) NOT NULL,
                Grade DECIMAL(5,2) NOT NULL,
                EnrollmentDate DATETIME DEFAULT GETDATE()
            )";

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();
            using (SqlCommand command = new SqlCommand(createTableQuery,
connection))
            {
                command.ExecuteNonQuery();
                Console.WriteLine("Table 'Students' created
successfully.");
            }
        }
    }

    public static void InsertStudentData()
    {
        string insertQuery = @"
            INSERT INTO Students (Name, Age, Course, Grade)
            VALUES (@Name, @Age, @Course, @Grade)";

        // Sample student data
        var students = new[]
        {
0
```

```
            new { Name = "Alice Johnson", Age = 20, Course = "Computer
Science", Grade = 85.5m },
            new { Name = "Bob Smith", Age = 22, Course = "Mathematics",
Grade = 92.0m },
            new { Name = "Charlie Brown", Age = 19, Course = "Physics",
Grade = 78.5m },
            new { Name = "Diana Prince", Age = 21, Course = "Chemistry",
Grade = 88.0m },
            new { Name = "Eve Wilson", Age = 20, Course = "Biology",
Grade = 95.5m }
        };

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            foreach (var student in students)
            {
                using (SqlCommand command = new SqlCommand(insertQuery,
connection))
                {
                    command.Parameters.AddWithValue("@Name",
student.Name);
                    command.Parameters.AddWithValue("@Age", student.Age);
                    command.Parameters.AddWithValue("@Course",
student.Course);
                    command.Parameters.AddWithValue("@Grade",
student.Grade);

                    int rowsAffected = command.ExecuteNonQuery();
                    Console.WriteLine($"Inserted student: {student.Name}
({rowsAffected} row(s) affected)");
                }
            }
        }
    }

    public static void DisplayStudentData()
    {
0
```

```
        string selectQuery = "SELECT Id, Name, Age, Course, Grade,
EnrollmentDate FROM Students ORDER BY Grade DESC";

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();
            using (SqlCommand command = new SqlCommand(selectQuery,
connection))
            {
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    Console.WriteLine("\n--- Student Information ---");
                    Console.WriteLine($"{"ID",-5} {"Name",-20} {"Age",-5}
{"Course",-20} {"Grade",-8} {"Enrollment Date",-20}");
                    Console.WriteLine(new string('-', 80));

                    while (reader.Read())
                    {
                        Console.WriteLine($"{reader["Id"],-5} " +
                                          $"{reader["Name"],-20} " +
                                          $"{reader["Age"],-5} " +
                                          $"{reader["Course"],-20} " +
                                          $"{reader["Grade"],-8} " +
                                          $"
{((DateTime)reader["EnrollmentDate"]).ToString("yyyy-MM-dd"),-20}");
                    }
                }
            }
        }
    }

    // Alternative method using DataSet (Disconnected architecture)
    public static void DisplayDataUsingDataSet()
    {
        string selectQuery = "SELECT * FROM Students";

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
0
```

```
            SqlDataAdapter adapter = new SqlDataAdapter(selectQuery,
connection);
            DataSet dataSet = new DataSet();

            adapter.Fill(dataSet, "Students");

            Console.WriteLine("\n--- Using DataSet ---");
            foreach (DataRow row in dataSet.Tables["Students"].Rows)
            {
                Console.WriteLine($"ID: {row["Id"]}, Name: {row["Name"]},
" +
                                  $"Course: {row["Course"]}, Grade:
{row["Grade"]}");
            }
        }
    }

    // Method to search students by course
    public static void SearchStudentsByCourse(string courseName)
    {
        string searchQuery = "SELECT * FROM Students WHERE Course =
@Course";

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();
            using (SqlCommand command = new SqlCommand(searchQuery,
connection))
            {
                command.Parameters.AddWithValue("@Course", courseName);

                using (SqlDataReader reader = command.ExecuteReader())
                {
                    Console.WriteLine($"\n--- Students in {courseName} --
-");
                    while (reader.Read())
                    {
                        Console.WriteLine($"{reader["Name"]} - Grade:
{reader["Grade"]}");
0
```

```
                    }
                }
            }
        }
    }
}
```

Question 4: What is the authentication and authorization in ASP.NET? Explain with example.

Authentication vs Authorization:

Aspect	Authentication	Authorization
Definition	Verifying user identity	Determining user permissions
Question	"Who are you?"	"What can you do?"
Process	Login process	Access control
Example	Username/Password	Admin vs User roles

Authentication Types in ASP.NET:

1. **Windows Authentication** - Uses Windows accounts
2. **Forms Authentication** - Custom login forms
3. **Identity Authentication** - ASP.NET Identity system
4. **JWT Authentication** - Token-based authentication

Example Implementation:

```
// Startup.cs (ASP.NET Core)
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add Identity services
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(connectionString));

        services.AddDefaultIdentity<IdentityUser>(options =>
        {
            // Password requirements
```

```
            options.Password.RequireDigit = true;
            options.Password.RequiredLength = 6;
            options.Password.RequireNonAlphanumeric = false;
        })
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

        services.AddControllersWithViews();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        // Authentication middleware
        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
            endpoints.MapRazorPages();
        });
    }
}

// Controllers with Authentication and Authorization
[Authorize] // Requires authentication
public class HomeController : Controller
{
    public IActionResult Index()
    {
0
```

```
        return View();
    }

    [Authorize(Roles = "Admin")] // Requires Admin role
    public IActionResult AdminPanel()
    {
        return View();
    }

    [Authorize(Policy = "MinimumAge")] // Custom policy
    public IActionResult RestrictedContent()
    {
        return View();
    }

    [AllowAnonymous] // Allows anonymous access
    public IActionResult PublicInfo()
    {
        return View();
    }
}

// Account Controller for Authentication
public class AccountController : Controller
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly SignInManager<IdentityUser> _signInManager;

    public AccountController(UserManager<IdentityUser> userManager,
                             SignInManager<IdentityUser> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    [HttpGet]
    public IActionResult Login()
    {
        return View();
    }
```

0

```
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(
            model.Email, model.Password, model.RememberMe, false);

        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }

        ModelState.AddModelError("", "Invalid login attempt.");
    }

    return View(model);
}

[HttpPost]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}

[HttpGet]
public IActionResult Register()
{
    return View();
}

[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
```

```
                UserName = model.Email,
                Email = model.Email
            };

            var result = await _userManager.CreateAsync(user,
model.Password);

            if (result.Succeeded)
            {
                await _signInManager.SignInAsync(user, isPersistent:
false);
                return RedirectToAction("Index", "Home");
            }

            foreach (var error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
        }

        return View(model);
    }
}

// ViewModels
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    public bool RememberMe { get; set; }
}

public class RegisterViewModel
{
```

```
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "Passwords don't match.")]
    public string ConfirmPassword { get; set; }
}
```

Question 5: Explain the ASP.NET page life cycle. What is the difference between Server.Redirect and Server.Transfer? Explain with example.

ASP.NET Page Life Cycle:

```
public partial class PageLifeCycleDemo : System.Web.UI.Page
{
    protected void Page_PreInit(object sender, EventArgs e)
    {
        // Occurs before page initialization
        // Can set themes, master pages, and create dynamic controls
        Response.Write("1. PreInit<br/>");
    }

    protected void Page_Init(object sender, EventArgs e)
    {
        // Page and control initialization
        // ViewState is not available yet
        Response.Write("2. Init<br/>");
    }

    protected void Page_InitComplete(object sender, EventArgs e)
    {
        // All controls are initialized
        Response.Write("3. InitComplete<br/>");
    }
```

0

```
protected void Page_PreLoad(object sender, EventArgs e)
{
    // Occurs before Load event
    Response.Write("4. PreLoad<br/>");
}

protected void Page_Load(object sender, EventArgs e)
{
    // Page and controls are loaded
    // ViewState is available
    if (!IsPostBack)
    {
        Response.Write("5. Load (First Time)<br/>");
    }
    else
    {
        Response.Write("5. Load (PostBack)<br/>");
    }
}

protected void Page_LoadComplete(object sender, EventArgs e)
{
    // All controls are loaded
    Response.Write("6. LoadComplete<br/>");
}

protected void Page_PreRender(object sender, EventArgs e)
{
    // Occurs before rendering
    // Last chance to modify controls
    Response.Write("7. PreRender<br/>");
}

protected void Page_PreRenderComplete(object sender, EventArgs e)
{
    // All controls have completed pre-rendering
    Response.Write("8. PreRenderComplete<br/>");
}
```

0

```
    protected void Page_SaveStateComplete(object sender, EventArgs e)
    {
        // ViewState has been saved
        Response.Write("9. SaveStateComplete<br/>");
    }

    protected void Page_Unload(object sender, EventArgs e)
    {
        // Cleanup - page is being destroyed
        // Cannot modify response here
        // Use for cleanup operations
    }
}
```

Server.Redirect vs Server.Transfer:

Aspect	Server.Redirect	Server.Transfer
Location	Client-side redirection	Server-side transfer
URL Change	Browser URL changes	Browser URL remains same
Round Trips	Two round trips	One round trip
Performance	Slower	Faster
Context Preservation	New request context	Same request context
External URLs	Can redirect to external sites	Cannot transfer to external sites

Examples:

```
// Server.Redirect Example
public partial class RedirectExample : System.Web.UI.Page
{
    protected void btnRedirect_Click(object sender, EventArgs e)
    {
        // Method 1: Simple redirect
        Response.Redirect("Welcome.aspx");

        // Method 2: Redirect with query string
        Response.Redirect("Welcome.aspx?name=" + txtName.Text);

        // Method 3: Redirect with end response
```

0

```
        Response.Redirect("Welcome.aspx", true); // Ends current page
execution

        // Method 4: Redirect to external site
        Response.Redirect("https://www.google.com");
    }

    protected void btnRedirectPermanent_Click(object sender, EventArgs e)
    {
        // Permanent redirect (301 status code)
        Response.RedirectPermanent("NewLocation.aspx");
    }
}

// Server.Transfer Example
public partial class TransferExample : System.Web.UI.Page
{
    protected void btnTransfer_Click(object sender, EventArgs e)
    {
        // Method 1: Simple transfer
        Server.Transfer("Welcome.aspx");

        // Method 2: Transfer with preserving form data
        Server.Transfer("Welcome.aspx", true); // Preserves form
collection

        // Method 3: Transfer with passing data through Context
        Context.Items["UserData"] = txtName.Text;
        Server.Transfer("Welcome.aspx");
    }
}

// Destination page to receive transferred data
public partial class Welcome : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        // Getting data from redirect (query string)
        string nameFromRedirect = Request.QueryString["name"];
        if (!string.IsNullOrEmpty(nameFromRedirect))
```

0

```
        {
            lblWelcome.Text = "Welcome " + nameFromRedirect + " (from
Redirect)";
        }

        // Getting data from transfer (Context.Items)
        string nameFromTransfer = Context.Items["UserData"] as string;
        if (!string.IsNullOrEmpty(nameFromTransfer))
        {
            lblWelcome.Text = "Welcome " + nameFromTransfer + " (from
Transfer)";
        }

        // Getting data from transfer (Previous page)
        if (PreviousPage != null)
        {
            TextBox txtPreviousName = PreviousPage.FindControl("txtName")
as TextBox;
            if (txtPreviousName != null)
            {
                lblWelcome.Text = "Welcome " + txtPreviousName.Text + "
(from Previous Page)";
            }
        }
    }
}
```

Question 6: Describe the following terms: Razor View, Stream Reader and Stream Writer.

Razor View

Razor is a markup syntax for embedding .NET code into web pages. It provides a clean and lightweight way to create dynamic web content.

Characteristics:

- Uses @ symbol to transition from HTML to C#
- Supports both C# and VB.NET
- IntelliSense support in Visual Studio

0

- Compile-time syntax checking

Examples:

```
@{
    ViewData["Title"] = "Student List";
    var students = ViewBag.Students as List<Student>;
}

<h2>@ViewData["Title"]</h2>

@if (students != null && students.Any())
{
    <table class="table">
        <thead>
            <tr>
                <th>Name</th>
                <th>Age</th>
                <th>Course</th>
                <th>Grade</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var student in students)
            {
                <tr>
                    <td>@student.Name</td>
                    <td>@student.Age</td>
                    <td>@student.Course</td>
                    <td>@student.Grade.ToString("F2")</td>
                    <td>
                        <a href="@Url.Action("Edit", "Student", new { id
= student.Id })">Edit</a> |
                        <a href="@Url.Action("Delete", "Student", new {
id = student.Id })">Delete</a>
                    </td>
                </tr>
            }
        </tbody>
```

0

```
    </table>
}
else
{
    <p>No students found.</p>
}

@section Scripts {
    <script>
        function confirmDelete(studentName) {
            return confirm('Are you sure you want to delete ' +
studentName + '?');
        }
    </script>
}
```

Stream Reader

StreamReader is used to read characters from a stream in a particular encoding.

Characteristics:

- Reads text files efficiently
- Supports different encodings (UTF-8, ASCII, etc.)
- Implements IDisposable (use with using statement)
- Provides both synchronous and asynchronous methods

Examples:

```
public class StreamReaderExamples
{
    public static void ReadTextFile()
    {
        string filePath = @"C:\temp\students.txt";

        // Method 1: Using 'using' statement (recommended)
        using (StreamReader reader = new StreamReader(filePath))
        {
            string content = reader.ReadToEnd();
            Console.WriteLine("File Content:");
            Console.WriteLine(content);
```

0

```
        }

        // Method 2: Reading line by line
        using (StreamReader reader = new StreamReader(filePath))
        {
            string line;
            int lineNumber = 1;

            while ((line = reader.ReadLine()) != null)
            {
                Console.WriteLine($"Line {lineNumber}: {line}");
                lineNumber++;
            }
        }

        // Method 3: Reading with specific encoding
        using (StreamReader reader = new StreamReader(filePath,
Encoding.UTF8))
        {
            string content = reader.ReadToEnd();
            Console.WriteLine(content);
        }
    }

    public static async Task ReadTextFileAsync()
    {
        string filePath = @"C:\temp\students.txt";

        using (StreamReader reader = new StreamReader(filePath))
        {
            string content = await reader.ReadToEndAsync();
            Console.WriteLine("Async File Content:");
            Console.WriteLine(content);
        }
    }

    public static void ProcessLargeFile()
    {
        string filePath = @"C:\temp\largefile.txt";
```

0

```
        using (StreamReader reader = new StreamReader(filePath))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                // Process each line individually to save memory
                ProcessLine(line);
            }
        }
    }

    private static void ProcessLine(string line)
    {
        // Process individual line
        if (line.Contains("ERROR"))
        {
            Console.WriteLine($"Error found: {line}");
        }
    }
}
```

Stream Writer

StreamWriter is used to write characters to a stream in a particular encoding.

Characteristics:

- Writes text to files efficiently
- Supports different encodings
- Can append to existing files
- Implements IDisposable
- Provides both synchronous and asynchronous methods

Examples:

```
public class StreamWriterExamples
{
    public static void WriteTextFile()
    {
        string filePath = @"C:\temp\output.txt";
```

0

```
        // Method 1: Create new file (overwrites existing)
        using (StreamWriter writer = new StreamWriter(filePath))
        {
            writer.WriteLine("Student Information");
            writer.WriteLine("==================");
            writer.WriteLine("Name: Alice Johnson");
            writer.WriteLine("Age: 20");
            writer.WriteLine("Course: Computer Science");
            writer.WriteLine($"Date: {DateTime.Now:yyyy-MM-dd
HH:mm:ss}");
        }

        // Method 2: Append to existing file
        using (StreamWriter writer = new StreamWriter(filePath, append:
true))
        {
            writer.WriteLine("\nAdditional Information:");
            writer.WriteLine("Grade: 85.5");
            writer.WriteLine("Status: Active");
        }

        // Method 3: With specific encoding
        using (StreamWriter writer = new StreamWriter(filePath, false,
Encoding.UTF8))
        {
            writer.WriteLine("UTF-8 encoded content with special
characters: Ã¡Ã©ÃÃ³Ãº");
        }
    }

    public static void WriteStudentData()
    {
        string filePath = @"C:\temp\students.csv";

        var students = new[]
        {
            new { Name = "Alice", Age = 20, Course = "CS", Grade = 85.5
},
            new { Name = "Bob", Age = 22, Course = "Math", Grade = 92.0
},
```

```
            new { Name = "Charlie", Age = 19, Course = "Physics", Grade =
78.5 }
        };

        using (StreamWriter writer = new StreamWriter(filePath))
        {
            // Write CSV header
            writer.WriteLine("Name,Age,Course,Grade");

            // Write student data
            foreach (var student in students)
            {
                writer.WriteLine($"{student.Name},{student.Age},
{student.Course},{student.Grade}");
            }
        }

        Console.WriteLine($"Student data written to {filePath}");
    }

    public static async Task WriteTextFileAsync()
    {
        string filePath = @"C:\temp\async_output.txt";

        using (StreamWriter writer = new StreamWriter(filePath))
        {
            await writer.WriteLineAsync("Async writing example");
            await writer.WriteLineAsync($"Written at: {DateTime.Now}");
        }
    }

    public static void LogExample()
    {
        string logPath = @"C:\temp\application.log";

        using (StreamWriter writer = new StreamWriter(logPath, append:
true))
        {
            writer.WriteLine($"[{DateTime.Now:yyyy-MM-dd HH:mm:ss}] INFO:
Application started");
0
```

```
            writer.WriteLine($"[{DateTime.Now:yyyy-MM-dd HH:mm:ss}]
DEBUG: Processing user request");
            writer.WriteLine($"[{DateTime.Now:yyyy-MM-dd HH:mm:ss}]
ERROR: Database connection failed");

            // Ensure data is written immediately
            writer.Flush();
        }
    }
}
```

Question 7: Describe different types of inheritance methods with example in C#.

Types of Inheritance in C#:

1. **Single Inheritance** - One class inherits from one base class
2. **Multilevel Inheritance** - Chain of inheritance
3. **Hierarchical Inheritance** - Multiple classes inherit from one base class
4. **Interface Inheritance** - Multiple interface implementation (Multiple inheritance alternative)

Note: C# does not support multiple inheritance of classes, but supports multiple interface inheritance.

Examples:

```
// 1. Single Inheritance
public class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }

    public virtual void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }

    public virtual void Sleep()
0   {
```

```
        Console.WriteLine($"{Name} is sleeping.");
    }
}

public class Dog : Animal // Single inheritance
{
    public string Breed { get; set; }

    public override void Eat()
    {
        Console.WriteLine($"{Name} the dog is eating dog food.");
    }

    public void Bark()
    {
        Console.WriteLine($"{Name} is barking!");
    }
}

// 2. Multilevel Inheritance
public class Mammal : Animal
{
    public double BodyTemperature { get; set; } = 37.0;

    public virtual void GiveBirth()
    {
        Console.WriteLine($"{Name} is giving birth to live offspring.");
    }
}

public class Canine : Mammal
{
    public string PackBehavior { get; set; }

    public virtual void Hunt()
    {
        Console.WriteLine($"{Name} is hunting in a pack.");
    }
}
```

0

```
public class Wolf : Canine // Multilevel: Wolf -> Canine -> Mammal ->
Animal
{
    public string Territory { get; set; }

    public override void Hunt()
    {
        Console.WriteLine($"{Name} the wolf is hunting in territory:
{Territory}");
    }

    public void Howl()
    {
        Console.WriteLine($"{Name} is howling at the moon!");
    }
}

// 3. Hierarchical Inheritance
public class Vehicle
{
    public string Brand { get; set; }
    public int Year { get; set; }

    public virtual void Start()
    {
        Console.WriteLine($"{Brand} vehicle is starting.");
    }

    public virtual void Stop()
    {
        Console.WriteLine($"{Brand} vehicle is stopping.");
    }
}

public class Car : Vehicle // Hierarchical inheritance
{
    public int NumberOfDoors { get; set; }

    public override void Start()
    {
0
```

```
        Console.WriteLine($"{Brand} car is starting with ignition.");
    }

    public void OpenTrunk()
    {
        Console.WriteLine("Car trunk is opened.");
    }
}

public class Motorcycle : Vehicle // Hierarchical inheritance
{
    public bool HasSidecar { get; set; }

    public override void Start()
    {
        Console.WriteLine($"{Brand} motorcycle is starting with
kick/button.");
    }

    public void Wheelie()
    {
        Console.WriteLine("Motorcycle is doing a wheelie!");
    }
}

public class Truck : Vehicle // Hierarchical inheritance
{
    public double CargoCapacity { get; set; }

    public override void Start()
    {
        Console.WriteLine($"{Brand} truck is starting with heavy
engine.");
    }

    public void LoadCargo()
    {
        Console.WriteLine($"Loading cargo up to {CargoCapacity} tons.");
    }
}
```

0

```
// 4. Interface Inheritance (Multiple inheritance alternative)
public interface IFlyable
{
    void Fly();
    double MaxAltitude { get; }
}

public interface ISwimmable
{
    void Swim();
    double MaxDepth { get; }
}

public interface IWalkable
{
    void Walk();
    double MaxSpeed { get; }
}

// Multiple interface inheritance
public class Duck : Animal, IFlyable, ISwimmable, IWalkable
{
    public double MaxAltitude { get; set; } = 1000;
    public double MaxDepth { get; set; } = 5;
    public double MaxSpeed { get; set; } = 10;

    public void Fly()
    {
        Console.WriteLine($"{Name} the duck is flying up to {MaxAltitude}
meters.");
    }

    public void Swim()
    {
        Console.WriteLine($"{Name} the duck is swimming up to {MaxDepth}
meters deep.");
    }

    public void Walk()
```

```
    {
        Console.WriteLine($"{Name} the duck is walking at {MaxSpeed}
km/h.");
    }

    public void Quack()
    {
        Console.WriteLine($"{Name} says: Quack quack!");
    }
}

// Usage Example
public class InheritanceDemo
{
    public static void Main()
    {
        // Single inheritance
        Dog dog = new Dog { Name = "Buddy", Age = 3, Breed = "Golden
Retriever" };
        dog.Eat();
        dog.Bark();

        // Multilevel inheritance
        Wolf wolf = new Wolf { Name = "Alpha", Age = 5, Territory =
"Forest" };
        wolf.Eat();      // From Animal
        wolf.GiveBirth(); // From Mammal
        wolf.Hunt();     // From Canine (overridden)
        wolf.Howl();     // From Wolf

        // Hierarchical inheritance
        Car car = new Car { Brand = "Toyota", Year = 2023, NumberOfDoors
= 4 };
        Motorcycle bike = new Motorcycle { Brand = "Honda", Year = 2022,
HasSidecar = false };
        Truck truck = new Truck { Brand = "Ford", Year = 2021,
CargoCapacity = 10.5 };

        car.Start();
        bike.Start();
```

```
        truck.Start();

        // Multiple interface inheritance
        Duck duck = new Duck { Name = "Donald", Age = 2 };
        duck.Eat();   // From Animal
        duck.Fly();   // From IFlyable
        duck.Swim();  // From ISwimmable
        duck.Walk();  // From IWalkable
        duck.Quack(); // From Duck
    }
}
```

Question 8: What are benefit of Ajax? How do you implement Ajax in MVC? Explain.

Benefits of AJAX:

1. **Improved User Experience** - No full page reloads
2. **Faster Response Time** - Only necessary data is transferred
3. **Reduced Server Load** - Less bandwidth usage
4. **Asynchronous Operations** - Non-blocking user interface
5. **Dynamic Content Updates** - Real-time data updates
6. **Better Interactivity** - More responsive applications

AJAX Implementation in MVC:

```
// 1. Controller Actions for AJAX
public class EmployeeController : Controller
{
    private List<Employee> employees = new List<Employee>
    {
        new Employee { Id = 1, Name = "Alice Johnson", Department = "IT",
Salary = 75000 },
        new Employee { Id = 2, Name = "Bob Smith", Department = "HR",
Salary = 65000 },
        new Employee { Id = 3, Name = "Charlie Brown", Department =
"Finance", Salary = 80000 }
    };

0
```

```
    public IActionResult Index()
    {
        return View();
    }

    // AJAX method to get all employees
    [HttpGet]
    public JsonResult GetEmployees()
    {
        return Json(employees);
    }

    // AJAX method to get employee by ID
    [HttpGet]
    public JsonResult GetEmployee(int id)
    {
        var employee = employees.FirstOrDefault(e => e.Id == id);
        if (employee != null)
        {
            return Json(new { success = true, data = employee });
        }
        return Json(new { success = false, message = "Employee not found"
});
    }

    // AJAX method to add employee
    [HttpPost]
    public JsonResult AddEmployee(Employee employee)
    {
        try
        {
            employee.Id = employees.Max(e => e.Id) + 1;
            employees.Add(employee);
            return Json(new { success = true, message = "Employee added
successfully", data = employee });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, message = ex.Message });
        }
```

```
    }

    // AJAX method to update employee
    [HttpPost]
    public JsonResult UpdateEmployee(Employee employee)
    {
        try
        {
            var existingEmployee = employees.FirstOrDefault(e => e.Id ==
employee.Id);
            if (existingEmployee != null)
            {
                existingEmployee.Name = employee.Name;
                existingEmployee.Department = employee.Department;
                existingEmployee.Salary = employee.Salary;
                return Json(new { success = true, message = "Employee
updated successfully" });
            }
            return Json(new { success = false, message = "Employee not
found" });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, message = ex.Message });
        }
    }

    // AJAX method to delete employee
    [HttpPost]
    public JsonResult DeleteEmployee(int id)
    {
        try
        {
            var employee = employees.FirstOrDefault(e => e.Id == id);
            if (employee != null)
            {
                employees.Remove(employee);
                return Json(new { success = true, message = "Employee
deleted successfully" });
            }
```

```
            return Json(new { success = false, message = "Employee not
found" });
        }
        catch (Exception ex)
        {
            return Json(new { success = false, message = ex.Message });
        }
    }
}

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
    public decimal Salary { get; set; }
}
```

Razor View with AJAX:

```
@{
    ViewData["Title"] = "Employee Management";
}

<h2>Employee Management System</h2>

<div class="container">
    <!-- Add Employee Form -->
    <div class="card mb-4">
        <div class="card-header">
            <h4>Add New Employee</h4>
        </div>
        <div class="card-body">
            <form id="employeeForm">
                <div class="row">
                    <div class="col-md-4">
                        <input type="text" id="employeeName" class="form-
control" placeholder="Employee Name" required>
                    </div>
```

0

```
                    <div class="col-md-4">
                        <input type="text" id="employeeDepartment"
class="form-control" placeholder="Department" required>
                    </div>
                    <div class="col-md-2">
                        <input type="number" id="employeeSalary"
class="form-control" placeholder="Salary" required>
                    </div>
                    <div class="col-md-2">
                        <button type="button" id="addEmployee" class="btn
btn-primary">Add Employee</button>
                    </div>
                </div>
            </form>
        </div>
    </div>

    <!-- Employee List -->
    <div class="card">
        <div class="card-header d-flex justify-content-between">
            <h4>Employee List</h4>
            <button type="button" id="refreshList" class="btn btn-
secondary">Refresh</button>
        </div>
        <div class="card-body">
            <div id="loadingSpinner" class="text-center" style="display:
none;">
                <div class="spinner-border" role="status">
                    <span class="sr-only">Loading...</span>
                </div>
            </div>
            <table class="table table-striped" id="employeeTable">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Name</th>
                        <th>Department</th>
                        <th>Salary</th>
                        <th>Actions</th>
                    </tr>
```

```
                </thead>
                <tbody id="employeeTableBody">
                    <!-- Employee data will be loaded here via AJAX -->
                </tbody>
            </table>
        </div>
    </div>
</div>

<!-- Edit Employee Modal -->
<div class="modal fade" id="editEmployeeModal" tabindex="-1">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">Edit Employee</h5>
                <button type="button" class="close" data-dismiss="modal">
                    <span>&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <form id="editEmployeeForm">
                    <input type="hidden" id="editEmployeeId">
                    <div class="form-group">
                        <label>Name:</label>
                        <input type="text" id="editEmployeeName"
class="form-control" required>
                    </div>
                    <div class="form-group">
                        <label>Department:</label>
                        <input type="text" id="editEmployeeDepartment"
class="form-control" required>
                    </div>
                    <div class="form-group">
                        <label>Salary:</label>
                        <input type="number" id="editEmployeeSalary"
class="form-control" required>
                    </div>
                </form>
            </div>
            <div class="modal-footer">
```

0

```
                <button type="button" class="btn btn-secondary" data-
dismiss="modal">Cancel</button>
                <button type="button" id="saveEmployee" class="btn btn-
primary">Save Changes</button>
            </div>
        </div>
    </div>
</div>

@section Scripts {
<script>
$(document).ready(function() {
    // Load employees on page load
    loadEmployees();

    // Add employee
    $('#addEmployee').click(function() {
        addEmployee();
    });

    // Refresh employee list
    $('#refreshList').click(function() {
        loadEmployees();
    });

    // Save edited employee
    $('#saveEmployee').click(function() {
        updateEmployee();
    });
});

function loadEmployees() {
    $('#loadingSpinner').show();
    $('#employeeTableBody').empty();

    $.ajax({
        url: '@Url.Action("GetEmployees", "Employee")',
        type: 'GET',
        dataType: 'json',
        success: function(data) {
```

```
            $('#loadingSpinner').hide();
            var tbody = $('#employeeTableBody');

            if (data && data.length > 0) {
                $.each(data, function(index, employee) {
                    var row = `
                        <tr>
                            <td>${employee.id}</td>
                            <td>${employee.name}</td>
                            <td>${employee.department}</td>
                            <td>$${employee.salary.toLocaleString()}</td>
                            <td>
                                <button class="btn btn-sm btn-warning"
onclick="editEmployee(${employee.id})">Edit</button>
                                <button class="btn btn-sm btn-danger"
onclick="deleteEmployee(${employee.id})">Delete</button>
                            </td>
                        </tr>
                    `;
                    tbody.append(row);
                });
            } else {
                tbody.append('<tr><td colspan="5" class="text-center">No
employees found</td></tr>');
            }
        },
        error: function() {
            $('#loadingSpinner').hide();
            alert('Failed to load employees');
        }
    });
}

function addEmployee() {
    var name = $('#employeeName').val();
    var department = $('#employeeDepartment').val();
    var salary = $('#employeeSalary').val();

    if (!name || !department || !salary) {
        alert('Please fill all fields');
```

0

```
        return;
    }

    $.ajax({
        url: '@Url.Action("AddEmployee", "Employee")',
        type: 'POST',
        data: {
            Name: name,
            Department: department,
            Salary: salary
        },
        success: function(response) {
            if (response.success) {
                alert(response.message);
                loadEmployees();
                $('#employeeName').val('');
                $('#employeeDepartment').val('');
                $('#employeeSalary').val('');
            } else {
                alert('Error: ' + response.message);
            }
        },
        error: function() {
            alert('Failed to add employee');
        }
    });
}

function editEmployee(id) {
    $.ajax({
        url: '@Url.Action("GetEmployee", "Employee")',
        type: 'GET',
        data: { id: id },
        success: function(response) {
            if (response.success) {
                var employee = response.data;
                $('#editEmployeeId').val(employee.id);
                $('#editEmployeeName').val(employee.name);
                $('#editEmployeeDepartment').val(employee.department);
                $('#editEmployeeSalary').val(employee.salary);
```

0

```
                $('#editEmployeeModal').modal('show');
            } else {
                alert('Error: ' + response.message);
            }
        },
        error: function() {
            alert('Failed to get employee details');
        }
    });
}

function updateEmployee() {
    var id = $('#editEmployeeId').val();
    var name = $('#editEmployeeName').val();
    var department = $('#editEmployeeDepartment').val();
    var salary = $('#editEmployeeSalary').val();

    $.ajax({
        url: '@Url.Action("UpdateEmployee", "Employee")',
        type: 'POST',
        data: {
            Id: id,
            Name: name,
            Department: department,
            Salary: salary
        },
        success: function(response) {
            if (response.success) {
                alert(response.message);
                $('#editEmployeeModal').modal('hide');
                loadEmployees();
            } else {
                alert('Error: ' + response.message);
            }
        },
        error: function() {
            alert('Failed to update employee');
        }
    });
}
```

0

```
function deleteEmployee(id) {
    if (confirm('Are you sure you want to delete this employee?')) {
        $.ajax({
            url: '@Url.Action("DeleteEmployee", "Employee")',
            type: 'POST',
            data: { id: id },
            success: function(response) {
                if (response.success) {
                    alert(response.message);
                    loadEmployees();
                } else {
                    alert('Error: ' + response.message);
                }
            },
            error: function() {
                alert('Failed to delete employee');
            }
        });
    }
}
</script>
}
```

Question 9: Explain difference between value type and reference type in CLR with example.

Value Types vs Reference Types in CLR:

Aspect	Value Type	Reference Type
Storage	Stack (for local variables)	Heap
Memory Allocation	Direct value storage	Reference to heap location
Assignment	Creates a copy	Copies the reference
Null Values	Cannot be null (except nullable)	Can be null
Performance	Faster access	Slower due to indirection
Garbage Collection	No GC needed for stack allocation	Subject to garbage collection
Default Values	Zero/false for primitives	null for references

0

CLR Memory Management Example:

```
public class ValueVsReferenceTypeDemo
{
    // Value type struct
    public struct Point
    {
        public int X, Y;
        public Point(int x, int y) { X = x; Y = y; }
        public override string ToString() => $"({X}, {Y})";
    }

    // Reference type class
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public Person(string name, int age) { Name = name; Age = age; }
        public override string ToString() => $"{Name}, {Age}";
    }

    public static void Main()
    {
        Console.WriteLine("=== Value Type Behavior ===");
        ValueTypeExample();

        Console.WriteLine("\n=== Reference Type Behavior ===");
        ReferenceTypeExample();

        Console.WriteLine("\n=== Boxing and Unboxing ===");
        BoxingUnboxingExample();
    }

    public static void ValueTypeExample()
    {
        // Value types - stored on stack
        int a = 10;
        int b = a;    // Copy of value
        a = 20;       // Only 'a' changes
```

0

```
        Console.WriteLine($"a = {a}, b = {b}"); // a = 20, b = 10

        // Struct example
        Point p1 = new Point(5, 10);
        Point p2 = p1;    // Copy created
        p1.X = 15;        // Only p1 changes

        Console.WriteLine($"p1 = {p1}, p2 = {p2}"); // p1 = (15, 10), p2
= (5, 10)
    }

    public static void ReferenceTypeExample()
    {
        // Reference types - stored on heap
        Person person1 = new Person("Alice", 25);
        Person person2 = person1;    // Copy of reference, not object
        person1.Age = 30;            // Both references point to same
object

        Console.WriteLine($"person1: {person1}"); // Alice, 30
        Console.WriteLine($"person2: {person2}"); // Alice, 30 (same
object)
    }

    public static void BoxingUnboxingExample()
    {
        // Boxing: Value type to object (heap allocation)
        int valueType = 42;
        object boxed = valueType;    // Boxing occurs

        Console.WriteLine($"Original: {valueType}");
        Console.WriteLine($"Boxed: {boxed}");

        // Unboxing: Object back to value type
        int unboxed = (int)boxed;    // Unboxing occurs
        Console.WriteLine($"Unboxed: {unboxed}");
    }
}
```

0

Question 10: What are two types of delegate? Explain each with an example.

Types of Delegates in C#:

1. **Single-cast Delegate (Simple Delegate)**
2. **Multi-cast Delegate**

1. Single-cast Delegate:

A delegate that holds reference to a single method.

```
public class SingleCastDelegateDemo
{
    // Declare delegate type
    public delegate int CalculateDelegate(int x, int y);
    public delegate void PrintDelegate(string message);

    public static void Main()
    {
        Console.WriteLine("=== Single-cast Delegate Examples ===");

        // Create delegate instance pointing to a method
        CalculateDelegate calc = Add;

        // Call delegate
        int result = calc(10, 5);
        Console.WriteLine($"Addition result: {result}");

        // Change delegate to point to different method
        calc = Multiply;
        result = calc(10, 5);
        Console.WriteLine($"Multiplication result: {result}");

        // Using lambda expression
        calc = (x, y) => x / y;
        result = calc(10, 5);
        Console.WriteLine($"Division result: {result}");
    }
```

0

```
    public static int Add(int x, int y)
    {
        Console.WriteLine($"Adding {x} + {y}");
        return x + y;
    }

    public static int Multiply(int x, int y)
    {
        Console.WriteLine($"Multiplying {x} * {y}");
        return x * y;
    }
}
```

2. Multi-cast Delegate:

A delegate that can hold references to multiple methods and call them in sequence.

```
public class MultiCastDelegateDemo
{
    public delegate void NotificationDelegate(string message);

    public static void Main()
    {
        Console.WriteLine("=== Multi-cast Delegate Examples ===");

        // Create multi-cast delegate
        NotificationDelegate notification = EmailNotification;
        notification += SmsNotification;        // Add another method
        notification += PushNotification;      // Add third method

        // Call all methods in the delegate
        Console.WriteLine("Sending notification...");
        notification("System maintenance scheduled");

        Console.WriteLine("\n--- Removing Methods ---");
        notification -= EmailNotification;    // Remove method
        notification("Emergency alert");
    }

0   public static void EmailNotification(string message)
```

```
    {
        Console.WriteLine($"EMAIL: {message}");
    }

    public static void SmsNotification(string message)
    {
        Console.WriteLine($"SMS: {message}");
    }

    public static void PushNotification(string message)
    {
        Console.WriteLine($"PUSH: {message}");
    }
}
```

Question 11: Define term serialization and deserialization and write a program to write user input into file and display using stream class.

Definitions:

- **Serialization**: Converting an object's state into a format that can be stored or transmitted
- **Deserialization**: Converting serialized data back into an object

Complete Program Example:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.Json;

[Serializable]
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Course { get; set; }
    public double Grade { get; set; }

0
```

```
    public Student() { }

    public Student(int id, string name, int age, string course, double
grade)
    {
        Id = id;
        Name = name;
        Age = age;
        Course = course;
        Grade = grade;
    }

    public override string ToString()
    {
        return $"ID: {Id}, Name: {Name}, Age: {Age}, Course: {Course},
Grade: {Grade:F2}";
    }
}

public class SerializationDemo
{
    private static List<Student> students = new List<Student>();
    private static string jsonFilePath = @"C:\temp\students.json";
    private static string textFilePath = @"C:\temp\students.txt";

    public static void Main()
    {
        Directory.CreateDirectory(@"C:\temp");

        bool exit = false;
        while (!exit)
        {
            Console.WriteLine("\n=== Student Management System ===");
            Console.WriteLine("1. Add Student");
            Console.WriteLine("2. Display All Students");
            Console.WriteLine("3. Save to JSON File");
            Console.WriteLine("4. Load from JSON File");
            Console.WriteLine("5. Save to Text File");
            Console.WriteLine("6. Load from Text File");
            Console.WriteLine("7. Exit");
```

0

```
            Console.Write("Choose option: ");

            string choice = Console.ReadLine();

            switch (choice)
            {
                case "1": AddStudent(); break;
                case "2": DisplayStudents(); break;
                case "3": SaveToJson(); break;
                case "4": LoadFromJson(); break;
                case "5": SaveToTextFile(); break;
                case "6": LoadFromTextFile(); break;
                case "7": exit = true; break;
                default: Console.WriteLine("Invalid option!"); break;
            }
        }
    }

    public static void AddStudent()
    {
        try
        {
            Console.Write("Enter Student ID: ");
            int id = int.Parse(Console.ReadLine());

            Console.Write("Enter Student Name: ");
            string name = Console.ReadLine();

            Console.Write("Enter Student Age: ");
            int age = int.Parse(Console.ReadLine());

            Console.Write("Enter Course: ");
            string course = Console.ReadLine();

            Console.Write("Enter Grade (0-100): ");
            double grade = double.Parse(Console.ReadLine());

            Student student = new Student(id, name, age, course, grade);
            students.Add(student);
```

0

```
            Console.WriteLine("Student added successfully!");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error adding student: {ex.Message}");
        }
    }

    public static void DisplayStudents()
    {
        if (students.Count == 0)
        {
            Console.WriteLine("No students found.");
            return;
        }

        Console.WriteLine("\n--- Student List ---");
        foreach (Student student in students)
        {
            Console.WriteLine(student);
        }
    }

    // JSON Serialization using System.Text.Json
    public static void SaveToJson()
    {
        try
        {
            var options = new JsonSerializerOptions { WriteIndented =
true };
            string jsonString = JsonSerializer.Serialize(students,
options);

            using (StreamWriter writer = new StreamWriter(jsonFilePath))
            {
                writer.Write(jsonString);
            }

            Console.WriteLine($"Data saved to JSON file:
{jsonFilePath}");
```

0

```
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error saving to JSON: {ex.Message}");
        }
    }

    public static void LoadFromJson()
    {
        try
        {
            if (!File.Exists(jsonFilePath))
            {
                Console.WriteLine("JSON file not found.");
                return;
            }

            string jsonString;
            using (StreamReader reader = new StreamReader(jsonFilePath))
            {
                jsonString = reader.ReadToEnd();
            }

            students = JsonSerializer.Deserialize<List<Student>>
(jsonString) ?? new List<Student>();

            Console.WriteLine($"Data loaded from JSON file.
{students.Count} students loaded.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error loading from JSON: {ex.Message}");
        }
    }

    // Custom Text File Format using StreamWriter/StreamReader
    public static void SaveToTextFile()
    {
        try
        {
0
```

```
            using (StreamWriter writer = new StreamWriter(textFilePath))
            {
                writer.WriteLine("# Student Data File");
                writer.WriteLine($"# Generated on: {DateTime.Now}");
                writer.WriteLine($"# Total Students: {students.Count}");
                writer.WriteLine("# Format: ID|Name|Age|Course|Grade");
                writer.WriteLine("---DATA---");

                foreach (Student student in students)
                {
                    writer.WriteLine($"{student.Id}|{student.Name}|
{student.Age}|{student.Course}|{student.Grade}");
                }

                writer.WriteLine("---END---");
            }

            Console.WriteLine($"Data saved to text file:
{textFilePath}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error saving to text file:
{ex.Message}");
        }
    }

    public static void LoadFromTextFile()
    {
        try
        {
            if (!File.Exists(textFilePath))
            {
                Console.WriteLine("Text file not found.");
                return;
            }

            students.Clear();
            bool dataSection = false;
```

0

```
            using (StreamReader reader = new StreamReader(textFilePath))
            {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    if (line == "---DATA---")
                    {
                        dataSection = true;
                        continue;
                    }

                    if (line == "---END---")
                    {
                        break;
                    }

                    if (dataSection && !line.StartsWith("#"))
                    {
                        string[] parts = line.Split('|');
                        if (parts.Length == 5)
                        {
                            Student student = new Student
                            {
                                Id = int.Parse(parts[0]),
                                Name = parts[1],
                                Age = int.Parse(parts[2]),
                                Course = parts[3],
                                Grade = double.Parse(parts[4])
                            };
                            students.Add(student);
                        }
                    }
                }
            }

            Console.WriteLine($"Data loaded from text file.
{students.Count} students loaded.");
        }
        catch (Exception ex)
        {
0
```

```
            Console.WriteLine($"Error loading from text file:
{ex.Message}");
        }
    }
}
```

Question 12: What is TPL? And write a program to create multiple threads along with main thread.

TPL (Task Parallel Library):

TPL is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces that simplifies the process of adding parallelism and concurrency to applications.

Benefits of TPL:

- Simplified thread management
- Automatic work distribution
- Better performance on multi-core systems
- Built-in cancellation support
- Exception handling across threads

Complete Program Example:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class TPLDemo
{
    public static void Main()
    {
        Console.WriteLine($"Main thread ID:
{Thread.CurrentThread.ManagedThreadId}");
        Console.WriteLine("=== TPL (Task Parallel Library) Demo ===\n");

        // 1. Basic Task creation
        BasicTaskExample();

        // 2. Multiple tasks
```

0

```
        MultipleTasksExample();

        // 3. Task with return values
        TaskWithReturnValueExample();

        // 4. Parallel loops
        ParallelLoopsExample();

        // 5. Task continuation
        TaskContinuationExample();

        Console.WriteLine($"\nMain thread
{Thread.CurrentThread.ManagedThreadId} completed.");
    }

    public static void BasicTaskExample()
    {
        Console.WriteLine("1. Basic Task Example:");

        // Create and start a task
        Task task1 = Task.Run(() =>
        {
            Console.WriteLine($"Task 1 running on thread
{Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(2000);
            Console.WriteLine("Task 1 completed");
        });

        // Wait for task to complete
        task1.Wait();
        Console.WriteLine("Basic task example finished\n");
    }

    public static void MultipleTasksExample()
    {
        Console.WriteLine("2. Multiple Tasks Example:");

        // Create multiple tasks
        Task[] tasks = new Task[5];
```

```
        for (int i = 0; i < 5; i++)
        {
            int taskId = i + 1; // Capture loop variable
            tasks[i] = Task.Run(() =>
            {
                Console.WriteLine($"Task {taskId} started on thread
{Thread.CurrentThread.ManagedThreadId}");
                Thread.Sleep(1000 * taskId); // Different sleep times
                Console.WriteLine($"Task {taskId} completed");
            });
        }

        // Wait for all tasks to complete
        Task.WaitAll(tasks);
        Console.WriteLine("All tasks completed\n");
    }

    public static void TaskWithReturnValueExample()
    {
        Console.WriteLine("3. Task with Return Values:");

        // Tasks that return values
        Task<int>[] calculationTasks = new Task<int>[3];

        calculationTasks[0] = Task.Run(() =>
        {
            Console.WriteLine($"Calculating factorial on thread
{Thread.CurrentThread.ManagedThreadId}");
            return CalculateFactorial(5);
        });

        calculationTasks[1] = Task.Run(() =>
        {
            Console.WriteLine($"Calculating sum on thread
{Thread.CurrentThread.ManagedThreadId}");
            return CalculateSum(1, 100);
        });

        calculationTasks[2] = Task.Run(() =>
        {
0
```

```
            Console.WriteLine($"Calculating power on thread
{Thread.CurrentThread.ManagedThreadId}");
            return CalculatePower(2, 10);
        });

        // Wait for all tasks and get results
        Task.WaitAll(calculationTasks);

        Console.WriteLine($"Factorial result:
{calculationTasks[0].Result}");
        Console.WriteLine($"Sum result: {calculationTasks[1].Result}");
        Console.WriteLine($"Power result:
{calculationTasks[2].Result}\n");
    }

    public static void ParallelLoopsExample()
    {
        Console.WriteLine("4. Parallel Loops Example:");

        // Parallel.For example
        Console.WriteLine("Parallel.For processing:");
        Parallel.For(1, 6, i =>
        {
            Console.WriteLine($"Processing item {i} on thread
{Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(500);
        });

        // Parallel.ForEach example
        Console.WriteLine("\nParallel.ForEach processing:");
        string[] data = { "File1.txt", "File2.txt", "File3.txt",
"File4.txt" };

        Parallel.ForEach(data, file =>
        {
            Console.WriteLine($"Processing {file} on thread
{Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(800);
            Console.WriteLine($"Completed {file}");
        });
```

```
        Console.WriteLine("Parallel loops completed\n");
    }

    public static void TaskContinuationExample()
    {
        Console.WriteLine("5. Task Continuation Example:");

        Task<string> downloadTask = Task.Run(() =>
        {
            Console.WriteLine($"Downloading data on thread
{Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(2000);
            return "Downloaded data content";
        });

        Task processTask = downloadTask.ContinueWith(antecedent =>
        {
            Console.WriteLine($"Processing data on thread
{Thread.CurrentThread.ManagedThreadId}");
            string data = antecedent.Result;
            Console.WriteLine($"Processing: {data}");
            Thread.Sleep(1000);
            Console.WriteLine("Data processing completed");
        });

        // Wait for the continuation task
        processTask.Wait();
        Console.WriteLine("Task continuation example completed\n");
    }

    // Helper methods for calculations
    public static int CalculateFactorial(int n)
    {
        Thread.Sleep(1000); // Simulate work
        if (n <= 1) return 1;
        return n * CalculateFactorial(n - 1);
    }

    public static int CalculateSum(int start, int end)
```

0

```
    {
        Thread.Sleep(1000); // Simulate work
        int sum = 0;
        for (int i = start; i <= end; i++)
        {
            sum += i;
        }
        return sum;
    }

    public static int CalculatePower(int baseNum, int exponent)
    {
        Thread.Sleep(1000); // Simulate work
        return (int)Math.Pow(baseNum, exponent);
    }
}

// Advanced TPL example with cancellation
public class AdvancedTPLDemo
{
    public static void CancellationExample()
    {
        Console.WriteLine("=== Cancellation Example ===");

        CancellationTokenSource cts = new CancellationTokenSource();

        // Start a long-running task
        Task longRunningTask = Task.Run(() =>
        {
            for (int i = 1; i <= 10; i++)
            {
                // Check for cancellation
                cts.Token.ThrowIfCancellationRequested();

                Console.WriteLine($"Working... Step {i}/10 on thread
{Thread.CurrentThread.ManagedThreadId}");
                Thread.Sleep(1000);
            }
        }, cts.Token);
```

0

```
        // Cancel after 3 seconds
        cts.CancelAfter(3000);

        try
        {
            longRunningTask.Wait();
            Console.WriteLine("Task completed successfully");
        }
        catch (AggregateException ex)
        {
            if (ex.InnerException is OperationCanceledException)
            {
                Console.WriteLine("Task was cancelled");
            }
        }
    }

    public static void ExceptionHandlingExample()
    {
        Console.WriteLine("=== Exception Handling Example ===");

        Task[] tasks = new Task[3];

        tasks[0] = Task.Run(() =>
        {
            Console.WriteLine("Task 1: Running normally");
            Thread.Sleep(1000);
        });

        tasks[1] = Task.Run(() =>
        {
            Console.WriteLine("Task 2: About to throw exception");
            throw new InvalidOperationException("Something went wrong in
Task 2");
        });

        tasks[2] = Task.Run(() =>
        {
            Console.WriteLine("Task 3: Running normally");
            Thread.Sleep(1500);
```

0

```
        });

        try
        {
            Task.WaitAll(tasks);
        }
        catch (AggregateException ex)
        {
            Console.WriteLine($"Caught {ex.InnerExceptions.Count}
exceptions:");
            foreach (var innerEx in ex.InnerExceptions)
            {
                Console.WriteLine($"- {innerEx.Message}");
            }
        }
    }
}
```

}

Question 13: Provided that a MSSQL database named "LibraryDb" with table named "Books" with following columns (Id as int, ISBN as varchar(20), Title as varchar(200), Publication Date as DateTime). Write a C# program to connect to the database and insert as many books as user wants, and finally display all the books in db. Explain the difference between ExecuteReader and ExecuteNonQuery.

Complete Library Management Program:

```csharp
using System;
using System.Data;
using System.Data.SqlClient;

public class LibraryManagement
{
    private static string connectionString =
"Server=localhost;Database=LibraryDb;Integrated Security=true;";
```

0

```
    public static void Main()
    {
        try
        {
            CreateDatabaseAndTable();

            bool continueAdding = true;
            while (continueAdding)
            {
                AddBook();

                Console.Write("Do you want to add another book? (y/n): 
");
                string response = Console.ReadLine().ToLower();
                continueAdding = (response == "y" || response == "yes");
            }

            DisplayAllBooks();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }

    public static void CreateDatabaseAndTable()
    {
        string createTableQuery = @"
            IF NOT EXISTS (SELECT * FROM sysobjects WHERE name='Books' 
AND xtype='U')
            CREATE TABLE Books (
                Id INT PRIMARY KEY IDENTITY(1,1),
                ISBN VARCHAR(20) NOT NULL,
                Title VARCHAR(200) NOT NULL,
                PublicationDate DATETIME NOT NULL
            )";

        using (SqlConnection connection = new 
SqlConnection(connectionString))
        {
0
```

```
            try
            {
                connection.Open();
                using (SqlCommand command = new
SqlCommand(createTableQuery, connection))
                {
                    command.ExecuteNonQuery(); // Uses ExecuteNonQuery
for CREATE TABLE
                    Console.WriteLine("Database table 'Books'
created/verified successfully.");
                }
            }
            catch (SqlException ex)
            {
                Console.WriteLine($"Database error: {ex.Message}");
            }
        }
    }

    public static void AddBook()
    {
        try
        {
            Console.WriteLine("\n--- Add New Book ---");

            Console.Write("Enter ISBN: ");
            string isbn = Console.ReadLine();

            Console.Write("Enter Title: ");
            string title = Console.ReadLine();

            Console.Write("Enter Publication Date (yyyy-mm-dd): ");
            DateTime publicationDate =
DateTime.Parse(Console.ReadLine());

            string insertQuery = @"
                INSERT INTO Books (ISBN, Title, PublicationDate)
                VALUES (@ISBN, @Title, @PublicationDate)";

            using (SqlConnection connection = new
```

```
SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand command = new SqlCommand(insertQuery,
connection))
                {
                    // Add parameters to prevent SQL injection
                    command.Parameters.AddWithValue("@ISBN", isbn);
                    command.Parameters.AddWithValue("@Title", title);
                    command.Parameters.AddWithValue("@PublicationDate",
publicationDate);

                    int rowsAffected = command.ExecuteNonQuery(); // Uses
ExecuteNonQuery for INSERT
                    Console.WriteLine($"Book added successfully!
({rowsAffected} row(s) affected)");
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error adding book: {ex.Message}");
        }
    }

    public static void DisplayAllBooks()
    {
        string selectQuery = "SELECT Id, ISBN, Title, PublicationDate
FROM Books ORDER BY Id";

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            try
            {
                connection.Open();
                using (SqlCommand command = new SqlCommand(selectQuery,
connection))
                {
                    using (SqlDataReader reader =
```

```
command.ExecuteReader()) // Uses ExecuteReader for SELECT
                    {
                        Console.WriteLine("\n--- All Books in Library ---
");
                        Console.WriteLine($"{"ID",-5} {"ISBN",-15}
{"Title",-30} {"Publication Date",-15}");
                        Console.WriteLine(new string('-', 70));

                        while (reader.Read())
                        {
                            Console.WriteLine($"{reader["Id"],-5} " +
                                              $"{reader["ISBN"],-15} " +
                                              $"{reader["Title"],-30} " +
                                              $"
{((DateTime)reader["PublicationDate"]).ToString("yyyy-MM-dd"),-15}");
                        }
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error displaying books:
{ex.Message}");
            }
        }
    }

    // Additional methods to demonstrate ExecuteReader vs ExecuteNonQuery
    public static void DemonstrateExecuteMethods()
    {
        Console.WriteLine("\n=== ExecuteReader vs ExecuteNonQuery Demo
===");

        // ExecuteNonQuery examples
        ExecuteNonQueryExamples();

        // ExecuteReader examples
        ExecuteReaderExamples();
    }
```

0

```
    public static void ExecuteNonQueryExamples()
    {
        Console.WriteLine("\nExecuteNonQuery Examples:");

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            // INSERT example
            string insertQuery = "INSERT INTO Books (ISBN, Title,
PublicationDate) VALUES ('123-456', 'Test Book', '2023-01-01')";
            using (SqlCommand command = new SqlCommand(insertQuery,
connection))
            {
                int rowsAffected = command.ExecuteNonQuery();
                Console.WriteLine($"INSERT: {rowsAffected} row(s)
affected");
            }

            // UPDATE example
            string updateQuery = "UPDATE Books SET Title = 'Updated Test
Book' WHERE ISBN = '123-456'";
            using (SqlCommand command = new SqlCommand(updateQuery,
connection))
            {
                int rowsAffected = command.ExecuteNonQuery();
                Console.WriteLine($"UPDATE: {rowsAffected} row(s)
affected");
            }

            // DELETE example
            string deleteQuery = "DELETE FROM Books WHERE ISBN = '123-
456'";
            using (SqlCommand command = new SqlCommand(deleteQuery,
connection))
            {
                int rowsAffected = command.ExecuteNonQuery();
                Console.WriteLine($"DELETE: {rowsAffected} row(s)
affected");
```

0

```
            }
        }
    }

    public static void ExecuteReaderExamples()
    {
        Console.WriteLine("\nExecuteReader Examples:");

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            // SELECT with specific columns
            string selectQuery = "SELECT TOP 3 ISBN, Title FROM Books";
            using (SqlCommand command = new SqlCommand(selectQuery,
connection))
            {
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    Console.WriteLine("Top 3 Books (ISBN and Title
only):");
                    while (reader.Read())
                    {
                        Console.WriteLine($"ISBN: {reader["ISBN"]},
Title: {reader["Title"]}");
                    }
                }
            }
        }
    }
}
```

ExecuteReader vs ExecuteNonQuery:

Aspect	ExecuteReader	ExecuteNonQuery
Purpose	Retrieve data from database	Execute commands that don't return data
Return Type	SqlDataReader	int (rows affected)
Used For	SELECT statements	INSERT, UPDATE, DELETE, CREATE, DROP

0

Aspect	ExecuteReader	ExecuteNonQuery
Data Access	Forward-only, read-only data stream	Number of affected rows
Performance	Efficient for large datasets	Fast for modification operations
Connection	Keeps connection open while reading	Releases connection immediately

Examples:

```
// ExecuteReader - for SELECT statements
using (SqlDataReader reader = command.ExecuteReader())
{
    while (reader.Read())
    {
        // Read data row by row
        string title = reader["Title"].ToString();
    }
}

// ExecuteNonQuery - for INSERT, UPDATE, DELETE
int rowsAffected = command.ExecuteNonQuery();
Console.WriteLine($"{rowsAffected} rows were modified");
```

Question 14: Write C# code for downloading web page from a web server e.g. http://myblog.com.np/

```
using System;
using System.IO;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

public class WebPageDownloader
{
    public static async Task Main()
    {
        Console.WriteLine("=== Web Page Downloader ===");

        // Method 1: Using HttpClient (Recommended - Modern approach)
        await DownloadWithHttpClient();
```

0

```
        // Method 2: Using WebClient (Legacy approach)
        DownloadWithWebClient();

        // Method 3: Download and save to file
        await DownloadAndSaveToFile();

        // Method 4: Download with custom headers
        await DownloadWithCustomHeaders();
    }

    // Method 1: Modern approach using HttpClient
    public static async Task DownloadWithHttpClient()
    {
        Console.WriteLine("\n1. Downloading with HttpClient:");

        string url = "http://myblog.com.np/";

        try
        {
            using (HttpClient client = new HttpClient())
            {
                // Set timeout
                client.Timeout = TimeSpan.FromSeconds(30);

                // Download the web page
                string content = await client.GetStringAsync(url);

                Console.WriteLine($"Successfully downloaded
{content.Length} characters");
                Console.WriteLine("First 200 characters:");
                Console.WriteLine(content.Substring(0, Math.Min(200,
content.Length)));
                Console.WriteLine("...");
            }
        }
        catch (HttpRequestException ex)
        {
            Console.WriteLine($"HTTP Error: {ex.Message}");
        }
0
```

```
        catch (TaskCanceledException ex)
        {
            Console.WriteLine($"Timeout Error: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"General Error: {ex.Message}");
        }
    }

    // Method 2: Legacy approach using WebClient
    public static void DownloadWithWebClient()
    {
        Console.WriteLine("\n2. Downloading with WebClient:");

        string url = "http://myblog.com.np/";

        try
        {
            using (System.Net.WebClient client = new
System.Net.WebClient())
            {
                // Set encoding
                client.Encoding = Encoding.UTF8;

                // Add user agent to avoid blocking
                client.Headers.Add("User-Agent", "Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36");

                // Download the web page
                string content = client.DownloadString(url);

                Console.WriteLine($"Successfully downloaded
{content.Length} characters");
                Console.WriteLine("First 200 characters:");
                Console.WriteLine(content.Substring(0, Math.Min(200,
content.Length)));
                Console.WriteLine("...");
            }
        }
```

0

```
        catch (System.Net.WebException ex)
        {
            Console.WriteLine($"Web Error: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"General Error: {ex.Message}");
        }
    }

    // Method 3: Download and save to file
    public static async Task DownloadAndSaveToFile()
    {
        Console.WriteLine("\n3. Downloading and saving to file:");

        string url = "http://myblog.com.np/";
        string fileName = "downloaded_webpage.html";

        try
        {
            using (HttpClient client = new HttpClient())
            {
                // Download content
                string content = await client.GetStringAsync(url);

                // Save to file
                await File.WriteAllTextAsync(fileName, content,
Encoding.UTF8);

                Console.WriteLine($"Web page saved to: {fileName}");
                Console.WriteLine($"File size: {new
FileInfo(fileName).Length} bytes");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
```

0

```
    // Method 4: Download with custom headers and detailed response info
    public static async Task DownloadWithCustomHeaders()
    {
        Console.WriteLine("\n4. Downloading with custom headers:");

        string url = "http://myblog.com.np/";

        try
        {
            using (HttpClient client = new HttpClient())
            {
                // Add custom headers
                client.DefaultRequestHeaders.Add("User-Agent", "C# Web
Downloader v1.0");
                client.DefaultRequestHeaders.Add("Accept",
"text/html,application/xhtml+xml");

                // Get full response (not just content)
                HttpResponseMessage response = await
client.GetAsync(url);

                Console.WriteLine($"Status Code: {response.StatusCode}");
                Console.WriteLine($"Content Type:
{response.Content.Headers.ContentType}");
                Console.WriteLine($"Content Length:
{response.Content.Headers.ContentLength}");

                // Check if successful
                if (response.IsSuccessStatusCode)
                {
                    string content = await
response.Content.ReadAsStringAsync();
                    Console.WriteLine($"Successfully downloaded
{content.Length} characters");

                    // Display response headers
                    Console.WriteLine("\nResponse Headers:");
                    foreach (var header in response.Headers)
                    {
                        Console.WriteLine($"{header.Key}: {string.Join(",
```

0

```
", header.Value)}");
                    }
                }
                else
                {
                    Console.WriteLine($"Failed to download:
{response.ReasonPhrase}");
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }

    // Bonus: Download multiple pages concurrently
    public static async Task DownloadMultiplePages()
    {
        Console.WriteLine("\n5. Downloading multiple pages
concurrently:");

        string[] urls = {
            "http://myblog.com.np/",
            "http://google.com/",
            "http://github.com/"
        };

        using (HttpClient client = new HttpClient())
        {
            // Create tasks for all downloads
            Task<string>[] downloadTasks = new Task<string>[urls.Length];

            for (int i = 0; i < urls.Length; i++)
            {
                string url = urls[i];
                downloadTasks[i] = client.GetStringAsync(url);
            }

            try
0
```

```
            {
                // Wait for all downloads to complete
                string[] results = await Task.WhenAll(downloadTasks);

                for (int i = 0; i < urls.Length; i++)
                {
                    Console.WriteLine($"{urls[i]}: {results[i].Length}
characters downloaded");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error in concurrent downloads:
{ex.Message}");
            }
        }
    }
}

// Alternative implementation with more error handling
public class RobustWebDownloader
{
    private static readonly HttpClient httpClient = new HttpClient();

    static RobustWebDownloader()
    {
        // Configure HttpClient
        httpClient.Timeout = TimeSpan.FromSeconds(30);
        httpClient.DefaultRequestHeaders.Add("User-Agent",
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36");
    }

    public static async Task<string> DownloadWebPageAsync(string url)
    {
        try
        {
            Console.WriteLine($"Downloading: {url}");

            using (HttpResponseMessage response = await
```

```
httpClient.GetAsync(url))
            {
                response.EnsureSuccessStatusCode();
                string content = await
response.Content.ReadAsStringAsync();

                Console.WriteLine($"Download completed: {content.Length}
characters");
                return content;
            }
        }
        catch (HttpRequestException ex)
        {
            Console.WriteLine($"HTTP request failed: {ex.Message}");
            return null;
        }
        catch (TaskCanceledException ex)
        {
            Console.WriteLine($"Request timed out: {ex.Message}");
            return null;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Unexpected error: {ex.Message}");
            return null;
        }
    }

    public static async Task<bool> DownloadToFileAsync(string url, string
fileName)
    {
        try
        {
            string content = await DownloadWebPageAsync(url);
            if (content != null)
            {
                await File.WriteAllTextAsync(fileName, content,
Encoding.UTF8);
                Console.WriteLine($"Content saved to: {fileName}");
                return true;
```

```
            }
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error saving to file: {ex.Message}");
            return false;
        }
    }
}
```

Question 15: Write a C# program to inherit a class BOX with properties Length, Breadth and Height. Class Box should have a read-only property named volume. Also add a method to display the properties in the console.

```
using System;

// Base class Box
public class Box
{
    // Protected fields - accessible to derived classes
    protected double length;
    protected double breadth;
    protected double height;

    // Default constructor
    public Box()
    {
        length = 0;
        breadth = 0;
        height = 0;
    }

    // Parameterized constructor
    public Box(double length, double breadth, double height)
    {
        this.length = length;
```

0

```
        this.breadth = breadth;
        this.height = height;
    }

    // Properties with validation
    public double Length
    {
        get { return length; }
        set
        {
            if (value >= 0)
                length = value;
            else
                throw new ArgumentException("Length cannot be negative");
        }
    }

    public double Breadth
    {
        get { return breadth; }
        set
        {
            if (value >= 0)
                breadth = value;
            else
                throw new ArgumentException("Breadth cannot be
negative");
        }
    }

    public double Height
    {
        get { return height; }
        set
        {
            if (value >= 0)
                height = value;
            else
                throw new ArgumentException("Height cannot be negative");
        }
```

0

```
    }

    // Read-only property for Volume
    public virtual double Volume
    {
        get { return length * breadth * height; }
    }

    // Virtual method to display properties (can be overridden)
    public virtual void DisplayProperties()
    {
        Console.WriteLine("=== Box Properties ===");
        Console.WriteLine($"Length: {length:F2}");
        Console.WriteLine($"Breadth: {breadth:F2}");
        Console.WriteLine($"Height: {height:F2}");
        Console.WriteLine($"Volume: {Volume:F2}");
        Console.WriteLine("======================");
    }

    // Method to calculate surface area
    public virtual double GetSurfaceArea()
    {
        return 2 * (length * breadth + breadth * height + height *
length);
    }
}

// Derived class - Cube (inherits from Box)
public class Cube : Box
{
    // Constructor for cube (all sides equal)
    public Cube(double side) : base(side, side, side)
    {
    }

    // Property for side (all dimensions are equal in a cube)
    public double Side
    {
        get { return length; }
        set
```

```
        {
            if (value >= 0)
            {
                length = breadth = height = value;
            }
            else
            {
                throw new ArgumentException("Side cannot be negative");
            }
        }
    }

    // Override display method for cube-specific output
    public override void DisplayProperties()
    {
        Console.WriteLine("=== Cube Properties ===");
        Console.WriteLine($"Side: {length:F2}");
        Console.WriteLine($"Volume: {Volume:F2}");
        Console.WriteLine($"Surface Area: {GetSurfaceArea():F2}");
        Console.WriteLine("======================");
    }
}

// Derived class - Rectangular Box with additional features
public class RectangularBox : Box
{
    private string material;
    private double weight;

    public RectangularBox(double length, double breadth, double height,
string material = "Cardboard")
        : base(length, breadth, height)
    {
        this.material = material;
        this.weight = 0;
    }

    public string Material
    {
        get { return material; }
```

```
        set { material = value ?? "Unknown"; }
    }

    public double Weight
    {
        get { return weight; }
        set
        {
            if (value >= 0)
                weight = value;
            else
                throw new ArgumentException("Weight cannot be negative");
        }
    }

    // Calculate density (weight per unit volume)
    public double Density
    {
        get
        {
            return Volume > 0 ? weight / Volume : 0;
        }
    }

    // Override display method
    public override void DisplayProperties()
    {
        Console.WriteLine("=== Rectangular Box Properties ===");
        Console.WriteLine($"Length: {length:F2}");
        Console.WriteLine($"Breadth: {breadth:F2}");
        Console.WriteLine($"Height: {height:F2}");
        Console.WriteLine($"Volume: {Volume:F2}");
        Console.WriteLine($"Surface Area: {GetSurfaceArea():F2}");
        Console.WriteLine($"Material: {material}");
        Console.WriteLine($"Weight: {weight:F2}");
        Console.WriteLine($"Density: {Density:F4}");
        Console.WriteLine("================================");
    }
}
```

0

```
// Another derived class - Gift Box
public class GiftBox : Box
{
    private string color;
    private bool hasRibbon;
    private string giftMessage;

    public GiftBox(double length, double breadth, double height, string
color = "Red")
        : base(length, breadth, height)
    {
        this.color = color;
        this.hasRibbon = false;
        this.giftMessage = "";
    }

    public string Color
    {
        get { return color; }
        set { color = value ?? "Unknown"; }
    }

    public bool HasRibbon
    {
        get { return hasRibbon; }
        set { hasRibbon = value; }
    }

    public string GiftMessage
    {
        get { return giftMessage; }
        set { giftMessage = value ?? ""; }
    }

    // Calculate wrapping paper needed (with some extra)
    public double WrappingPaperNeeded
    {
        get { return GetSurfaceArea() * 1.2; } // 20% extra for overlap
    }
```

0

```
    public override void DisplayProperties()
    {
        Console.WriteLine("=== Gift Box Properties ===");
        Console.WriteLine($"Dimensions: {length:F2} x {breadth:F2} x
{height:F2}");
        Console.WriteLine($"Volume: {Volume:F2}");
        Console.WriteLine($"Color: {color}");
        Console.WriteLine($"Has Ribbon: {(hasRibbon ? "Yes" : "No")}");
        Console.WriteLine($"Gift Message:
{(string.IsNullOrEmpty(giftMessage) ? "None" : giftMessage)}");
        Console.WriteLine($"Wrapping Paper Needed:
{WrappingPaperNeeded:F2}");
        Console.WriteLine("============================");
    }
}

// Main program to demonstrate inheritance
public class BoxInheritanceDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Box Inheritance Demonstration ===\n");

        try
        {
            // Create base Box object
            Console.WriteLine("1. Basic Box:");
            Box basicBox = new Box(10, 8, 6);
            basicBox.DisplayProperties();

            // Create Cube object
            Console.WriteLine("\n2. Cube:");
            Cube cube = new Cube(5);
            cube.DisplayProperties();

            // Create Rectangular Box
            Console.WriteLine("\n3. Rectangular Box:");
            RectangularBox rectBox = new RectangularBox(12, 8, 4,
"Wood");
            rectBox.Weight = 2.5;
```

```
            rectBox.DisplayProperties();

            // Create Gift Box
            Console.WriteLine("\n4. Gift Box:");
            GiftBox giftBox = new GiftBox(15, 10, 8, "Blue");
            giftBox.HasRibbon = true;
            giftBox.GiftMessage = "Happy Birthday!";
            giftBox.DisplayProperties();

            // Demonstrate polymorphism
            Console.WriteLine("\n5. Polymorphism Demo:");
            DemonstratePolymorphism();

            // Interactive box creation
            Console.WriteLine("\n6. Create Your Own Box:");
            CreateInteractiveBox();
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Validation Error: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }

    public static void DemonstratePolymorphism()
    {
        // Array of Box references pointing to different derived objects
        Box[] boxes = {
            new Box(5, 4, 3),
            new Cube(4),
            new RectangularBox(6, 5, 4, "Plastic"),
            new GiftBox(8, 6, 5, "Green")
        };

        Console.WriteLine("Processing different box types
polymorphically:");
        foreach (Box box in boxes)
0
```

```
        {
            Console.WriteLine($"Box type: {box.GetType().Name}");
            Console.WriteLine($"Volume: {box.Volume:F2}");
            Console.WriteLine($"Surface Area:
{box.GetSurfaceArea():F2}");
            Console.WriteLine("---");
        }
    }

    public static void CreateInteractiveBox()
    {
        try
        {
            Console.Write("Enter length: ");
            double length = double.Parse(Console.ReadLine());

            Console.Write("Enter breadth: ");
            double breadth = double.Parse(Console.ReadLine());

            Console.Write("Enter height: ");
            double height = double.Parse(Console.ReadLine());

            Box userBox = new Box(length, breadth, height);

            Console.WriteLine("\nYour custom box:");
            userBox.DisplayProperties();

            // Modify properties
            Console.Write("Want to modify length? (y/n): ");
            if (Console.ReadLine().ToLower() == "y")
            {
                Console.Write("Enter new length: ");
                userBox.Length = double.Parse(Console.ReadLine());

                Console.WriteLine("\nUpdated box:");
                userBox.DisplayProperties();
            }
        }
        catch (FormatException)
        {
0
```

```
            Console.WriteLine("Invalid input format. Please enter numeric
values.");
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Invalid value: {ex.Message}");
        }
    }
}
```

Question 16: What is event in C#? Explain briefly about event source and event listener. How are delegate and event related?

What is an Event?

An **event** in C# is a special kind of multicast delegate that provides notifications when something of interest happens. Events follow the publisher-subscriber pattern where:

- **Event Source (Publisher)**: The class that raises the event
- **Event Listener (Subscriber)**: The class that handles the event

Event Source and Event Listener:

```
using System;

// Event arguments class
public class TemperatureChangedEventArgs : EventArgs
{
    public double OldTemperature { get; }
    public double NewTemperature { get; }
    public DateTime TimeStamp { get; }

    public TemperatureChangedEventArgs(double oldTemp, double newTemp)
    {
        OldTemperature = oldTemp;
        NewTemperature = newTemp;
        TimeStamp = DateTime.Now;
    }
}
```

0

```
// Event Source (Publisher)
public class TemperatureSensor
{
    private double temperature;

    // Declare event using delegate
    public event EventHandler<TemperatureChangedEventArgs>
TemperatureChanged;

    public double Temperature
    {
        get { return temperature; }
        set
        {
            if (Math.Abs(temperature - value) > 0.1) // Only fire if
significant change
            {
                double oldTemp = temperature;
                temperature = value;
                OnTemperatureChanged(oldTemp, temperature);
            }
        }
    }

    // Protected virtual method to raise the event
    protected virtual void OnTemperatureChanged(double oldTemp, double
newTemp)
    {
        TemperatureChanged?.Invoke(this, new
TemperatureChangedEventArgs(oldTemp, newTemp));
    }

    // Method to simulate temperature reading
    public void StartMonitoring()
    {
        Console.WriteLine("Temperature monitoring started...");
        Random random = new Random();

        for (int i = 0; i < 10; i++)
        {
```

0

```
            Temperature = 20 + random.NextDouble() * 15; // Random temp
between 20-35
            System.Threading.Thread.Sleep(1000);
        }
    }
}

// Event Listeners (Subscribers)
public class TemperatureDisplay
{
    private string deviceName;

    public TemperatureDisplay(string name)
    {
        deviceName = name;
    }

    // Event handler method
    public void OnTemperatureChanged(object sender,
TemperatureChangedEventArgs e)
    {
        Console.WriteLine($"[{deviceName}] Temperature changed from
{e.OldTemperature:F1}Â°C to {e.NewTemperature:F1}Â°C at
{e.TimeStamp:HH:mm:ss}");
    }
}

public class TemperatureAlarm
{
    private double alertThreshold;

    public TemperatureAlarm(double threshold)
    {
        alertThreshold = threshold;
    }

    public void OnTemperatureChanged(object sender,
TemperatureChangedEventArgs e)
    {
        if (e.NewTemperature > alertThreshold)
```

```
        {
            Console.WriteLine($"ðŸš¨ ALERT: Temperature
{e.NewTemperature:F1}Â°C exceeds threshold {alertThreshold:F1}Â°C!");
        }
    }
}

public class TemperatureLogger
{
    public void OnTemperatureChanged(object sender,
TemperatureChangedEventArgs e)
    {
        string logEntry = $"[LOG] {e.TimeStamp:yyyy-MM-dd HH:mm:ss} -
Temperature: {e.NewTemperature:F2}Â°C";
        Console.WriteLine(logEntry);

        // In real application, write to file or database
        System.IO.File.AppendAllText("temperature.log", logEntry +
Environment.NewLine);
    }
}

// Main program demonstrating events
public class EventDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Event Demonstration ===\n");

        // Create event source
        TemperatureSensor sensor = new TemperatureSensor();

        // Create event listeners
        TemperatureDisplay display = new TemperatureDisplay("Main
Display");
        TemperatureAlarm alarm = new TemperatureAlarm(30.0);
        TemperatureLogger logger = new TemperatureLogger();

        // Subscribe to events (register event handlers)
        sensor.TemperatureChanged += display.OnTemperatureChanged;
```

0

```
        sensor.TemperatureChanged += alarm.OnTemperatureChanged;
        sensor.TemperatureChanged += logger.OnTemperatureChanged;

        // Add anonymous event handler
        sensor.TemperatureChanged += (sender, e) =>
        {
            if (e.NewTemperature < 22)
            {
                Console.WriteLine("â„ï¸ Temperature is getting cold!");
            }
        };

        // Start monitoring (this will trigger events)
        sensor.StartMonitoring();

        Console.WriteLine("\n--- Unsubscribing some handlers ---");

        // Unsubscribe some handlers
        sensor.TemperatureChanged -= alarm.OnTemperatureChanged;

        Console.WriteLine("Setting final temperature...");
        sensor.Temperature = 25.5;

        DemonstrateCustomEvents();
    }

    public static void DemonstrateCustomEvents()
    {
        Console.WriteLine("\n=== Custom Event Example ===");

        var player = new MusicPlayer();
        var display = new MusicDisplay();
        var notification = new MusicNotification();

        // Subscribe to events
        player.SongStarted += display.OnSongStarted;
        player.SongStarted += notification.OnSongStarted;
        player.SongEnded += display.OnSongEnded;

        // Play some songs
```

```
        player.PlaySong("Bohemian Rhapsody", "Queen");
        player.PlaySong("Hotel California", "Eagles");
    }
}

// Custom event example - Music Player
public class SongEventArgs : EventArgs
{
    public string SongTitle { get; }
    public string Artist { get; }
    public DateTime EventTime { get; }

    public SongEventArgs(string title, string artist)
    {
        SongTitle = title;
        Artist = artist;
        EventTime = DateTime.Now;
    }
}

public class MusicPlayer
{
    public event EventHandler<SongEventArgs> SongStarted;
    public event EventHandler<SongEventArgs> SongEnded;

    public void PlaySong(string title, string artist)
    {
        Console.WriteLine($"ðŸŽµ Playing: {title} by {artist}");

        // Raise SongStarted event
        OnSongStarted(new SongEventArgs(title, artist));

        // Simulate song playing
        System.Threading.Thread.Sleep(2000);

        // Raise SongEnded event
        OnSongEnded(new SongEventArgs(title, artist));
    }

    protected virtual void OnSongStarted(SongEventArgs e)
```

0

```
    {
        SongStarted?.Invoke(this, e);
    }

    protected virtual void OnSongEnded(SongEventArgs e)
    {
        SongEnded?.Invoke(this, e);
    }
}

public class MusicDisplay
{
    public void OnSongStarted(object sender, SongEventArgs e)
    {
        Console.WriteLine($"ðŸ“º Now Playing: {e.SongTitle} -
{e.Artist}");
    }

    public void OnSongEnded(object sender, SongEventArgs e)
    {
        Console.WriteLine($"ðŸ“º Finished: {e.SongTitle}");
    }
}

public class MusicNotification
{
    public void OnSongStarted(object sender, SongEventArgs e)
    {
        Console.WriteLine($"ðŸ”” Notification: Started playing
{e.SongTitle}");
    }
}
```

Relationship between Delegates and Events:

Aspect	Delegate	Event
Definition	Type-safe function pointer	Special form of multicast delegate
Access	Can be called directly from outside	Can only be raised from within the class
Assignment	Supports = operator	Only supports += and -= operators

0

Aspect	Delegate	Event
Security	Less secure (external classes can invoke)	More secure (encapsulated)
Purpose	General callback mechanism	Notification mechanism

```
// Delegate vs Event comparison
public class DelegateVsEventDemo
{
    // Regular delegate
    public Action<string> MyDelegate;

    // Event based on delegate
    public event Action<string> MyEvent;

    public void DemonstrateRelationship()
    {
        // Delegate usage
        MyDelegate = msg => Console.WriteLine($"Delegate: {msg}");
        MyDelegate += msg => Console.WriteLine($"Delegate 2: {msg}");

        // Event usage
        MyEvent += msg => Console.WriteLine($"Event: {msg}");

        // Calling delegate directly (allowed)
        MyDelegate?.Invoke("Hello from delegate");

        // Calling event directly (NOT allowed from outside class)
        // MyEvent?.Invoke("Hello from event"); // Would cause compile
error if called from outside

        // Proper way to raise event
        OnMyEvent("Hello from event");
    }

    protected virtual void OnMyEvent(string message)
    {
        MyEvent?.Invoke(message); // Can only be called from within the
class
    }
}
```

0

Key Points:

1. **Events are built on delegates** - They use delegates internally
2. **Events provide encapsulation** - Only the class that declares the event can raise it
3. **Events follow conventions** - Usually named with verbs (Started, Changed, Completed)
4. **Event handlers follow pattern** - `EventHandler<T>` or custom delegate types
5. **Events support += and -= only** - Cannot be assigned directly with = operator

Question 17: What are the access modifiers in C#? Explain each of them in brief.

Access Modifiers in C#:

C# provides several access modifiers that control the visibility and accessibility of classes, methods, fields, and other members.

```
using System;

// Demonstration of all access modifiers
public class AccessModifierDemo
{
    // PUBLIC - Accessible from anywhere
    public string PublicField = "I'm accessible from anywhere";

    // PRIVATE - Accessible only within the same class
    private string privateField = "I'm only accessible within this
class";

    // PROTECTED - Accessible within the same class and derived classes
    protected string protectedField = "I'm accessible in this class and
derived classes";

    // INTERNAL - Accessible within the same assembly
    internal string internalField = "I'm accessible within the same
assembly";

    // PROTECTED INTERNAL - Accessible within same assembly OR derived
classes
    protected internal string protectedInternalField = "I'm accessible
within assembly or derived classes";
```

0

```
    // PRIVATE PROTECTED - Accessible within same assembly AND derived
classes only
    private protected string privateProtectedField = "I'm accessible in
derived classes within same assembly";

    public AccessModifierDemo()
    {
        // All fields are accessible within the same class
        Console.WriteLine("=== Inside AccessModifierDemo Constructor
===");
        Console.WriteLine($"Public: {PublicField}");
        Console.WriteLine($"Private: {privateField}");
        Console.WriteLine($"Protected: {protectedField}");
        Console.WriteLine($"Internal: {internalField}");
        Console.WriteLine($"Protected Internal:
{protectedInternalField}");
        Console.WriteLine($"Private Protected: {privateProtectedField}");
    }

    // PUBLIC method
    public void PublicMethod()
    {
        Console.WriteLine("Public method - accessible from anywhere");
        PrivateMethod(); // Can call private method from within same
class
    }

    // PRIVATE method
    private void PrivateMethod()
    {
        Console.WriteLine("Private method - only accessible within this
class");
    }

    // PROTECTED method
    protected void ProtectedMethod()
    {
        Console.WriteLine("Protected method - accessible in derived
classes");
```

```
    }

    // INTERNAL method
    internal void InternalMethod()
    {
        Console.WriteLine("Internal method - accessible within same
assembly");
    }

    // PROTECTED INTERNAL method
    protected internal void ProtectedInternalMethod()
    {
        Console.WriteLine("Protected Internal method");
    }

    // PRIVATE PROTECTED method
    private protected void PrivateProtectedMethod()
    {
        Console.WriteLine("Private Protected method");
    }
}

// DERIVED CLASS - demonstrating inherited access
public class DerivedClass : AccessModifierDemo
{
    public void TestInheritedAccess()
    {
        Console.WriteLine("\n=== Inside Derived Class ===");

        // Accessible in derived class
        Console.WriteLine($"Public: {PublicField}");
        // Console.WriteLine($"Private: {privateField}"); // ERROR: Not
accessible
        Console.WriteLine($"Protected: {protectedField}");
        Console.WriteLine($"Internal: {internalField}");
        Console.WriteLine($"Protected Internal:
{protectedInternalField}");
        Console.WriteLine($"Private Protected: {privateProtectedField}");

        // Method calls
```

0

```
        PublicMethod();
        // PrivateMethod(); // ERROR: Not accessible
        ProtectedMethod();
        InternalMethod();
        ProtectedInternalMethod();
        PrivateProtectedMethod();
    }
}

// SEPARATE CLASS in same assembly
public class SeparateClass
{
    public void TestExternalAccess()
    {
        Console.WriteLine("\n=== Inside Separate Class (Same Assembly)
===");

        AccessModifierDemo obj = new AccessModifierDemo();

        // Accessible from separate class in same assembly
        Console.WriteLine($"Public: {obj.PublicField}");
        // Console.WriteLine($"Private: {obj.privateField}"); // ERROR:
Not accessible
        // Console.WriteLine($"Protected: {obj.protectedField}"); //
ERROR: Not accessible (not derived)
        Console.WriteLine($"Internal: {obj.internalField}");
        Console.WriteLine($"Protected Internal:
{obj.protectedInternalField}");
        // Console.WriteLine($"Private Protected:
{obj.privateProtectedField}"); // ERROR: Not accessible (not derived)

        // Method calls
        obj.PublicMethod();
        // obj.PrivateMethod(); // ERROR: Not accessible
        // obj.ProtectedMethod(); // ERROR: Not accessible (not derived)
        obj.InternalMethod();
        obj.ProtectedInternalMethod();
        // obj.PrivateProtectedMethod(); // ERROR: Not accessible (not
derived)
    }
0
```

```
}

// CLASS ACCESS MODIFIERS
public class PublicClass
{
    public void Method() { Console.WriteLine("Public class method"); }
}

internal class InternalClass
{
    public void Method() { Console.WriteLine("Internal class method"); }
}

// NESTED CLASS ACCESS MODIFIERS
public class OuterClass
{
    private int outerPrivateField = 100;

    // Public nested class
    public class PublicNestedClass
    {
        public void AccessOuter(OuterClass outer)
        {
            // Nested class can access private members of outer class
            Console.WriteLine($"Accessing outer private field:
{outer.outerPrivateField}");
        }
    }

    // Private nested class
    private class PrivateNestedClass
    {
        public void Method()
        {
            Console.WriteLine("Private nested class method");
        }
    }

    // Protected nested class
    protected class ProtectedNestedClass
0
```

```
    {
        public void Method()
        {
            Console.WriteLine("Protected nested class method");
        }
    }

    public void CreateNestedInstances()
    {
        var publicNested = new PublicNestedClass();
        var privateNested = new PrivateNestedClass();
        var protectedNested = new ProtectedNestedClass();

        publicNested.AccessOuter(this);
        privateNested.Method();
        protectedNested.Method();
    }
}

// INTERFACE ACCESS MODIFIERS
public interface IPublicInterface
{
    void PublicInterfaceMethod();
}

internal interface IInternalInterface
{
    void InternalInterfaceMethod();
}

// PROPERTY ACCESS MODIFIERS
public class PropertyAccessDemo
{
    private string _name;

    // Property with different access levels for get/set
    public string Name
    {
        get { return _name; }
        private set { _name = value; } // Private setter
```

0

```
    }

    // Auto-property with private setter
    public int Id { get; private set; }

    // Protected setter
    public DateTime CreatedDate { get; protected set; }

    public PropertyAccessDemo(string name, int id)
    {
        Name = name; // Can set within same class
        Id = id;
        CreatedDate = DateTime.Now;
    }
}

// MAIN PROGRAM
public class Program
{
    public static void Main()
    {
        Console.WriteLine("=== C# Access Modifiers Demonstration ===");

        // Test base class
        AccessModifierDemo baseObj = new AccessModifierDemo();
        baseObj.PublicMethod();
        // baseObj.PrivateMethod(); // ERROR: Not accessible
        baseObj.InternalMethod();

        // Test derived class
        DerivedClass derivedObj = new DerivedClass();
        derivedObj.TestInheritedAccess();

        // Test separate class
        SeparateClass separateObj = new SeparateClass();
        separateObj.TestExternalAccess();

        // Test nested classes
        Console.WriteLine("\n=== Nested Classes ===");
        OuterClass outerObj = new OuterClass();
```

0

```
        outerObj.CreateNestedInstances();

        var publicNested = new OuterClass.PublicNestedClass();
        // var privateNested = new OuterClass.PrivateNestedClass(); //
ERROR: Not accessible

        // Test property access
        Console.WriteLine("\n=== Property Access ===");
        PropertyAccessDemo propObj = new PropertyAccessDemo("Test", 123);
        Console.WriteLine($"Name: {propObj.Name}");
        Console.WriteLine($"ID: {propObj.Id}");
        // propObj.Name = "New Name"; // ERROR: Setter is private
        // propObj.Id = 456; // ERROR: Setter is private
    }
}
```

Summary of Access Modifiers:

Access Modifier	Class Level	Member Level	Accessibility
public	âœ“	âœ“	Everywhere
private	âœ—	âœ“	Same class only
protected	âœ—	âœ“	Same class + derived classes
internal	âœ“	âœ“	Same assembly
protected internal	âœ—	âœ“	Same assembly OR derived classes
private protected	âœ—	âœ“	Same assembly AND derived classes

Key Points:

1. **Default Access Levels:**
    - Classes: `internal`
    - Class members: `private`
    - Interface members: `public`
2. **Most Restrictive to Least Restrictive:**
    - private -> private protected -> protected -> internal -> protected internal -> public
3. **Best Practices:**

0

- Use the most restrictive access level possible
- Prefer `private` for implementation details
- Use `public` only for intended API
- Use `protected` for extensibility in inheritance
- Use `internal` for assembly-level collaboration

Question 18: What do we mean by exceptions? Write basic C# code using try catch to catch an arithmetic exception.

What are Exceptions?

Exceptions are runtime errors that occur during program execution. They represent unexpected or exceptional circumstances that disrupt the normal flow of a program. C# uses a structured exception handling mechanism with try-catch-finally blocks.

Exception Handling with Try-Catch:

```
using System;

public class ExceptionHandlingDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Exception Handling Demonstration ===\n");

        // Basic arithmetic exception handling
        BasicArithmeticExceptionDemo();

        // Multiple exception types
        MultipleExceptionDemo();

        // Finally block demonstration
        FinallyBlockDemo();

        // Custom exception handling
        CustomExceptionDemo();

        // Nested try-catch
        NestedTryCatchDemo();
    }
```

```
    // Basic arithmetic exception handling
    public static void BasicArithmeticExceptionDemo()
    {
        Console.WriteLine("1. Basic Arithmetic Exception Handling:");

        try
        {
            Console.Write("Enter first number: ");
            int number1 = int.Parse(Console.ReadLine());

            Console.Write("Enter second number: ");
            int number2 = int.Parse(Console.ReadLine());

            // This can throw DivideByZeroException
            int result = number1 / number2;
            Console.WriteLine($"Result: {number1} / {number2} =
{result}");
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine($"Arithmetic Error: Cannot divide by
zero!");
            Console.WriteLine($"Exception Message: {ex.Message}");
        }
        catch (FormatException ex)
        {
            Console.WriteLine($"Input Error: Please enter valid
integers!");
            Console.WriteLine($"Exception Message: {ex.Message}");
        }
        catch (OverflowException ex)
        {
            Console.WriteLine($"Overflow Error: Number is too large!");
            Console.WriteLine($"Exception Message: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"General Error: {ex.Message}");
        }
```

0

```
        Console.WriteLine();
    }

    // Multiple arithmetic operations with exception handling
    public static void MultipleExceptionDemo()
    {
        Console.WriteLine("2. Multiple Arithmetic Operations:");

        int[] numbers = { 10, 5, 0, -3 };
        int divisor = 0;

        for (int i = 0; i < numbers.Length; i++)
        {
            try
            {
                Console.WriteLine($"Processing number: {numbers[i]}");

                // Division by zero exception
                int division = numbers[i] / divisor;
                Console.WriteLine($"Division result: {division}");

                // Array index out of bounds
                int nextNumber = numbers[i + 10]; // Will throw
IndexOutOfRangeException

                // Arithmetic overflow (if using checked context)
                checked
                {
                    int overflow = int.MaxValue + 1;
                }
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("  â Division by zero detected!");
            }
            catch (IndexOutOfRangeException)
            {
                Console.WriteLine("  â Array index out of range!");
            }
```

```
            catch (OverflowException)
            {
                Console.WriteLine("  âŒ Arithmetic overflow occurred!");
            }
            catch (ArithmeticException ex)
            {
                Console.WriteLine($"  âŒ Arithmetic exception:
{ex.Message}");
            }
        }

        Console.WriteLine();
    }

    // Finally block demonstration
    public static void FinallyBlockDemo()
    {
        Console.WriteLine("3. Finally Block Demonstration:");

        System.IO.FileStream fileStream = null;

        try
        {
            // Simulate file operations that might fail
            Console.WriteLine("Opening file...");
            fileStream = new System.IO.FileStream("test.txt",
System.IO.FileMode.Create);

            Console.Write("Enter a number to write to file: ");
            int number = int.Parse(Console.ReadLine());

            // Potential arithmetic exception
            int result = 100 / number;

            byte[] data = System.Text.Encoding.UTF8.GetBytes($"Result:
{result}");
            fileStream.Write(data, 0, data.Length);

            Console.WriteLine("File written successfully!");
        }
```

```
        catch (DivideByZeroException)
        {
            Console.WriteLine("âŒ Cannot divide by zero while writing to
file!");
        }
        catch (FormatException)
        {
            Console.WriteLine("âŒ Invalid number format!");
        }
        catch (System.IO.IOException ex)
        {
            Console.WriteLine($"âŒ File I/O error: {ex.Message}");
        }
        finally
        {
            // This block always executes, even if exception occurs
            Console.WriteLine("Executing finally block...");

            if (fileStream != null)
            {
                fileStream.Close();
                Console.WriteLine("File stream closed.");
            }

            // Clean up temporary file
            if (System.IO.File.Exists("test.txt"))
            {
                System.IO.File.Delete("test.txt");
                Console.WriteLine("Temporary file deleted.");
            }
        }

        Console.WriteLine();
    }

    // Custom exception handling
    public static void CustomExceptionDemo()
    {
        Console.WriteLine("4. Custom Exception Handling:");
```

0

```
        try
        {
            Calculator calculator = new Calculator();

            // Operations that might throw custom exceptions
            Console.WriteLine("Testing calculator operations:");

            double result1 = calculator.Divide(10, 2);
            Console.WriteLine($"10 / 2 = {result1}");

            double result2 = calculator.Divide(10, 0); // Will throw
custom exception
        }
        catch (CalculatorException ex)
        {
            Console.WriteLine($"âŒ Calculator Error: {ex.Message}");
            Console.WriteLine($"Error Code: {ex.ErrorCode}");
        }
        catch (ArithmeticException ex)
        {
            Console.WriteLine($"âŒ Arithmetic Error: {ex.Message}");
        }

        Console.WriteLine();
    }

    // Nested try-catch blocks
    public static void NestedTryCatchDemo()
    {
        Console.WriteLine("5. Nested Try-Catch Blocks:");

        try
        {
            Console.WriteLine("Outer try block");

            try
            {
                Console.WriteLine("Inner try block");

                Console.Write("Enter a number for nested operation: ");
0
```

```
                int number = int.Parse(Console.ReadLine());

                // Nested arithmetic operation
                int result = 1000 / number;

                // Array operation that might fail
                int[] array = new int[number];
                array[number] = 42; // Index out of bounds if number > 0
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("Inner catch: Division by zero
handled");
                throw; // Re-throw to outer catch
            }
            catch (IndexOutOfRangeException)
            {
                Console.WriteLine("Inner catch: Array index error handled
locally");
                // Not re-throwing, so outer catch won't see this
            }
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Outer catch: Handling re-thrown division
by zero");
        }
        catch (FormatException)
        {
            Console.WriteLine("Outer catch: Invalid input format");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Outer catch: General exception -
{ex.Message}");
        }

        Console.WriteLine();
    }
}
```

0

```
// Custom Exception Class
public class CalculatorException : ArithmeticException
{
    public int ErrorCode { get; }

    public CalculatorException(string message, int errorCode) :
base(message)
    {
        ErrorCode = errorCode;
    }

    public CalculatorException(string message, int errorCode, Exception
innerException)
        : base(message, innerException)
    {
        ErrorCode = errorCode;
    }
}

// Calculator class with custom exception handling
public class Calculator
{
    public double Add(double a, double b)
    {
        try
        {
            checked
            {
                return a + b;
            }
        }
        catch (OverflowException)
        {
            throw new CalculatorException("Addition overflow occurred",
1001);
        }
    }

    public double Subtract(double a, double b)
```

```
    {
        try
        {
            checked
            {
                return a - b;
            }
        }
        catch (OverflowException)
        {
            throw new CalculatorException("Subtraction overflow
occurred", 1002);
        }
    }

    public double Multiply(double a, double b)
    {
        try
        {
            checked
            {
                return a * b;
            }
        }
        catch (OverflowException)
        {
            throw new CalculatorException("Multiplication overflow
occurred", 1003);
        }
    }

    public double Divide(double dividend, double divisor)
    {
        if (divisor == 0)
        {
            throw new CalculatorException("Division by zero is not
allowed", 1004);
        }

        if (double.IsInfinity(dividend / divisor))
```

```
        {
            throw new CalculatorException("Division result is infinity",
1005);
        }

        return dividend / divisor;
    }

    public double SquareRoot(double number)
    {
        if (number < 0)
        {
            throw new CalculatorException("Cannot calculate square root
of negative number", 1006);
        }

        return Math.Sqrt(number);
    }
}

// Exception hierarchy demonstration
public class MathUtilities
{
    public static void DemonstrateExceptionHierarchy()
    {
        Console.WriteLine("6. Exception Hierarchy Demonstration:");

        try
        {
            // Different types of arithmetic exceptions
            ThrowDifferentExceptions(1); // DivideByZeroException
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine($"Specific: DivideByZeroException -
{ex.Message}");
        }
        catch (ArithmeticException ex)
        {
            Console.WriteLine($"General: ArithmeticException -
```

```
{ex.Message}");
        }
        catch (SystemException ex)
        {
            Console.WriteLine($"System: SystemException - {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Base: Exception - {ex.Message}");
        }
    }

    public static void ThrowDifferentExceptions(int type)
    {
        switch (type)
        {
            case 1:
                throw new DivideByZeroException("Division by zero
occurred");
            case 2:
                throw new OverflowException("Arithmetic overflow
occurred");
            case 3:
                throw new ArithmeticException("General arithmetic
error");
            default:
                throw new InvalidOperationException("Invalid operation");
        }
    }
}
```

Common Arithmetic Exceptions:

Exception Type	Description	Common Causes
DivideByZeroException	Division by zero	`x / 0`, `x % 0`
OverflowException	Arithmetic overflow	Result exceeds data type limits
ArithmeticException	Base class for arithmetic errors	Parent of above exceptions
InvalidOperationException	Invalid operation for current state	Math operations on invalid data

0

Exception Handling Best Practices:

```
public class ExceptionBestPractices
{
    // â… GOOD: Specific exception handling
    public static int SafeDivide(int dividend, int divisor)
    {
        try
        {
            return dividend / divisor;
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Warning: Division by zero, returning 0");
            return 0;
        }
    }

    // â… GOOD: Input validation to prevent exceptions
    public static int ValidatedDivide(int dividend, int divisor)
    {
        if (divisor == 0)
        {
            throw new ArgumentException("Divisor cannot be zero",
nameof(divisor));
        }

        return dividend / divisor;
    }

    // â… GOOD: Using finally for cleanup
    public static void ProcessWithCleanup()
    {
        System.IO.FileStream stream = null;
        try
        {
            stream = new System.IO.FileStream("data.txt",
System.IO.FileMode.Create);
            // Process file
0       }
```

```
        catch (System.IO.IOException ex)
        {
            Console.WriteLine($"File error: {ex.Message}");
        }
        finally
        {
            stream?.Close(); // Always cleanup
        }
    }

    // âœ… GOOD: Using 'using' statement for automatic disposal
    public static void ProcessWithUsing()
    {
        try
        {
            using (var stream = new System.IO.FileStream("data.txt",
System.IO.FileMode.Create))
            {
                // Process file - automatic cleanup
            }
        }
        catch (System.IO.IOException ex)
        {
            Console.WriteLine($"File error: {ex.Message}");
        }
    }
}
```

Key Points:

- **Exception handling is for exceptional circumstances**, not normal control flow
- **Catch specific exceptions** before general ones
- **Use finally blocks** for cleanup code that must run
- **Consider using 'using' statements** for automatic resource disposal
- **Don't catch exceptions you can't handle meaningfully**
- **Log exceptions** for debugging and monitoring

Question 19: C# is Object Oriented Language. Give one reason supporting the given argument.

0

C# is an Object-Oriented Programming (OOP) language because it supports all four fundamental principles of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction.

Primary Reason - Encapsulation:

C# fully supports encapsulation by allowing data and methods to be bundled together within classes, while controlling access through access modifiers.

```
using System;

// Example demonstrating Encapsulation - the core OOP principle
public class BankAccount
{
    // Private fields - data is hidden from outside access
    private string accountNumber;
    private string accountHolder;
    private decimal balance;
    private DateTime createdDate;

    // Constructor to initialize object state
    public BankAccount(string accountNumber, string accountHolder,
decimal initialBalance)
    {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = initialBalance >= 0 ? initialBalance : 0;
        this.createdDate = DateTime.Now;
    }

    // Public properties provide controlled access to private data
    public string AccountNumber
    {
        get { return accountNumber; }
        // No setter - account number cannot be changed after creation
    }

    public string AccountHolder
    {
        get { return accountHolder; }
```

0

```
        set
        {
            if (!string.IsNullOrEmpty(value))
                accountHolder = value;
        }
    }

    // Read-only property for balance (can only be modified through
methods)
    public decimal Balance
    {
        get { return balance; }
    }

    public DateTime CreatedDate
    {
        get { return createdDate; }
    }

    // Public methods provide controlled operations on private data
    public bool Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
            Console.WriteLine($"Deposited: ${amount:F2}. New balance:
${balance:F2}");
            return true;
        }
        Console.WriteLine("Invalid deposit amount. Amount must be
positive.");
        return false;
    }

    public bool Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= balance)
        {
            balance -= amount;
            Console.WriteLine($"Withdrawn: ${amount:F2}. New balance:
```

```
${balance:F2}");
            return true;
        }
        Console.WriteLine("Invalid withdrawal. Check amount and
balance.");
        return false;
    }

    // Method to display account information
    public void DisplayAccountInfo()
    {
        Console.WriteLine($"Account: {accountNumber}");
        Console.WriteLine($"Holder: {accountHolder}");
        Console.WriteLine($"Balance: ${balance:F2}");
        Console.WriteLine($"Created: {createdDate:yyyy-MM-dd}");
    }
}

// Additional example showing all OOP principles
public class OOPDemonstration
{
    public static void Main()
    {
        Console.WriteLine("=== C# OOP Demonstration ===\n");

        // 1. ENCAPSULATION - Data and methods bundled together with
access control
        Console.WriteLine("1. ENCAPSULATION:");
        BankAccount account = new BankAccount("ACC001", "John Doe",
1000.00m);

        // Data is accessed through controlled methods/properties
        Console.WriteLine($"Account Number: {account.AccountNumber}"); //
Accessible
        Console.WriteLine($"Balance: ${account.Balance:F2}"); // Read-
only access

        // Direct access to private fields is not allowed:
        // account.balance = 5000; // ERROR: Cannot access private field
```

0

```
        // Operations through public methods
        account.Deposit(500);
        account.Withdraw(200);
        account.DisplayAccountInfo();

        Console.WriteLine();

        // 2. INHERITANCE - Creating specialized classes from base
classes
        Console.WriteLine("2. INHERITANCE:");
        SavingsAccount savings = new SavingsAccount("SAV001", "Jane
Smith", 2000, 0.05m);
        savings.Deposit(100);
        savings.ApplyInterest();
        savings.DisplayAccountInfo();

        Console.WriteLine();

        // 3. POLYMORPHISM - Same interface, different implementations
        Console.WriteLine("3. POLYMORPHISM:");
        BankAccount[] accounts =
        {
            new BankAccount("ACC002", "Alice Brown", 1500),
            new SavingsAccount("SAV002", "Bob Wilson", 3000, 0.03m),
            new CheckingAccount("CHK001", "Carol Davis", 800, 500)
        };

        foreach (BankAccount acc in accounts)
        {
            Console.WriteLine($"Processing {acc.GetType().Name}:");
            acc.DisplayAccountInfo(); // Polymorphic behavior
            Console.WriteLine();
        }

        // 4. ABSTRACTION - Hiding complex implementation details
        Console.WriteLine("4. ABSTRACTION:");
        IPaymentProcessor processor = new CreditCardProcessor();
        processor.ProcessPayment(250.00m); // Abstract interface,
concrete implementation
    }
0
```

```
}

// INHERITANCE example
public class SavingsAccount : BankAccount
{
    private decimal interestRate;

    public SavingsAccount(string accountNumber, string accountHolder,
decimal initialBalance, decimal interestRate)
        : base(accountNumber, accountHolder, initialBalance)
    {
        this.interestRate = interestRate;
    }

    public decimal InterestRate
    {
        get { return interestRate; }
        set { interestRate = value >= 0 ? value : 0; }
    }

    public void ApplyInterest()
    {
        decimal interest = Balance * interestRate;
        Deposit(interest);
        Console.WriteLine($"Interest applied: ${interest:F2} at rate
{interestRate:P2}");
    }

    public override void DisplayAccountInfo()
    {
        base.DisplayAccountInfo();
        Console.WriteLine($"Interest Rate: {interestRate:P2}");
    }
}

public class CheckingAccount : BankAccount
{
    private decimal overdraftLimit;

    public CheckingAccount(string accountNumber, string accountHolder,
```

```
decimal initialBalance, decimal overdraftLimit)
        : base(accountNumber, accountHolder, initialBalance)
    {
        this.overdraftLimit = overdraftLimit;
    }

    public decimal OverdraftLimit
    {
        get { return overdraftLimit; }
        set { overdraftLimit = value >= 0 ? value : 0; }
    }

    // Override withdraw method to allow overdraft
    public new bool Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= (Balance + overdraftLimit))
        {
            // Use reflection or internal access to modify balance
            // For simplicity, using base withdrawal with validation
            if (amount <= Balance)
            {
                return base.Withdraw(amount);
            }
            else
            {
                decimal overdraftAmount = amount - Balance;
                Console.WriteLine($"Using overdraft:
${overdraftAmount:F2}");
                // In real implementation, would modify balance directly
                return true;
            }
        }
        Console.WriteLine("Withdrawal exceeds account balance plus
overdraft limit.");
        return false;
    }

    public override void DisplayAccountInfo()
    {
        base.DisplayAccountInfo();
```

0

```
        Console.WriteLine($"Overdraft Limit: ${overdraftLimit:F2}");
    }
}

// ABSTRACTION example with interfaces
public interface IPaymentProcessor
{
    bool ProcessPayment(decimal amount);
    string GetTransactionId();
}

public class CreditCardProcessor : IPaymentProcessor
{
    private Random random = new Random();

    public bool ProcessPayment(decimal amount)
    {
        // Complex internal logic hidden from user
        Console.WriteLine($"Processing credit card payment of
${amount:F2}");

        // Simulate processing
        System.Threading.Thread.Sleep(1000);

        // Simulate success/failure
        bool success = random.Next(1, 10) > 2; // 80% success rate

        if (success)
        {
            Console.WriteLine($"Payment successful. Transaction ID:
{GetTransactionId()}");
        }
        else
        {
            Console.WriteLine("Payment failed. Please try again.");
        }

        return success;
    }
```

0

```
    public string GetTransactionId()
    {
        return $"TXN{random.Next(100000, 999999)}";
    }
}
```

Why This Proves C# is Object-Oriented:

1. **Classes and Objects**: C# allows creating custom data types (classes) and instances (objects)
2. **Encapsulation**: Private fields with public methods/properties control data access
3. **Data Hiding**: Implementation details are hidden from external code
4. **Method Binding**: Data and methods that operate on that data are bundled together
5. **Access Control**: Access modifiers (private, public, protected) enforce proper encapsulation

This example shows that **C# treats everything as objects** (except primitive types, which are also represented as objects through boxing), and **enforces proper data encapsulation through access modifiers**, which is the fundamental principle of Object-Oriented Programming.

Question 20: Explain Garbage collection.

What is Garbage Collection?

Garbage Collection (GC) in C# is an automatic memory management feature that automatically frees memory occupied by objects that are no longer reachable or referenced by the application. It's handled by the .NET runtime's Garbage Collector.

How Garbage Collection Works:

```
using System;
using System.Collections.Generic;

public class GarbageCollectionDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Garbage Collection Demonstration ===\n");
```

0

```
        // Show initial memory state
        ShowMemoryInfo("Initial state");

        // Create objects that will become eligible for GC
        CreateObjectsForGC();

        // Force garbage collection
        Console.WriteLine("Forcing garbage collection...");
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        ShowMemoryInfo("After forced GC");

        // Demonstrate different generations
        DemonstrateGenerations();

        // Demonstrate finalizers
        DemonstrateFinalization();

        // Demonstrate weak references
        DemonstrateWeakReferences();
    }

    public static void CreateObjectsForGC()
    {
        Console.WriteLine("Creating objects that will become
garbage...");

        // Create many objects in local scope
        for (int i = 0; i < 100000; i++)
        {
            var tempObject = new TempClass($"Object {i}");
            var tempList = new List<int> { 1, 2, 3, 4, 5 };
            var tempString = $"Temporary string {i}";
        }
        // Objects go out of scope here and become eligible for GC

        ShowMemoryInfo("After creating temporary objects");
    }
0
```

```
    public static void ShowMemoryInfo(string label)
    {
        long memoryBefore = GC.GetTotalMemory(false);

        Console.WriteLine($"\n--- {label} ---");
        Console.WriteLine($"Total Memory: {memoryBefore:N0} bytes");
        Console.WriteLine($"Gen 0 Collections: {GC.CollectionCount(0)}");
        Console.WriteLine($"Gen 1 Collections: {GC.CollectionCount(1)}");
        Console.WriteLine($"Gen 2 Collections: {GC.CollectionCount(2)}");
        Console.WriteLine($"Max Generation: {GC.MaxGeneration}");
    }

    public static void DemonstrateGenerations()
    {
        Console.WriteLine("\n=== Generation Demonstration ===");

        // Create objects of different lifespans
        var shortLived = new TempClass("Short-lived");
        var mediumLived = new TempClass("Medium-lived");
        var longLived = new TempClass("Long-lived");

        // Check generations
        Console.WriteLine($"Short-lived generation:
{GC.GetGeneration(shortLived)}");
        Console.WriteLine($"Medium-lived generation:
{GC.GetGeneration(mediumLived)}");
        Console.WriteLine($"Long-lived generation:
{GC.GetGeneration(longLived)}");

        // Create pressure to trigger Gen 0 collection
        for (int i = 0; i < 10000; i++)
        {
            var temp = new TempClass($"Pressure {i}");
        }

        Console.WriteLine($"\nAfter memory pressure:");
        Console.WriteLine($"Medium-lived generation:
{GC.GetGeneration(mediumLived)}");
        Console.WriteLine($"Long-lived generation:
0
```

```
{GC.GetGeneration(longLived)}");

        // Keep references to prevent collection
        GC.KeepAlive(mediumLived);
        GC.KeepAlive(longLived);
    }

    public static void DemonstrateFinalization()
    {
        Console.WriteLine("\n=== Finalization Demonstration ===");

        // Create objects with finalizers
        for (int i = 0; i < 5; i++)
        {
            var finalizableObject = new FinalizableClass(i);
        }

        Console.WriteLine("Objects with finalizers created");

        // Force collection and finalization
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        Console.WriteLine("Finalization completed");
    }

    public static void DemonstrateWeakReferences()
    {
        Console.WriteLine("\n=== Weak References Demonstration ===");

        // Create object and weak reference
        var strongRef = new TempClass("Strong Reference");
        var weakRef = new WeakReference(strongRef);

        Console.WriteLine($"Strong ref alive: {strongRef != null}");
        Console.WriteLine($"Weak ref alive: {weakRef.IsAlive}");
        Console.WriteLine($"Weak ref target: {weakRef.Target}");

        // Remove strong reference
```

```
        strongRef = null;

        Console.WriteLine("\nAfter removing strong reference:");
        Console.WriteLine($"Weak ref alive (before GC):
{weakRef.IsAlive}");

        // Force garbage collection
        GC.Collect();
        GC.WaitForPendingFinalizers();

        Console.WriteLine($"Weak ref alive (after GC):
{weakRef.IsAlive}");
        Console.WriteLine($"Weak ref target: {weakRef.Target}");
    }
}

// Example class for GC demonstration
public class TempClass
{
    public string Name { get; set; }
    public DateTime CreatedAt { get; set; }
    private byte[] data; // Some data to make object larger

    public TempClass(string name)
    {
        Name = name;
        CreatedAt = DateTime.Now;
        data = new byte[1024]; // 1KB of data
    }

    public override string ToString()
    {
        return $"TempClass: {Name} (Created: {CreatedAt:HH:mm:ss})";
    }
}

// Class with finalizer (destructor)
public class FinalizableClass : IDisposable
{
    private int id;
```

```
    private bool disposed = false;

    public FinalizableClass(int id)
    {
        this.id = id;
        Console.WriteLine($"FinalizableClass {id} created");
    }

    // Finalizer (destructor) - called by GC
    ~FinalizableClass()
    {
        Console.WriteLine($"FinalizableClass {id} finalized");
        Dispose(false);
    }

    // IDisposable implementation
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this); // Don't call finalizer if disposed
manually
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources
                Console.WriteLine($"FinalizableClass {id} disposing
managed resources");
            }

            // Dispose unmanaged resources
            Console.WriteLine($"FinalizableClass {id} disposing unmanaged
resources");
            disposed = true;
        }
    }
0
```

```
}

// Advanced GC concepts demonstration
public class AdvancedGCDemo
{
    private static List<object> keepAlive = new List<object>();

    public static void DemonstrateGCPressure()
    {
        Console.WriteLine("\n=== GC Pressure Demonstration ===");

        // Create memory pressure
        for (int i = 0; i < 1000; i++)
        {
            var largeObject = new byte[85000]; // Large Object Heap (LOH)
            if (i % 100 == 0)
            {
                keepAlive.Add(largeObject); // Keep some alive
            }
        }

        ShowMemoryInfo("After creating large objects");

        // Clear references
        keepAlive.Clear();

        GC.Collect();
        GC.WaitForPendingFinalizers();

        ShowMemoryInfo("After clearing references and GC");
    }

    public static void DemonstrateGCNotifications()
    {
        Console.WriteLine("\n=== GC Notifications ===");

        // Register for GC notifications
        GC.RegisterForFullGCNotification(10, 10);

        // Create memory pressure to trigger GC
```

```
        for (int i = 0; i < 50000; i++)
        {
            var temp = new byte[1024];
        }

        // Check for GC notification
        var status = GC.WaitForFullGCApproach(1000);
        if (status == GCNotificationStatus.Succeeded)
        {
            Console.WriteLine("Full GC is approaching");
        }
    }

    private static void ShowMemoryInfo(string label)
    {
        Console.WriteLine($"\n{label}:");
        Console.WriteLine($"Total Memory: {GC.GetTotalMemory(false):N0}
bytes");
        Console.WriteLine($"Collections - Gen0: {GC.CollectionCount(0)},
Gen1: {GC.CollectionCount(1)}, Gen2: {GC.CollectionCount(2)}");
    }
}

// Resource management best practices
public class ProperResourceManagement : IDisposable
{
    private System.IO.FileStream fileStream;
    private bool disposed = false;

    public ProperResourceManagement(string filename)
    {
        fileStream = new System.IO.FileStream(filename,
System.IO.FileMode.Create);
    }

    public void WriteData(string data)
    {
        if (disposed) throw new
ObjectDisposedException(nameof(ProperResourceManagement));
```

0

```
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(data);
        fileStream.Write(bytes, 0, bytes.Length);
    }

    // Proper dispose pattern
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources
                fileStream?.Dispose();
            }

            disposed = true;
        }
    }

    // Finalizer as safety net
    ~ProperResourceManagement()
    {
        Dispose(false);
    }
}
```

Garbage Collection Process:

Phase	Description	Action
Mark	Identify reachable objects	GC walks object graph from roots
Sweep	Free unreachable objects	Deallocate memory of unmarked objects
Compact	Defragment heap	Move objects to eliminate fragmentation

0

Generation-Based Collection:

```
public class GenerationDemo
{
    public static void ExplainGenerations()
    {
        Console.WriteLine("=== Generation-Based Collection ===");

        /*
         * Generation 0: New objects, collected frequently
         * Generation 1: Objects that survived one GC, collected less
frequently
         * Generation 2: Long-lived objects, collected rarely
         * Large Object Heap (LOH): Objects >= 85KB, collected with Gen 2
         */

        // Gen 0 objects (short-lived)
        for (int i = 0; i < 1000; i++)
        {
            var temp = new StringBuilder($"Temp {i}");
            // These become eligible for collection immediately
        }

        // Gen 1/2 objects (longer-lived)
        var persistent = new List<string>();
        for (int i = 0; i < 100; i++)
        {
            persistent.Add($"Persistent {i}");
        }

        Console.WriteLine($"List generation:
{GC.GetGeneration(persistent)}");

        // Trigger collections to promote objects
        GC.Collect(0); // Collect Gen 0 only
        GC.Collect(1); // Collect Gen 0 and 1
        GC.Collect(2); // Full collection (all generations)

        Console.WriteLine($"List generation after GC:
{GC.GetGeneration(persistent)}");
```

```
    }
}
```

Key Points about Garbage Collection:

1. **Automatic**: No manual memory deallocation needed
2. **Generational**: Different generations collected at different frequencies
3. **Mark and Sweep**: Identifies and frees unreachable objects
4. **Compacting**: Reduces heap fragmentation
5. **Concurrent**: Can run concurrently with application (in some modes)
6. **Tunable**: Various GC modes available (Workstation, Server, Concurrent, etc.)

Best Practices:

- **Implement IDisposable** for resources that need deterministic cleanup
- **Use `using` statements** for automatic disposal
- **Avoid unnecessary object creation** in tight loops
- **Be careful with event handlers** - they can prevent garbage collection
- **Use weak references** when appropriate to avoid memory leaks
- **Don't call GC.Collect() manually** unless absolutely necessary

Question 21: What are the roles of C# Compiler and JIT compiler?

C# Compiler vs JIT Compiler

C# compilation involves a two-step process: **C# Compiler** compiles source code to Intermediate Language (IL), and **JIT Compiler** compiles IL to native machine code at runtime.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class CompilerDemo
{
    public static void Main()
    {
        Console.WriteLine("=== C# Compiler vs JIT Compiler Demonstration
===\n");

        // Demonstrate compilation process
0
```

```
        DemonstrateCompilationProcess();

        // Show JIT compilation in action
        DemonstrateJITCompilation();

        // Show runtime code generation
        DemonstrateRuntimeCodeGeneration();

        // Performance comparison
        DemonstrateJITPerformance();
    }

    public static void DemonstrateCompilationProcess()
    {
        Console.WriteLine("1. COMPILATION PROCESS:");
        Console.WriteLine("   Source Code (.cs) -> C# Compiler -> IL Code
(.dll/.exe) -> JIT Compiler -> Native Code");
        Console.WriteLine();

        // Get information about current assembly
        Assembly currentAssembly = Assembly.GetExecutingAssembly();
        Console.WriteLine($"Assembly: {currentAssembly.GetName().Name}");
        Console.WriteLine($"Location: {currentAssembly.Location}");
        Console.WriteLine($"Runtime Version:
{currentAssembly.ImageRuntimeVersion}");
        Console.WriteLine();
    }

    public static void DemonstrateJITCompilation()
    {
        Console.WriteLine("2. JIT COMPILATION DEMONSTRATION:");

        // First call - method gets JIT compiled
        Console.WriteLine("First call to SampleMethod (JIT compilation
occurs):");
        var start = DateTime.UtcNow;
        int result1 = SampleMethod(10, 20);
        var duration1 = DateTime.UtcNow - start;
        Console.WriteLine($"Result: {result1}, Duration:
{duration1.TotalMilliseconds:F4} ms");
```

```
        // Second call - method already compiled
        Console.WriteLine("Second call to SampleMethod (already JIT
compiled):");
        start = DateTime.UtcNow;
        int result2 = SampleMethod(30, 40);
        var duration2 = DateTime.UtcNow - start;
        Console.WriteLine($"Result: {result2}, Duration:
{duration2.TotalMilliseconds:F4} ms");

        Console.WriteLine($"Performance improvement:
{(duration1.TotalMilliseconds / duration2.TotalMilliseconds):F2}x
faster");
        Console.WriteLine();
    }

    // Sample method for JIT demonstration
    public static int SampleMethod(int a, int b)
    {
        // Simulate some computation
        int result = 0;
        for (int i = 0; i < 1000; i++)
        {
            result += (a * b) + (i % 10);
        }
        return result;
    }

    public static void DemonstrateRuntimeCodeGeneration()
    {
        Console.WriteLine("3. RUNTIME CODE GENERATION
(Reflection.Emit):");

        // Create dynamic assembly
        AssemblyName assemblyName = new AssemblyName("DynamicAssembly");
        AssemblyBuilder assemblyBuilder =
AssemblyBuilder.DefineDynamicAssembly(
            assemblyName, AssemblyBuilderAccess.Run);

        // Create dynamic module
```

```
        ModuleBuilder moduleBuilder =
assemblyBuilder.DefineDynamicModule("DynamicModule");

        // Create dynamic type
        TypeBuilder typeBuilder =
moduleBuilder.DefineType("DynamicCalculator",
            TypeAttributes.Public);

        // Create dynamic method
        MethodBuilder methodBuilder = typeBuilder.DefineMethod("Add",
            MethodAttributes.Public | MethodAttributes.Static,
            typeof(int), new Type[] { typeof(int), typeof(int) });

        // Generate IL code
        ILGenerator il = methodBuilder.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0);  // Load first argument
        il.Emit(OpCodes.Ldarg_1);  // Load second argument
        il.Emit(OpCodes.Add);      // Add them
        il.Emit(OpCodes.Ret);      // Return result

        // Create type and get method
        Type dynamicType = typeBuilder.CreateType();
        MethodInfo dynamicMethod = dynamicType.GetMethod("Add");

        // Invoke dynamically generated method
        object result = dynamicMethod.Invoke(null, new object[] { 15, 25
});
        Console.WriteLine($"Dynamic method result: 15 + 25 = {result}");
        Console.WriteLine();
    }

    public static void DemonstrateJITPerformance()
    {
        Console.WriteLine("4. JIT PERFORMANCE IMPACT:");

        const int iterations = 1000000;

        // Warm up JIT
        for (int i = 0; i < 1000; i++)
        {
0
```

```
            MathOperations.ComplexCalculation(i);
        }

        // Measure performance after JIT compilation
        var stopwatch = System.Diagnostics.Stopwatch.StartNew();

        for (int i = 0; i < iterations; i++)
        {
            MathOperations.ComplexCalculation(i);
        }

        stopwatch.Stop();

        Console.WriteLine($"Executed {iterations:N0} operations in
{stopwatch.ElapsedMilliseconds} ms");
        Console.WriteLine($"Average per operation:
{(double)stopwatch.ElapsedTicks / iterations:F2} ticks");
        Console.WriteLine();
    }
}

// Class for performance testing
public static class MathOperations
{
    public static double ComplexCalculation(int input)
    {
        double result = Math.Sqrt(input);
        result = Math.Sin(result) + Math.Cos(result);
        result *= Math.PI;
        return Math.Abs(result);
    }
}

// IL Code inspection utility
public class ILInspection
{
    public static void InspectMethodIL()
    {
        Console.WriteLine("5. IL CODE INSPECTION:");
```

0

```
        MethodInfo method =
typeof(CompilerDemo).GetMethod("SampleMethod");
        MethodBody methodBody = method.GetMethodBody();

        if (methodBody != null)
        {
            Console.WriteLine($"Method: {method.Name}");
            Console.WriteLine($"IL Code Size:
{methodBody.GetILAsByteArray().Length} bytes");
            Console.WriteLine($"Max Stack Size:
{methodBody.MaxStackSize}");
            Console.WriteLine($"Local Variables:
{methodBody.LocalVariables.Count}");

            foreach (var localVar in methodBody.LocalVariables)
            {
                Console.WriteLine($"  Local {localVar.LocalIndex}:
{localVar.LocalType}");
            }
        }
        Console.WriteLine();
    }
}

// Custom JIT optimization demonstration
public class JITOptimizationDemo
{
    public static void DemonstrateOptimizations()
    {
        Console.WriteLine("6. JIT OPTIMIZATIONS:");

        // Method inlining demonstration
        Console.WriteLine("Testing method inlining...");

        var stopwatch = System.Diagnostics.Stopwatch.StartNew();

        for (int i = 0; i < 10000000; i++)
        {
            int result = AddNumbers(i, i + 1); // Small method likely to
be inlined
```

```
        }

        stopwatch.Stop();
        Console.WriteLine($"Method calls completed in
{stopwatch.ElapsedMilliseconds} ms");

        // Loop optimization
        Console.WriteLine("Testing loop optimization...");

        stopwatch.Restart();

        int sum = 0;
        for (int i = 0; i < 10000000; i++)
        {
            sum += i; // JIT will optimize this loop
        }

        stopwatch.Stop();
        Console.WriteLine($"Loop optimization completed in
{stopwatch.ElapsedMilliseconds} ms");
        Console.WriteLine($"Sum: {sum}");
        Console.WriteLine();
    }

    // Small method that may be inlined by JIT
    private static int AddNumbers(int a, int b)
    {
        return a + b;
    }
}
```

Detailed Comparison:

Aspect	C# Compiler (csc.exe/Roslyn)	JIT Compiler
Input	C# source code (.cs files)	IL code (.dll/.exe files)
Output	Intermediate Language (IL) code	Native machine code
When	Compile time (before execution)	Runtime (during execution)
Platform	Platform independent	Platform specific
Optimization	High-level optimizations	Low-level, runtime optimizations

0

Aspect	C# Compiler (csc.exe/Roslyn)	JIT Compiler
Speed	Slower (full analysis)	Faster (targeted compilation)

C# Compiler Role:

```
// Example showing what C# compiler does:

// 1. SYNTAX ANALYSIS
public class CompilerTasks
{
    // Compiler checks syntax, semantics, types
    public void CompilerResponsibilities()
    {
        // Syntax validation
        int number = 42; // â“ Valid syntax
        // int number = ; // â— Compiler error: syntax error

        // Type checking
        string text = "Hello"; // â“ Valid type assignment
        // int wrongType = "Hello"; // â— Compiler error: type mismatch

        // Method resolution
        Console.WriteLine(text); // â“ Compiler finds correct overload

        // Constant folding
        const int result = 10 + 20; // Compiler calculates at compile
time
    }
}

// 2. IL CODE GENERATION
/*
Original C# code:
public int Add(int a, int b)
{
    return a + b;
}

Generated IL code (simplified):
```

0

```
.method public hidebysig instance int32 Add(int32 a, int32 b)
{
    .maxstack 2
    ldarg.1     // Load argument 'a'
    ldarg.2     // Load argument 'b'
    add         // Add them
    ret         // Return result
}
*/
```

JIT Compiler Role:

```
public class JITCompilerTasks
{
    public static void JITResponsibilities()
    {
        Console.WriteLine("JIT Compiler Tasks:");
        Console.WriteLine("1. Convert IL to native machine code");
        Console.WriteLine("2. Perform runtime optimizations");
        Console.WriteLine("3. Handle platform-specific details");
        Console.WriteLine("4. Manage method compilation on-demand");
        Console.WriteLine("5. Apply processor-specific optimizations");

        // JIT compilation happens here (first call)
        PerformanceTestMethod();

        // Subsequent calls use already compiled native code
        PerformanceTestMethod();
    }

    private static void PerformanceTestMethod()
    {
        // JIT will optimize this based on:
        // - CPU architecture (x86, x64, ARM)
        // - Available CPU features (SSE, AVX)
        // - Runtime profiling data

        for (int i = 0; i < 1000; i++)
        {
0
```

```
            double result = Math.Sqrt(i) * Math.PI;
        }
    }
}
```

JIT Optimization Examples:

```
public class JITOptimizations
{
    // 1. METHOD INLINING
    public static void InliningDemo()
    {
        // Small methods like GetConstant() may be inlined
        int value = GetConstant() * 2;
        Console.WriteLine(value);
    }

    private static int GetConstant() => 42; // Likely to be inlined

    // 2. DEAD CODE ELIMINATION
    public static void DeadCodeDemo()
    {
        bool condition = false; // JIT knows this is always false

        if (condition) // JIT will eliminate this entire block
        {
            Console.WriteLine("This code will be eliminated");
            ExpensiveOperation();
        }

        Console.WriteLine("This code remains");
    }

    // 3. LOOP OPTIMIZATION
    public static void LoopOptimizationDemo()
    {
        int[] array = new int[1000];

        // JIT will optimize bounds checking and loop structure
```

0

```
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i * 2; // Bounds check may be eliminated
        }
    }

    // 4. REGISTER ALLOCATION
    public static int RegisterDemo(int a, int b, int c)
    {
        // JIT will efficiently allocate CPU registers for variables
        int temp1 = a + b;
        int temp2 = temp1 * c;
        int result = temp2 + a;
        return result;
    }

    private static void ExpensiveOperation()
    {
        System.Threading.Thread.Sleep(1000);
    }
}
```

Key Differences Summary:

C# Compiler (Compile-time):

- Converts source code to IL
- Performs syntax and semantic analysis
- Type checking and method resolution
- High-level optimizations
- Generates metadata
- Platform independent output

JIT Compiler (Runtime):

- Converts IL to native code
- Just-in-time compilation (on first method call)
- Platform-specific optimizations
- Runtime profiling and adaptive optimization
- Processor-specific code generation
- Memory layout optimization

0

This two-stage compilation allows C# to be both platform-independent (IL) and highly optimized (native code).

Question 22: Define an interface with methods Area(), Volume(). Define a constant PI having value 3.14. Create class Cylinder which implements this interface. Create one object and calculate area and volume.

```
using System;

// Interface definition with methods and constant
public interface IShape3D
{
    // Constant PI with value 3.14
    const double PI = 3.14;

    // Abstract methods that implementing classes must define
    double Area();
    double Volume();

    // Optional: Additional interface members
    string GetShapeInfo();
}

// Cylinder class implementing the interface
public class Cylinder : IShape3D
{
    // Private fields
    private double radius;
    private double height;

    // Constructor
    public Cylinder(double radius, double height)
    {
        if (radius <= 0 || height <= 0)
            throw new ArgumentException("Radius and height must be
positive");

        this.radius = radius;
        this.height = height;
```

0

```
    }

    // Properties
    public double Radius
    {
        get { return radius; }
        set
        {
            if (value > 0)
                radius = value;
            else
                throw new ArgumentException("Radius must be positive");
        }
    }

    public double Height
    {
        get { return height; }
        set
        {
            if (value > 0)
                height = value;
            else
                throw new ArgumentException("Height must be positive");
        }
    }

    // Implementation of Area() method - Surface Area of Cylinder
    public double Area()
    {
        // Surface Area = 2Ï€rÂ² + 2Ï€rh (top + bottom + lateral surface)
        double topAndBottom = 2 * IShape3D.PI * radius * radius;
        double lateralSurface = 2 * IShape3D.PI * radius * height;
        return topAndBottom + lateralSurface;
    }

    // Implementation of Volume() method
    public double Volume()
    {
        // Volume = Ï€rÂ²h
```

```
        return IShape3D.PI * radius * radius * height;
    }

    // Implementation of GetShapeInfo() method
    public string GetShapeInfo()
    {
        return $"Cylinder - Radius: {radius:F2}, Height: {height:F2}";
    }

    // Additional methods specific to Cylinder
    public double GetLateralArea()
    {
        // Lateral Area = 2Ï€rh
        return 2 * IShape3D.PI * radius * height;
    }

    public double GetBaseArea()
    {
        // Base Area = Ï€rÂ²
        return IShape3D.PI * radius * radius;
    }

    public void DisplayDetails()
    {
        Console.WriteLine($"=== Cylinder Details ===");
        Console.WriteLine($"Radius: {radius:F2}");
        Console.WriteLine($"Height: {height:F2}");
        Console.WriteLine($"Base Area: {GetBaseArea():F2}");
        Console.WriteLine($"Lateral Area: {GetLateralArea():F2}");
        Console.WriteLine($"Total Surface Area: {Area():F2}");
        Console.WriteLine($"Volume: {Volume():F2}");
        Console.WriteLine($"========================");
    }
}

// Additional shape classes implementing the same interface
public class Sphere : IShape3D
{
    private double radius;
```

0

```
    public Sphere(double radius)
    {
        if (radius <= 0)
            throw new ArgumentException("Radius must be positive");
        this.radius = radius;
    }

    public double Radius
    {
        get { return radius; }
        set
        {
            if (value > 0)
                radius = value;
            else
                throw new ArgumentException("Radius must be positive");
        }
    }

    public double Area()
    {
        // Surface Area of Sphere = 4Ï€rÂ²
        return 4 * IShape3D.PI * radius * radius;
    }

    public double Volume()
    {
        // Volume of Sphere = (4/3)Ï€rÂ³
        return (4.0 / 3.0) * IShape3D.PI * radius * radius * radius;
    }

    public string GetShapeInfo()
    {
        return $"Sphere - Radius: {radius:F2}";
    }
}

public class Cone : IShape3D
{
    private double radius;
```

```
    private double height;

    public Cone(double radius, double height)
    {
        if (radius <= 0 || height <= 0)
            throw new ArgumentException("Radius and height must be
positive");
        this.radius = radius;
        this.height = height;
    }

    public double Radius => radius;
    public double Height => height;

    // Slant height calculation
    public double SlantHeight => Math.Sqrt(radius * radius + height *
height);

    public double Area()
    {
        // Surface Area = Ï€rÂ² + Ï€rl (base + lateral surface)
        double baseArea = IShape3D.PI * radius * radius;
        double lateralArea = IShape3D.PI * radius * SlantHeight;
        return baseArea + lateralArea;
    }

    public double Volume()
    {
        // Volume = (1/3)Ï€rÂ²h
        return (1.0 / 3.0) * IShape3D.PI * radius * radius * height;
    }

    public string GetShapeInfo()
    {
        return $"Cone - Radius: {radius:F2}, Height: {height:F2}, Slant
Height: {SlantHeight:F2}";
    }
}

// Main program demonstrating interface implementation
```

```
public class InterfaceDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Interface Implementation Demonstration
===\n");

        try
        {
            // Create Cylinder object and calculate area and volume
            CreateAndTestCylinder();

            // Demonstrate polymorphism with interface
            DemonstratePolymorphism();

            // Interactive cylinder creation
            CreateInteractiveCylinder();

            // Compare different shapes
            CompareShapes();
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Unexpected error: {ex.Message}");
        }
    }

    public static void CreateAndTestCylinder()
    {
        Console.WriteLine("1. Creating and Testing Cylinder:");

        // Create cylinder object
        Cylinder cylinder = new Cylinder(5.0, 10.0);

        // Calculate and display area and volume
        double area = cylinder.Area();
```

```
        double volume = cylinder.Volume();

        Console.WriteLine($"Cylinder: Radius = {cylinder.Radius}, Height
= {cylinder.Height}");
        Console.WriteLine($"Surface Area = {area:F2}");
        Console.WriteLine($"Volume = {volume:F2}");
        Console.WriteLine($"Shape Info: {cylinder.GetShapeInfo()}");

        // Display detailed information
        cylinder.DisplayDetails();

        Console.WriteLine();
    }

    public static void DemonstratePolymorphism()
    {
        Console.WriteLine("2. Polymorphism with Interface:");

        // Array of interface references
        IShape3D[] shapes = {
            new Cylinder(3, 8),
            new Sphere(4),
            new Cone(5, 12),
            new Cylinder(2.5, 6)
        };

        Console.WriteLine($"{"Shape",-20} {"Area",-15} {"Volume",-15}");
        Console.WriteLine(new string('-', 50));

        foreach (IShape3D shape in shapes)
        {
            Console.WriteLine($"{shape.GetShapeInfo(),-20}
{shape.Area(),-15:F2} {shape.Volume(),-15:F2}");
        }

        Console.WriteLine();
    }

    public static void CreateInteractiveCylinder()
    {
0
```

```
        Console.WriteLine("3. Interactive Cylinder Creation:");

        try
        {
            Console.Write("Enter cylinder radius: ");
            double radius = double.Parse(Console.ReadLine());

            Console.Write("Enter cylinder height: ");
            double height = double.Parse(Console.ReadLine());

            Cylinder userCylinder = new Cylinder(radius, height);

            Console.WriteLine("\nYour cylinder:");
            userCylinder.DisplayDetails();

            // Calculate some additional properties
            double baseArea = userCylinder.GetBaseArea();
            double lateralArea = userCylinder.GetLateralArea();

            Console.WriteLine($"Additional calculations:");
            Console.WriteLine($"Base Area: {baseArea:F2}");
            Console.WriteLine($"Lateral Area: {lateralArea:F2}");
            Console.WriteLine($"Total Surface Area:
{userCylinder.Area():F2}");
            Console.WriteLine($"Volume: {userCylinder.Volume():F2}");
        }
        catch (FormatException)
        {
            Console.WriteLine("Invalid input. Please enter numeric
values.");
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Invalid values: {ex.Message}");
        }

        Console.WriteLine();
    }

    public static void CompareShapes()
0
```

```
    {
        Console.WriteLine("4. Shape Comparison:");

        // Create shapes with same dimensions where applicable
        double commonRadius = 3.0;
        double commonHeight = 6.0;

        IShape3D cylinder = new Cylinder(commonRadius, commonHeight);
        IShape3D sphere = new Sphere(commonRadius);
        IShape3D cone = new Cone(commonRadius, commonHeight);

        Console.WriteLine("Comparing shapes with radius = 3.0:");
        Console.WriteLine($"Cylinder (h=6.0): Area =
{cylinder.Area():F2}, Volume = {cylinder.Volume():F2}");
        Console.WriteLine($"Sphere: Area = {sphere.Area():F2}, Volume =
{sphere.Volume():F2}");
        Console.WriteLine($"Cone (h=6.0): Area = {cone.Area():F2}, Volume
= {cone.Volume():F2}");

        // Find shape with maximum volume
        IShape3D[] compareShapes = { cylinder, sphere, cone };
        IShape3D maxVolumeShape = null;
        double maxVolume = 0;

        foreach (IShape3D shape in compareShapes)
        {
            if (shape.Volume() > maxVolume)
            {
                maxVolume = shape.Volume();
                maxVolumeShape = shape;
            }
        }

        Console.WriteLine($"\nShape with maximum volume:
{maxVolumeShape?.GetShapeInfo()} (Volume: {maxVolume:F2})");

        Console.WriteLine();
    }
}
```

0

```
// Utility class for shape calculations
public static class ShapeUtilities
{
    // Extension methods for interface
    public static double GetDensity(this IShape3D shape, double mass)
    {
        return mass / shape.Volume();
    }

    public static double GetSurfaceToVolumeRatio(this IShape3D shape)
    {
        return shape.Area() / shape.Volume();
    }

    // Helper method to convert degrees to radians
    public static double DegreesToRadians(double degrees)
    {
        return degrees * IShape3D.PI / 180.0;
    }

    // Calculate volume using different PI precision
    public static void ComparePrecision(IShape3D shape)
    {
        // Using interface constant PI = 3.14
        double volumeInterfacePI = shape.Volume();

        // Using Math.PI for comparison
        double precisePIRatio = Math.PI / IShape3D.PI;
        double volumeMathPI = volumeInterfacePI * precisePIRatio;

        Console.WriteLine($"Volume with PI = 3.14:
{volumeInterfacePI:F6}");
        Console.WriteLine($"Volume with Math.PI: {volumeMathPI:F6}");
        Console.WriteLine($"Difference: {Math.Abs(volumeMathPI -
volumeInterfacePI):F6}");
    }
}

// Advanced interface example with generic constraints
public interface IComparable3DShape : IShape3D,
```

0

```
IComparable<IComparable3DShape>
{
    double GetCharacteristicLength();
}

public class AdvancedCylinder : Cylinder, IComparable3DShape
{
    public AdvancedCylinder(double radius, double height) : base(radius,
height)
    {
    }

    public double GetCharacteristicLength()
    {
        // Return the larger of diameter or height
        return Math.Max(2 * Radius, Height);
    }

    public int CompareTo(IComparable3DShape other)
    {
        if (other == null) return 1;
        return this.Volume().CompareTo(other.Volume());
    }
}
```

Key Interface Concepts Demonstrated:

1. **Interface Definition**: `IShape3D` with methods `Area()`, `Volume()`, and constant `PI`
2. **Interface Implementation**: `Cylinder` class implements all interface members
3. **Constant Usage**: Interface constant `PI = 3.14` used in calculations
4. **Polymorphism**: Different shapes can be treated uniformly through interface
5. **Multiple Implementations**: Several classes implement the same interface

Output Example:

```
Cylinder: Radius = 5, Height = 10
Surface Area = 471.00
Volume = 785.00
Shape Info: Cylinder - Radius: 5.00, Height: 10.00
```

0

This demonstrates how interfaces provide a contract that implementing classes must follow, enabling polymorphism and code reusability.

Question 23: What are the streams in C#? Write a program to save content to file and read from it using stream.

What are Streams?

Streams in C# represent a sequence of bytes that can be read from or written to. They provide a unified way to work with different data sources like files, memory, network connections, etc.

Stream Hierarchy and Types:

```
using System;
using System.IO;
using System.Text;

public class StreamDemo
{
    public static void Main()
    {
        Console.WriteLine("=== C# Streams Demonstration ===\n");

        // Basic file stream operations
        BasicFileStreamDemo();

        // StreamWriter and StreamReader
        StreamWriterReaderDemo();

        // Binary streams
        BinaryStreamDemo();

        // Memory streams
        MemoryStreamDemo();

        // Buffered streams
        BufferedStreamDemo();

        // Advanced stream operations
        AdvancedStreamOperations();
```

```
    }

    public static void BasicFileStreamDemo()
    {
        Console.WriteLine("1. Basic FileStream Operations:");

        string fileName = "basic_stream_demo.txt";

        try
        {
            // WRITING to file using FileStream
            using (FileStream writeStream = new FileStream(fileName,
FileMode.Create, FileAccess.Write))
            {
                string content = "Hello, World! This is written using
FileStream.";
                byte[] data = Encoding.UTF8.GetBytes(content);

                writeStream.Write(data, 0, data.Length);
                Console.WriteLine($"âœ“ Written {data.Length} bytes to
{fileName}");
            }

            // READING from file using FileStream
            using (FileStream readStream = new FileStream(fileName,
FileMode.Open, FileAccess.Read))
            {
                byte[] buffer = new byte[readStream.Length];
                int bytesRead = readStream.Read(buffer, 0,
buffer.Length);

                string content = Encoding.UTF8.GetString(buffer, 0,
bytesRead);
                Console.WriteLine($"âœ“ Read {bytesRead} bytes from
{fileName}");
                Console.WriteLine($"Content: {content}");
            }
        }
        catch (Exception ex)
        {
0
```

```
            Console.WriteLine($"Error: {ex.Message}");
        }
        finally
        {
            // Cleanup
            if (File.Exists(fileName))
                File.Delete(fileName);
        }

        Console.WriteLine();
    }

    public static void StreamWriterReaderDemo()
    {
        Console.WriteLine("2. StreamWriter and StreamReader:");

        string fileName = "text_stream_demo.txt";

        try
        {
            // WRITING text using StreamWriter
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Create))
            using (StreamWriter writer = new StreamWriter(fileStream,
Encoding.UTF8))
            {
                writer.WriteLine("=== Student Records ===");
                writer.WriteLine($"Date: {DateTime.Now:yyyy-MM-dd
HH:mm:ss}");
                writer.WriteLine();

                // Write multiple lines
                string[] students = {
                    "John Doe, 23, Computer Science",
                    "Jane Smith, 22, Mathematics",
                    "Bob Johnson, 24, Physics",
                    "Alice Brown, 21, Chemistry"
                };

                foreach (string student in students)
```

```
                {
                    writer.WriteLine(student);
                }

                writer.WriteLine();
                writer.WriteLine($"Total students: {students.Length}");

                Console.WriteLine($"âœ“ Written student records to
{fileName}");
            }

            // READING text using StreamReader
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Open))
            using (StreamReader reader = new StreamReader(fileStream,
Encoding.UTF8))
            {
                Console.WriteLine("âœ“ Reading content from file:");

                string line;
                int lineNumber = 1;

                while ((line = reader.ReadLine()) != null)
                {
                    Console.WriteLine($"Line {lineNumber:D2}: {line}");
                    lineNumber++;
                }

                // Alternative: Read entire file at once
                // string allContent = reader.ReadToEnd();
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        finally
        {
            if (File.Exists(fileName))
                File.Delete(fileName);
0
```

```
        }

        Console.WriteLine();
    }

    public static void BinaryStreamDemo()
    {
        Console.WriteLine("3. Binary Stream Operations:");

        string fileName = "binary_stream_demo.dat";

        try
        {
            // WRITING binary data
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Create))
            using (BinaryWriter writer = new BinaryWriter(fileStream))
            {
                // Write different data types
                writer.Write(42);                           // int
                writer.Write(3.14159);                      // double
                writer.Write("Binary Stream Demo");         // string
                writer.Write(true);                         // bool
                writer.Write(DateTime.Now.ToBinary());      // DateTime as
binary

                // Write array
                int[] numbers = { 1, 2, 3, 4, 5 };
                writer.Write(numbers.Length);
                foreach (int number in numbers)
                {
                    writer.Write(number);
                }

                Console.WriteLine("âœ“ Written binary data to file");
            }

            // READING binary data
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Open))
```

```
            using (BinaryReader reader = new BinaryReader(fileStream))
            {
                int intValue = reader.ReadInt32();
                double doubleValue = reader.ReadDouble();
                string stringValue = reader.ReadString();
                bool boolValue = reader.ReadBoolean();
                DateTime dateValue =
DateTime.FromBinary(reader.ReadInt64());

                Console.WriteLine("âœ“ Read binary data from file:");
                Console.WriteLine($"  Integer: {intValue}");
                Console.WriteLine($"  Double: {doubleValue}");
                Console.WriteLine($"  String: {stringValue}");
                Console.WriteLine($"  Boolean: {boolValue}");
                Console.WriteLine($"  DateTime: {dateValue:yyyy-MM-dd
HH:mm:ss}");

                // Read array
                int arrayLength = reader.ReadInt32();
                int[] numbers = new int[arrayLength];
                for (int i = 0; i < arrayLength; i++)
                {
                    numbers[i] = reader.ReadInt32();
                }
                Console.WriteLine($"  Array: [{string.Join(", ",
numbers)}]");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        finally
        {
            if (File.Exists(fileName))
                File.Delete(fileName);
        }

        Console.WriteLine();
    }
0
```

```
    public static void MemoryStreamDemo()
    {
        Console.WriteLine("4. Memory Stream Operations:");

        try
        {
            // Create and work with MemoryStream
            using (MemoryStream memoryStream = new MemoryStream())
            {
                // Write to memory stream
                string data = "This data is stored in memory, not on
disk!";
                byte[] bytes = Encoding.UTF8.GetBytes(data);

                memoryStream.Write(bytes, 0, bytes.Length);
                Console.WriteLine($"âœ“ Written {bytes.Length} bytes to
memory stream");

                // Reset position to beginning
                memoryStream.Position = 0;

                // Read from memory stream
                byte[] readBuffer = new byte[memoryStream.Length];
                int bytesRead = memoryStream.Read(readBuffer, 0,
readBuffer.Length);

                string readData = Encoding.UTF8.GetString(readBuffer, 0,
bytesRead);
                Console.WriteLine($"âœ“ Read {bytesRead} bytes from
memory stream");
                Console.WriteLine($"Content: {readData}");

                // Get all data as byte array
                byte[] allData = memoryStream.ToArray();
                Console.WriteLine($"Total memory stream size:
{allData.Length} bytes");
            }

            // Memory stream with initial data
0
```

```
            byte[] initialData = Encoding.UTF8.GetBytes("Initial memory
data");
            using (MemoryStream preloadedStream = new
MemoryStream(initialData))
            {
                using (StreamReader reader = new
StreamReader(preloadedStream))
                {
                    string content = reader.ReadToEnd();
                    Console.WriteLine($"Preloaded content: {content}");
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        Console.WriteLine();
    }

    public static void BufferedStreamDemo()
    {
        Console.WriteLine("5. Buffered Stream Operations:");

        string fileName = "buffered_stream_demo.txt";

        try
        {
            // Writing with BufferedStream
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Create))
            using (BufferedStream bufferedStream = new
BufferedStream(fileStream, 4096)) // 4KB buffer
            {
                byte[] data = Encoding.UTF8.GetBytes("Buffered stream
improves performance for small, frequent operations.\n");

                // Write the same data multiple times
                for (int i = 0; i < 100; i++)
```

```
                {
                    bufferedStream.Write(data, 0, data.Length);
                }

                Console.WriteLine("âœ“ Written data using
BufferedStream");
            }

            // Reading with BufferedStream
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Open))
            using (BufferedStream bufferedStream = new
BufferedStream(fileStream, 4096))
            {
                byte[] buffer = new byte[1024];
                int totalBytesRead = 0;
                int bytesRead;

                while ((bytesRead = bufferedStream.Read(buffer, 0,
buffer.Length)) > 0)
                {
                    totalBytesRead += bytesRead;
                }

                Console.WriteLine($"âœ“ Read {totalBytesRead} bytes using
BufferedStream");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        finally
        {
            if (File.Exists(fileName))
                File.Delete(fileName);
        }

        Console.WriteLine();
    }
0
```

```
    public static void AdvancedStreamOperations()
    {
        Console.WriteLine("6. Advanced Stream Operations:");

        string fileName = "advanced_stream_demo.json";

        try
        {
            // Create complex data structure
            var studentData = new
            {
                Students = new[]
                {
                    new { Name = "John Doe", Age = 23, GPA = 3.75 },
                    new { Name = "Jane Smith", Age = 22, GPA = 3.95 },
                    new { Name = "Bob Johnson", Age = 24, GPA = 3.60 }
                },
                CourseCode = "CS101",
                Semester = "Fall 2024"
            };

            // Write JSON-like data
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Create))
            using (StreamWriter writer = new StreamWriter(fileStream,
Encoding.UTF8))
            {
                writer.WriteLine("{");
                writer.WriteLine($"  \"CourseCode\": \"
{studentData.CourseCode}\",");
                writer.WriteLine($"  \"Semester\": \"
{studentData.Semester}\",");
                writer.WriteLine("  \"Students\": [");

                for (int i = 0; i < studentData.Students.Length; i++)
                {
                    var student = studentData.Students[i];
                    writer.WriteLine("    {");
                    writer.WriteLine($"      \"Name\": \"
```

```
{student.Name}\",");
                    writer.WriteLine($"      \"Age\": {student.Age},");
                    writer.WriteLine($"      \"GPA\": {student.GPA}");
                    writer.Write("    }");

                    if (i < studentData.Students.Length - 1)
                        writer.WriteLine(",");
                    else
                        writer.WriteLine();
                }

                writer.WriteLine("  ]");
                writer.WriteLine("}");
            }

            Console.WriteLine("âœ“ Written JSON-like data to file");

            // Read and parse the data
            using (FileStream fileStream = new FileStream(fileName,
FileMode.Open))
            using (StreamReader reader = new StreamReader(fileStream))
            {
                Console.WriteLine("âœ“ File contents:");

                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    Console.WriteLine($"  {line}");
                }
            }

            // Demonstrate stream copying
            CopyStreamDemo(fileName);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        finally
        {
0
```

```
            if (File.Exists(fileName))
                File.Delete(fileName);
            if (File.Exists("copied_" + fileName))
                File.Delete("copied_" + fileName);
        }
    }

    public static void CopyStreamDemo(string sourceFileName)
    {
        string targetFileName = "copied_" + sourceFileName;

        try
        {
            using (FileStream source = new FileStream(sourceFileName,
FileMode.Open, FileAccess.Read))
            using (FileStream target = new FileStream(targetFileName,
FileMode.Create, FileAccess.Write))
            {
                source.CopyTo(target);
                Console.WriteLine($"âœ“ Copied {sourceFileName} to
{targetFileName}");
            }

            // Verify copy
            FileInfo sourceInfo = new FileInfo(sourceFileName);
            FileInfo targetInfo = new FileInfo(targetFileName);

            Console.WriteLine($"Source size: {sourceInfo.Length} bytes");
            Console.WriteLine($"Target size: {targetInfo.Length} bytes");
            Console.WriteLine($"Copy successful: {sourceInfo.Length ==
targetInfo.Length}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Copy error: {ex.Message}");
        }
    }
}

// Utility class for stream operations
```

0

```
public static class StreamUtilities
{
    // Read entire file as string
    public static string ReadAllText(string fileName)
    {
        using (FileStream stream = new FileStream(fileName,
FileMode.Open))
        using (StreamReader reader = new StreamReader(stream))
        {
            return reader.ReadToEnd();
        }
    }

    // Write string to file
    public static void WriteAllText(string fileName, string content)
    {
        using (FileStream stream = new FileStream(fileName,
FileMode.Create))
        using (StreamWriter writer = new StreamWriter(stream))
        {
            writer.Write(content);
        }
    }

    // Append text to file
    public static void AppendText(string fileName, string content)
    {
        using (FileStream stream = new FileStream(fileName,
FileMode.Append))
        using (StreamWriter writer = new StreamWriter(stream))
        {
            writer.WriteLine(content);
        }
    }

    // Count lines in a text file
    public static int CountLines(string fileName)
    {
        int lineCount = 0;
        using (FileStream stream = new FileStream(fileName,
```

```
FileMode.Open))
        using (StreamReader reader = new StreamReader(stream))
        {
            while (reader.ReadLine() != null)
            {
                lineCount++;
            }
        }
        return lineCount;
    }

    // Convert stream to byte array
    public static byte[] StreamToByteArray(Stream stream)
    {
        using (MemoryStream memoryStream = new MemoryStream())
        {
            stream.CopyTo(memoryStream);
            return memoryStream.ToArray();
        }
    }
}

// Custom stream wrapper example
public class LoggingStream : Stream
{
    private Stream baseStream;
    private string logPrefix;

    public LoggingStream(Stream baseStream, string logPrefix)
    {
        this.baseStream = baseStream;
        this.logPrefix = logPrefix;
    }

    public override bool CanRead => baseStream.CanRead;
    public override bool CanSeek => baseStream.CanSeek;
    public override bool CanWrite => baseStream.CanWrite;
    public override long Length => baseStream.Length;

    public override long Position
```

```
    {
        get => baseStream.Position;
        set => baseStream.Position = value;
    }

    public override void Flush()
    {
        Console.WriteLine($"{logPrefix}: Flushing stream");
        baseStream.Flush();
    }

    public override int Read(byte[] buffer, int offset, int count)
    {
        int bytesRead = baseStream.Read(buffer, offset, count);
        Console.WriteLine($"{logPrefix}: Read {bytesRead} bytes");
        return bytesRead;
    }

    public override long Seek(long offset, SeekOrigin origin)
    {
        long position = baseStream.Seek(offset, origin);
        Console.WriteLine($"{logPrefix}: Seek to position {position}");
        return position;
    }

    public override void SetLength(long value)
    {
        Console.WriteLine($"{logPrefix}: Set length to {value}");
        baseStream.SetLength(value);
    }

    public override void Write(byte[] buffer, int offset, int count)
    {
        Console.WriteLine($"{logPrefix}: Writing {count} bytes");
        baseStream.Write(buffer, offset, count);
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
```

0

```
        {
            Console.WriteLine($"{logPrefix}: Disposing stream");
            baseStream?.Dispose();
        }
        base.Dispose(disposing);
    }
}
```

Types of Streams in C#:

Stream Type	Purpose	Use Case
FileStream	File I/O operations	Reading/writing files
MemoryStream	In-memory data	Temporary data storage
NetworkStream	Network communication	TCP/UDP data transfer
BufferedStream	Buffered I/O	Performance improvement
StreamReader/Writer	Text operations	Reading/writing text
BinaryReader/Writer	Binary operations	Reading/writing binary data

Key Stream Properties:

- **CanRead**: Whether stream supports reading
- **CanWrite**: Whether stream supports writing
- **CanSeek**: Whether stream supports seeking
- **Length**: Total length of stream
- **Position**: Current position in stream

Best Practices:

1. **Always use `using` statements** for automatic disposal
2. **Choose appropriate stream type** for your data
3. **Use buffered streams** for frequent small operations
4. **Handle exceptions** properly
5. **Set appropriate buffer sizes** for performance
6. **Close streams explicitly** or use `using` blocks

Question 24: Write a program to create a thread along with main thread.

```
using System;
using System.Threading;
```

```
public class ThreadDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Threading Demonstration ===\n");

        // Show main thread information
        ShowMainThreadInfo();

        // Basic thread creation
        BasicThreadCreation();

        // Thread with parameters
        ThreadWithParameters();

        // Multiple threads
        MultipleThreadsDemo();

        // Thread synchronization
        ThreadSynchronizationDemo();

        // Background vs Foreground threads
        BackgroundThreadDemo();

        Console.WriteLine("\n=== Main thread ending ===");
    }

    public static void ShowMainThreadInfo()
    {
        Thread mainThread = Thread.CurrentThread;
        Console.WriteLine("MAIN THREAD INFORMATION:");
        Console.WriteLine($"Thread ID: {mainThread.ManagedThreadId}");
        Console.WriteLine($"Thread Name: {mainThread.Name ?? "Not
Set"}");
        Console.WriteLine($"Is Background: {mainThread.IsBackground}");
        Console.WriteLine($"Thread State: {mainThread.ThreadState}");
        Console.WriteLine($"Priority: {mainThread.Priority}");
        Console.WriteLine();
```

0

```
        // Set main thread name
        mainThread.Name = "MainThread";
    }

    public static void BasicThreadCreation()
    {
        Console.WriteLine("1. BASIC THREAD CREATION:");

        // Create and start a new thread
        Thread workerThread = new Thread(WorkerMethod);
        workerThread.Name = "WorkerThread1";

        Console.WriteLine($"Main thread: Creating worker thread...");
        workerThread.Start();

        // Main thread continues its work
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"Main thread: Working step {i + 1}");
            Thread.Sleep(500); // Sleep for 500ms
        }

        // Wait for worker thread to complete
        workerThread.Join();
        Console.WriteLine("Main thread: Worker thread completed\n");
    }

    public static void WorkerMethod()
    {
        Thread currentThread = Thread.CurrentThread;
        Console.WriteLine($"Worker thread started - ID:
{currentThread.ManagedThreadId}, Name: {currentThread.Name}");

        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine($"  Worker thread: Task {i + 1}
executing...");
            Thread.Sleep(700); // Simulate work
        }
```

0

```
        Console.WriteLine("Worker thread finished");
    }

    public static void ThreadWithParameters()
    {
        Console.WriteLine("2. THREAD WITH PARAMETERS:");

        // Using ParameterizedThreadStart
        Thread paramThread = new Thread(ParameterizedWorker);
        paramThread.Name = "ParameterThread";

        // Pass parameter to thread
        string message = "Hello from parameterized thread!";
        paramThread.Start(message);

        // Using lambda expression with captured variables
        int count = 3;
        string prefix = "Lambda";

        Thread lambdaThread = new Thread(() => LambdaWorker(count,
prefix));
        lambdaThread.Name = "LambdaThread";
        lambdaThread.Start();

        // Wait for both threads
        paramThread.Join();
        lambdaThread.Join();

        Console.WriteLine();
    }

    public static void ParameterizedWorker(object parameter)
    {
        string message = parameter as string;
        Thread currentThread = Thread.CurrentThread;

        Console.WriteLine($"Parameterized worker started - Thread:
{currentThread.Name}");
        Console.WriteLine($"Received parameter: {message}");
```

0

```
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine($"  {currentThread.Name}: Processing {i +
1}");
            Thread.Sleep(400);
        }
    }

    public static void LambdaWorker(int count, string prefix)
    {
        Thread currentThread = Thread.CurrentThread;
        Console.WriteLine($"Lambda worker started - Thread:
{currentThread.Name}");

        for (int i = 0; i < count; i++)
        {
            Console.WriteLine($"  {prefix} thread: Item {i + 1}");
            Thread.Sleep(300);
        }
    }

    public static void MultipleThreadsDemo()
    {
        Console.WriteLine("3. MULTIPLE THREADS:");

        // Create array of threads
        Thread[] threads = new Thread[3];

        for (int i = 0; i < threads.Length; i++)
        {
            int threadNumber = i + 1; // Capture loop variable
            threads[i] = new Thread(() => MultiWorker(threadNumber));
            threads[i].Name = $"MultiThread{threadNumber}";
            threads[i].Start();
        }

        Console.WriteLine("Main thread: All worker threads started");

        // Wait for all threads to complete
        foreach (Thread thread in threads)
```

```
        {
            thread.Join();
        }

        Console.WriteLine("Main thread: All worker threads completed\n");
    }

    public static void MultiWorker(int threadNumber)
    {
        Thread currentThread = Thread.CurrentThread;

        for (int i = 0; i < 4; i++)
        {
            Console.WriteLine($"  Thread {threadNumber}: Step {i + 1}
(ID: {currentThread.ManagedThreadId})");
            Thread.Sleep(200 * threadNumber); // Different sleep times
        }
    }

    // Shared resource for synchronization demo
    private static int sharedCounter = 0;
    private static readonly object lockObject = new object();

    public static void ThreadSynchronizationDemo()
    {
        Console.WriteLine("4. THREAD SYNCHRONIZATION:");

        sharedCounter = 0;

        // Create threads that access shared resource
        Thread[] syncThreads = new Thread[3];

        for (int i = 0; i < syncThreads.Length; i++)
        {
            int threadId = i + 1;
            syncThreads[i] = new Thread(() =>
SynchronizedWorker(threadId));
            syncThreads[i].Name = $"SyncThread{threadId}";
            syncThreads[i].Start();
        }
```

```
        // Wait for all threads
        foreach (Thread thread in syncThreads)
        {
            thread.Join();
        }

        Console.WriteLine($"Final shared counter value:
{sharedCounter}");
        Console.WriteLine();
    }

    public static void SynchronizedWorker(int threadId)
    {
        for (int i = 0; i < 5; i++)
        {
            // Synchronize access to shared resource
            lock (lockObject)
            {
                int currentValue = sharedCounter;
                Console.WriteLine($"  Thread {threadId}: Reading counter
= {currentValue}");

                // Simulate some processing time
                Thread.Sleep(10);

                sharedCounter = currentValue + 1;
                Console.WriteLine($"  Thread {threadId}: Updated counter
= {sharedCounter}");
            }

            // Do some work outside the lock
            Thread.Sleep(50);
        }
    }

    public static void BackgroundThreadDemo()
    {
        Console.WriteLine("5. BACKGROUND vs FOREGROUND THREADS:");
```

0

```
        // Foreground thread (default)
        Thread foregroundThread = new Thread(LongRunningTask);
        foregroundThread.Name = "ForegroundThread";
        foregroundThread.IsBackground = false;

        // Background thread
        Thread backgroundThread = new Thread(LongRunningTask);
        backgroundThread.Name = "BackgroundThread";
        backgroundThread.IsBackground = true;

        Console.WriteLine("Starting foreground and background
threads...");

        foregroundThread.Start();
        backgroundThread.Start();

        // Wait for foreground thread only
        // Background thread will be terminated when main thread ends
        foregroundThread.Join();

        Console.WriteLine("Foreground thread completed");
    }

    public static void LongRunningTask()
    {
        Thread currentThread = Thread.CurrentThread;

        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"  {currentThread.Name}: Long task step {i
+ 1}");
            Thread.Sleep(200);

            // Check if thread should abort
            if (currentThread.IsBackground && i > 5)
            {
                Console.WriteLine($"  {currentThread.Name}: Background
thread may be terminated soon");
            }
        }
0
```

```
        Console.WriteLine($"  {currentThread.Name}: Long task
completed");
    }
}

// Advanced threading examples
public class AdvancedThreadingDemo
{
    private static AutoResetEvent autoEvent = new AutoResetEvent(false);
    private static ManualResetEvent manualEvent = new
ManualResetEvent(false);

    public static void WaitHandleDemo()
    {
        Console.WriteLine("\n6. WAIT HANDLES DEMONSTRATION:");

        // AutoResetEvent example
        Thread autoResetThread = new Thread(AutoResetWorker);
        autoResetThread.Name = "AutoResetThread";
        autoResetThread.Start();

        Thread.Sleep(1000);
        Console.WriteLine("Main: Signaling AutoResetEvent");
        autoEvent.Set(); // Signal the waiting thread

        autoResetThread.Join();

        // ManualResetEvent example
        Thread[] manualResetThreads = new Thread[3];

        for (int i = 0; i < manualResetThreads.Length; i++)
        {
            int threadNum = i + 1;
            manualResetThreads[i] = new Thread(() =>
ManualResetWorker(threadNum));
            manualResetThreads[i].Name = $"ManualResetThread{threadNum}";
            manualResetThreads[i].Start();
        }
```

```
        Thread.Sleep(1000);
        Console.WriteLine("Main: Signaling ManualResetEvent (all threads
will proceed)");
        manualEvent.Set(); // Signal all waiting threads

        foreach (Thread thread in manualResetThreads)
        {
            thread.Join();
        }

        Console.WriteLine();
    }

    public static void AutoResetWorker()
    {
        Console.WriteLine("AutoReset worker: Waiting for signal...");
        autoEvent.WaitOne(); // Wait for signal
        Console.WriteLine("AutoReset worker: Received signal, continuing
work");

        Thread.Sleep(500);
        Console.WriteLine("AutoReset worker: Work completed");
    }

    public static void ManualResetWorker(int threadNumber)
    {
        Console.WriteLine($"ManualReset worker {threadNumber}: Waiting
for signal...");
        manualEvent.WaitOne(); // Wait for signal
        Console.WriteLine($"ManualReset worker {threadNumber}: Received
signal, doing work");

        Thread.Sleep(300 * threadNumber);
        Console.WriteLine($"ManualReset worker {threadNumber}: Work
completed");
    }
}

// Producer-Consumer pattern example
public class ProducerConsumerDemo
```

```
{
    private static Queue<int> queue = new Queue<int>();
    private static readonly object queueLock = new object();
    private static bool stopProducing = false;

    public static void RunDemo()
    {
        Console.WriteLine("\n7. PRODUCER-CONSUMER PATTERN:");

        // Create producer thread
        Thread producer = new Thread(Producer);
        producer.Name = "Producer";
        producer.Start();

        // Create consumer threads
        Thread consumer1 = new Thread(Consumer);
        Thread consumer2 = new Thread(Consumer);
        consumer1.Name = "Consumer1";
        consumer2.Name = "Consumer2";

        consumer1.Start();
        consumer2.Start();

        // Let them run for a while
        Thread.Sleep(3000);

        // Signal to stop producing
        stopProducing = true;

        // Wait for all threads
        producer.Join();
        consumer1.Join();
        consumer2.Join();

        Console.WriteLine("Producer-Consumer demo completed\n");
    }

    public static void Producer()
    {
        int item = 1;
```

0

```
        while (!stopProducing)
        {
            lock (queueLock)
            {
                queue.Enqueue(item);
                Console.WriteLine($"Producer: Produced item {item}");
                item++;
            }

            Thread.Sleep(100); // Produce every 100ms
        }

        Console.WriteLine("Producer: Stopped producing");
    }

    public static void Consumer()
    {
        Thread currentThread = Thread.CurrentThread;

        while (!stopProducing || queue.Count > 0)
        {
            int item = -1;
            bool hasItem = false;

            lock (queueLock)
            {
                if (queue.Count > 0)
                {
                    item = queue.Dequeue();
                    hasItem = true;
                }
            }

            if (hasItem)
            {
                Console.WriteLine($"{currentThread.Name}: Consumed item
{item}");
                Thread.Sleep(150); // Simulate processing time
            }
```

0

```
            else
            {
                Thread.Sleep(50); // Wait a bit before checking again
            }
        }

        Console.WriteLine($"{currentThread.Name}: Stopped consuming");
    }
}
```

Key Threading Concepts:

Concept	Description	Example Usage
Thread Creation	`new Thread(method)`	Basic thread creation
Thread.Start()	Begin thread execution	Start the thread
Thread.Join()	Wait for thread completion	Synchronize threads
Thread.Sleep()	Pause thread execution	Delay operations
Lock statement	Synchronize access	Protect shared resources
Background threads	Die with main thread	Service operations

Thread States:

- **Unstarted**: Created but not started
- **Running**: Currently executing
- **WaitSleepJoin**: Waiting or sleeping
- **Stopped**: Execution completed
- **Aborted**: Thread was aborted

Best Practices:

1. **Use `Thread.Join()`** to wait for thread completion
2. **Synchronize access** to shared resources with `lock`
3. **Set meaningful thread names** for debugging
4. **Handle exceptions** within thread methods
5. **Consider using `Task`** instead of `Thread` for newer applications
6. **Use background threads** for cleanup operations

Question 25: What is polymorphism? Explain with a brief example.

0

What is Polymorphism?

Polymorphism (Greek: "many forms") is an Object-Oriented Programming principle that allows objects of different types to be treated as instances of the same base type, while each object maintains its own specific behavior.

Types of Polymorphism in C#:

1. **Compile-time Polymorphism** (Method Overloading, Operator Overloading)
2. **Runtime Polymorphism** (Method Overriding, Interface Implementation)

```
using System;
using System.Collections.Generic;

// Base class for runtime polymorphism
public abstract class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }

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

    // Virtual method - can be overridden
    public virtual void MakeSound()
    {
        Console.WriteLine($"{Name} makes a generic animal sound");
    }

    // Abstract method - must be overridden
    public abstract void Move();

    // Regular method - inherited as-is
    public void DisplayInfo()
    {
        Console.WriteLine($"Animal: {Name}, Age: {Age}");
    }
```

0

```
}

// Derived classes demonstrating polymorphism
public class Dog : Animal
{
    public string Breed { get; set; }

    public Dog(string name, int age, string breed) : base(name, age)
    {
        Breed = breed;
    }

    // Override virtual method - Runtime Polymorphism
    public override void MakeSound()
    {
        Console.WriteLine($"{Name} the {Breed} says: Woof! Woof!");
    }

    // Override abstract method
    public override void Move()
    {
        Console.WriteLine($"{Name} runs around energetically");
    }

    // Dog-specific method
    public void Fetch()
    {
        Console.WriteLine($"{Name} fetches the ball");
    }
}

public class Cat : Animal
{
    public bool IsIndoor { get; set; }

    public Cat(string name, int age, bool isIndoor) : base(name, age)
    {
        IsIndoor = isIndoor;
    }
```

0

```
    public override void MakeSound()
    {
        Console.WriteLine($"{Name} the cat says: Meow! Meow!");
    }

    public override void Move()
    {
        Console.WriteLine($"{Name} moves gracefully and silently");
    }

    public void Purr()
    {
        Console.WriteLine($"{Name} purrs contentedly");
    }
}

public class Bird : Animal
{
    public bool CanFly { get; set; }

    public Bird(string name, int age, bool canFly) : base(name, age)
    {
        CanFly = canFly;
    }

    public override void MakeSound()
    {
        Console.WriteLine($"{Name} the bird says: Tweet! Tweet!");
    }

    public override void Move()
    {
        if (CanFly)
            Console.WriteLine($"{Name} soars through the sky");
        else
            Console.WriteLine($"{Name} hops around on the ground");
    }

    public void BuildNest()
    {
```

0

```
        Console.WriteLine($"{Name} builds a cozy nest");
    }
}

// Interface for polymorphism
public interface IPlayable
{
    void Play();
    void Rest();
}

// Classes implementing interface polymorphism
public class PlayfulDog : Dog, IPlayable
{
    public PlayfulDog(string name, int age, string breed) : base(name,
age, breed)
    {
    }

    public void Play()
    {
        Console.WriteLine($"{Name} plays fetch and runs around happily");
    }

    public void Rest()
    {
        Console.WriteLine($"{Name} takes a nap in the sun");
    }
}

public class PlayfulCat : Cat, IPlayable
{
    public PlayfulCat(string name, int age, bool isIndoor) : base(name,
age, isIndoor)
    {
    }

    public void Play()
    {
        Console.WriteLine($"{Name} plays with a ball of yarn");
```

```
    }

    public void Rest()
    {
        Console.WriteLine($"{Name} curls up for a cozy nap");
    }
}

// Demonstration of compile-time polymorphism (Method Overloading)
public class Calculator
{
    // Method overloading - same name, different parameters
    public int Add(int a, int b)
    {
        Console.WriteLine("Adding two integers");
        return a + b;
    }

    public double Add(double a, double b)
    {
        Console.WriteLine("Adding two doubles");
        return a + b;
    }

    public int Add(int a, int b, int c)
    {
        Console.WriteLine("Adding three integers");
        return a + b + c;
    }

    public string Add(string a, string b)
    {
        Console.WriteLine("Concatenating two strings");
        return a + b;
    }
}

// Main demonstration program
public class PolymorphismDemo
{
```

0

```
    public static void Main()
    {
        Console.WriteLine("=== Polymorphism Demonstration ===\n");

        // Runtime Polymorphism Demo
        RuntimePolymorphismDemo();

        // Interface Polymorphism Demo
        InterfacePolymorphismDemo();

        // Compile-time Polymorphism Demo
        CompileTimePolymorphismDemo();

        // Advanced Polymorphism Examples
        AdvancedPolymorphismDemo();
    }

    public static void RuntimePolymorphismDemo()
    {
        Console.WriteLine("1. RUNTIME POLYMORPHISM (Method
Overriding):");

        // Create array of Animal references pointing to different
derived objects
        Animal[] animals = {
            new Dog("Buddy", 5, "Golden Retriever"),
            new Cat("Whiskers", 3, true),
            new Bird("Tweety", 2, true),
            new Dog("Rex", 7, "German Shepherd")
        };

        Console.WriteLine("Polymorphic behavior - same method call,
different implementations:");

        foreach (Animal animal in animals)
        {
            // Polymorphic method calls
            animal.DisplayInfo();           // Inherited method
            animal.MakeSound();            // Virtual method - different
implementation for each type
```

```
            animal.Move();                // Abstract method - must be
implemented by each type
            Console.WriteLine();
        }

        // Demonstrate virtual method behavior
        Console.WriteLine("Virtual method demonstration:");
        Animal genericAnimal = new Dog("Max", 4, "Labrador");
        genericAnimal.MakeSound(); // Calls Dog's implementation, not
Animal's

        Console.WriteLine();
    }

    public static void InterfacePolymorphismDemo()
    {
        Console.WriteLine("2. INTERFACE POLYMORPHISM:");

        // Create array of interface references
        IPlayable[] playableAnimals = {
            new PlayfulDog("Rover", 3, "Beagle"),
            new PlayfulCat("Mittens", 2, false),
            new PlayfulDog("Spot", 6, "Dalmatian")
        };

        Console.WriteLine("Interface polymorphism - different classes,
same interface:");

        foreach (IPlayable playable in playableAnimals)
        {
            playable.Play();
            playable.Rest();
            Console.WriteLine();
        }
    }

    public static void CompileTimePolymorphismDemo()
    {
        Console.WriteLine("3. COMPILE-TIME POLYMORPHISM (Method
Overloading):");
```

```
        Calculator calc = new Calculator();

        // Same method name, different parameter types - resolved at
compile time
        Console.WriteLine($"Result 1: {calc.Add(5, 10)}");
// int version
        Console.WriteLine($"Result 2: {calc.Add(3.14, 2.86)}");
// double version
        Console.WriteLine($"Result 3: {calc.Add(1, 2, 3)}");
// three int version
        Console.WriteLine($"Result 4: {calc.Add("Hello ", "World")}");
// string version

        Console.WriteLine();
    }

    public static void AdvancedPolymorphismDemo()
    {
        Console.WriteLine("4. ADVANCED POLYMORPHISM EXAMPLES:");

        // Polymorphic collection processing
        List<Animal> animalShelter = new List<Animal>
        {
            new Dog("Luna", 2, "Husky"),
            new Cat("Shadow", 4, true),
            new Bird("Phoenix", 1, false)
        };

        Console.WriteLine("Animal shelter daily routine:");
        ProcessAnimals(animalShelter);

        // Type checking and casting
        Console.WriteLine("\nType checking and specific behaviors:");
        foreach (Animal animal in animalShelter)
        {
            // Check type and call specific methods
            if (animal is Dog dog)
            {
                dog.Fetch();
```

```
            }
            else if (animal is Cat cat)
            {
                cat.Purr();
            }
            else if (animal is Bird bird)
            {
                bird.BuildNest();
            }
        }

        // Using 'as' operator
        Console.WriteLine("\nUsing 'as' operator for safe casting:");
        foreach (Animal animal in animalShelter)
        {
            Dog dog = animal as Dog;
            if (dog != null)
            {
                Console.WriteLine($"{dog.Name} is a {dog.Breed}");
            }
        }

        Console.WriteLine();
    }

    // Polymorphic method - works with any Animal type
    public static void ProcessAnimals(List<Animal> animals)
    {
        foreach (Animal animal in animals)
        {
            Console.WriteLine($"Processing {animal.GetType().Name}:
{animal.Name}");
            animal.MakeSound();
            animal.Move();

            // Polymorphic feeding
            FeedAnimal(animal);
            Console.WriteLine();
        }
    }
0
```

```
    // Another polymorphic method
    public static void FeedAnimal(Animal animal)
    {
        switch (animal)
        {
            case Dog _:
                Console.WriteLine($"Giving {animal.Name} dog food and
treats");
                break;
            case Cat _:
                Console.WriteLine($"Giving {animal.Name} cat food and
milk");
                break;
            case Bird _:
                Console.WriteLine($"Giving {animal.Name} seeds and
water");
                break;
            default:
                Console.WriteLine($"Giving {animal.Name} generic animal
food");
                break;
        }
    }
}

// Additional polymorphism example - Shape hierarchy
public abstract class Shape
{
    public abstract double CalculateArea();
    public abstract double CalculatePerimeter();

    public virtual void DisplayInfo()
    {
        Console.WriteLine($"{GetType().Name}: Area =
{CalculateArea():F2}, Perimeter = {CalculatePerimeter():F2}");
    }
}

public class Rectangle : Shape
```

```
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public override double CalculateArea()
    {
        return Width * Height;
    }

    public override double CalculatePerimeter()
    {
        return 2 * (Width + Height);
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }

    public override double CalculatePerimeter()
    {
        return 2 * Math.PI * Radius;
    }
}
```

0

```
public class Triangle : Shape
{
    public double Base { get; set; }
    public double Height { get; set; }
    public double Side1 { get; set; }
    public double Side2 { get; set; }

    public Triangle(double baseLength, double height, double side1,
double side2)
    {
        Base = baseLength;
        Height = height;
        Side1 = side1;
        Side2 = side2;
    }

    public override double CalculateArea()
    {
        return 0.5 * Base * Height;
    }

    public override double CalculatePerimeter()
    {
        return Base + Side1 + Side2;
    }
}

// Demonstration of shape polymorphism
public class ShapeDemo
{
    public static void DemonstrateShapePolymorphism()
    {
        Console.WriteLine("5. SHAPE POLYMORPHISM EXAMPLE:");

        Shape[] shapes = {
            new Rectangle(5, 4),
            new Circle(3),
            new Triangle(6, 4, 5, 5)
        };
```

0

```
        Console.WriteLine("Calculating areas and perimeters
polymorphically:");

        double totalArea = 0;
        foreach (Shape shape in shapes)
        {
            shape.DisplayInfo();
            totalArea += shape.CalculateArea();
        }

        Console.WriteLine($"Total area of all shapes: {totalArea:F2}");
    }
}
```

Key Benefits of Polymorphism:

1. **Code Reusability**: Same interface works with different implementations
2. **Flexibility**: Easy to add new types without changing existing code
3. **Maintainability**: Changes in implementation don't affect client code
4. **Abstraction**: Client code works with abstractions, not concrete types

Polymorphism Summary:

Type	Mechanism	Resolution Time	Example
Compile-time	Method Overloading	Compile time	`Add(int, int)` vs `Add(double, double)`
Runtime	Method Overriding	Runtime	Virtual/Abstract method calls
Interface	Interface Implementation	Runtime	Different classes implementing same interface

Real-world Example:

```
Animal shelter = new Dog();  // Dog treated as Animal
shelter.MakeSound();         // Calls Dog's MakeSound(), not Animal's
// Output: "Buddy the Golden Retriever says: Woof! Woof!"
```

0

This demonstrates polymorphism: same method call (`MakeSound()`), different behavior based on the actual object type at runtime.

Question 26: What is the use of using block in C#?

What is the Using Block?

The **using block** in C# provides a convenient syntax to ensure that resources are properly disposed of when they go out of scope. It automatically calls the `Dispose()` method on objects that implement the `IDisposable` interface.

```
using System;
using System.IO;
using System.Data.SqlClient;

public class UsingBlockDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Using Block Demonstration ===\n");

        // File operations with using block
        FileOperationsDemo();

        // Database operations with using block
        DatabaseOperationsDemo();

        // Multiple resources in using block
        MultipleResourcesDemo();

        // Custom disposable resources
        CustomDisposableDemo();

        // Nested using blocks
        NestedUsingDemo();

        // Using block vs manual disposal
        ComparisonDemo();
    }
```

0

```
    public static void FileOperationsDemo()
    {
        Console.WriteLine("1. FILE OPERATIONS WITH USING BLOCK:");

        string fileName = "using_demo.txt";

        // WRITING to file with using block
        using (FileStream fileStream = new FileStream(fileName,
FileMode.Create))
        using (StreamWriter writer = new StreamWriter(fileStream))
        {
            writer.WriteLine("This file was created using 'using'
block");
            writer.WriteLine($"Created at: {DateTime.Now}");
            writer.WriteLine("The file stream will be automatically
disposed");

            Console.WriteLine("âœ“ File written successfully");
            // FileStream and StreamWriter automatically disposed here
        }

        // READING from file with using block
        using (FileStream fileStream = new FileStream(fileName,
FileMode.Open))
        using (StreamReader reader = new StreamReader(fileStream))
        {
            Console.WriteLine("File contents:");
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                Console.WriteLine($"  {line}");
            }
            // FileStream and StreamReader automatically disposed here
        }

        // Cleanup
        if (File.Exists(fileName))
            File.Delete(fileName);

        Console.WriteLine();
0
```

```
    }

    public static void DatabaseOperationsDemo()
    {
        Console.WriteLine("2. DATABASE OPERATIONS WITH USING BLOCK:");

        // Note: This example shows the pattern, actual connection string
would be needed
        string connectionString =
"Server=localhost;Database=TestDB;Integrated Security=true;";

        try
        {
            // Database connection with using block
            using (SqlConnection connection = new
SqlConnection(connectionString))
            {
                connection.Open();
                Console.WriteLine("âœ“ Database connection opened");

                using (SqlCommand command = new SqlCommand("SELECT
COUNT(*) FROM Users", connection))
                {
                    // Simulate database operation
                    Console.WriteLine("âœ“ Executing SQL command");

                    // In real scenario, you would execute the command
                    // object result = command.ExecuteScalar();

                    Console.WriteLine("âœ“ Command executed
successfully");
                    // SqlCommand automatically disposed here
                }

                Console.WriteLine("âœ“ Database operations completed");
                // SqlConnection automatically disposed here (connection
closed)
            }
        }
        catch (Exception ex)
0
```

```
        {
            Console.WriteLine($"Database error: {ex.Message}");
        }

        Console.WriteLine();
    }

    public static void MultipleResourcesDemo()
    {
        Console.WriteLine("3. MULTIPLE RESOURCES WITH USING BLOCK:");

        string sourceFile = "source.txt";
        string targetFile = "target.txt";

        try
        {
            // Create source file
            File.WriteAllText(sourceFile, "This content will be copied to
another file.");

            // Multiple using statements for file copying
            using (FileStream source = new FileStream(sourceFile,
FileMode.Open))
            using (FileStream target = new FileStream(targetFile,
FileMode.Create))
            using (StreamReader reader = new StreamReader(source))
            using (StreamWriter writer = new StreamWriter(target))
            {
                string content = reader.ReadToEnd();
                writer.Write(content);

                Console.WriteLine("âœ“ File copied successfully using
multiple using statements");
                // All four resources automatically disposed here
            }

            // Verify copy
            string copiedContent = File.ReadAllText(targetFile);
            Console.WriteLine($"Copied content: {copiedContent}");
        }
0
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"File operation error: {ex.Message}");
        }
        finally
        {
            // Cleanup
            if (File.Exists(sourceFile)) File.Delete(sourceFile);
            if (File.Exists(targetFile)) File.Delete(targetFile);
        }

        Console.WriteLine();
    }

    public static void CustomDisposableDemo()
    {
        Console.WriteLine("4. CUSTOM DISPOSABLE RESOURCES:");

        // Using custom disposable class
        using (var resource = new CustomResource("ResourceA"))
        {
            resource.DoWork();
            resource.ProcessData("Important data");

            Console.WriteLine("âœ“ Custom resource work completed");
            // CustomResource.Dispose() automatically called here
        }

        // Nested using with custom resources
        using (var resource1 = new CustomResource("Resource1"))
        using (var resource2 = new CustomResource("Resource2"))
        {
            resource1.DoWork();
            resource2.DoWork();

            Console.WriteLine("âœ“ Multiple custom resources used");
            // Both resources automatically disposed in reverse order
        }

        Console.WriteLine();
0
```

```
    }

    public static void NestedUsingDemo()
    {
        Console.WriteLine("5. NESTED USING BLOCKS:");

        string outerFile = "outer.txt";
        string innerFile = "inner.txt";

        try
        {
            using (var outerStream = new FileStream(outerFile,
FileMode.Create))
            using (var outerWriter = new StreamWriter(outerStream))
            {
                outerWriter.WriteLine("Outer file content");

                using (var innerStream = new FileStream(innerFile,
FileMode.Create))
                using (var innerWriter = new StreamWriter(innerStream))
                {
                    innerWriter.WriteLine("Inner file content");

                    Console.WriteLine("âœ“ Both files written in nested
using blocks");

                    // Inner resources disposed first, then outer
resources
                }

                outerWriter.WriteLine("Back to outer file");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error in nested operations:
{ex.Message}");
        }
        finally
        {
0
```

```
            if (File.Exists(outerFile)) File.Delete(outerFile);
            if (File.Exists(innerFile)) File.Delete(innerFile);
        }

        Console.WriteLine();
    }

    public static void ComparisonDemo()
    {
        Console.WriteLine("6. USING BLOCK vs MANUAL DISPOSAL:");

        // WITHOUT using block (manual disposal)
        Console.WriteLine("Manual disposal approach:");
        FileStream manualStream = null;
        StreamWriter manualWriter = null;

        try
        {
            manualStream = new FileStream("manual.txt", FileMode.Create);
            manualWriter = new StreamWriter(manualStream);

            manualWriter.WriteLine("Manually managed resources");
            Console.WriteLine("âœ“ Manual approach - resources created
and used");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
        finally
        {
            // Must manually dispose in finally block
            manualWriter?.Dispose();
            manualStream?.Dispose();
            Console.WriteLine("âœ“ Manual approach - resources manually
disposed");
        }

        // WITH using block (automatic disposal)
        Console.WriteLine("\nUsing block approach:");
0
```

```
        using (var autoStream = new FileStream("auto.txt",
FileMode.Create))
        using (var autoWriter = new StreamWriter(autoStream))
        {
            autoWriter.WriteLine("Automatically managed resources");
            Console.WriteLine("âœ“ Using block approach - resources
created and used");
            // Automatic disposal happens here, even if exception occurs
        }
        Console.WriteLine("âœ“ Using block approach - resources
automatically disposed");

        // Cleanup
        if (File.Exists("manual.txt")) File.Delete("manual.txt");
        if (File.Exists("auto.txt")) File.Delete("auto.txt");

        Console.WriteLine();
    }
}

// Custom disposable class for demonstration
public class CustomResource : IDisposable
{
    private string resourceName;
    private bool disposed = false;

    public CustomResource(string name)
    {
        resourceName = name;
        Console.WriteLine($"  {resourceName} created");
    }

    public void DoWork()
    {
        if (disposed)
            throw new ObjectDisposedException(resourceName);

        Console.WriteLine($"  {resourceName} is doing work");
    }
```

0

```
    public void ProcessData(string data)
    {
        if (disposed)
            throw new ObjectDisposedException(resourceName);

        Console.WriteLine($"  {resourceName} processing: {data}");
    }

    // IDisposable implementation
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this); // Prevent finalizer from running
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources
                Console.WriteLine($"  {resourceName} disposed (managed
resources cleaned up)");
            }

            // Dispose unmanaged resources
            // (None in this example)

            disposed = true;
        }
    }

    // Finalizer (destructor)
    ~CustomResource()
    {
        Dispose(false);
    }
}
```

0

```
// Advanced using block examples
public class AdvancedUsingExamples
{
    public static void UsingWithVariableDeclaration()
    {
        Console.WriteLine("7. USING WITH VARIABLE DECLARATION:");

        // C# 8.0+ using declaration (without block)
        using var file = new FileStream("declaration.txt",
FileMode.Create);
        using var writer = new StreamWriter(file);

        writer.WriteLine("Using declaration syntax");
        Console.WriteLine("âœ“ Resources declared with 'using' keyword");

        // Resources automatically disposed at end of enclosing scope
        // No explicit block needed

        if (File.Exists("declaration.txt"))
            File.Delete("declaration.txt");
    }

    public static void UsingWithExceptionHandling()
    {
        Console.WriteLine("8. USING WITH EXCEPTION HANDLING:");

        try
        {
            using (var resource = new
CustomResource("ExceptionResource"))
            {
                resource.DoWork();

                // Simulate an exception
                throw new InvalidOperationException("Simulated error");

                // This line won't execute
                resource.ProcessData("This won't be processed");
            }
        }
0
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"  Exception caught: {ex.Message}");
            Console.WriteLine("  Resource was still properly disposed
despite exception");
        }

        Console.WriteLine();
    }

    public static void UsingReturnEarly()
    {
        Console.WriteLine("9. USING WITH EARLY RETURN:");

        using (var resource = new CustomResource("EarlyReturnResource"))
        {
            resource.DoWork();

            if (DateTime.Now.Millisecond > 500)
            {
                Console.WriteLine("  Early return condition met");
                return; // Resource still gets disposed
            }

            resource.ProcessData("Additional processing");
        }
        // Resource disposed here if no early return

        Console.WriteLine();
    }
}
```

Key Benefits of Using Block:

Benefit	Description	Example
Automatic Disposal	Resources disposed automatically	`using (var file = ...) { }`
Exception Safety	Disposal happens even if exception occurs	Resources cleaned up in finally

0

Benefit	Description	Example
Cleaner Code	No need for explicit try-finally blocks	Less boilerplate code
Deterministic Cleanup	Resources released immediately	Not dependent on GC timing

What Using Block Does:

```
// Using block syntax:
using (FileStream fs = new FileStream("file.txt", FileMode.Create))
{
    // Use the resource
}

// Is equivalent to:
FileStream fs = new FileStream("file.txt", FileMode.Create);
try
{
    // Use the resource
}
finally
{
    fs?.Dispose(); // Always called, even if exception occurs
}
```

Common Use Cases:

1. **File Operations**: FileStream, StreamReader, StreamWriter
2. **Database Connections**: SqlConnection, SqlCommand
3. **Network Streams**: NetworkStream, TcpClient
4. **Graphics Resources**: Bitmap, Graphics objects
5. **Custom Resources**: Any class implementing IDisposable

Best Practices:

- **Always use `using` blocks** for IDisposable objects
- **Prefer using blocks over manual disposal** for exception safety
- **Use multiple using statements** for multiple resources
- **Consider using declarations (C# 8+)** for simpler syntax
- **Implement IDisposable properly** in custom classes

0

The using block ensures that resources are always properly cleaned up, making your code more reliable and preventing resource leaks.

Question 27: Differentiate between using block and using statement.

Using Block vs Using Statement in C#

While often confused, "using block" and "using statement" refer to different concepts in C#. Let me clarify the differences:

```
using System;           // <- Using STATEMENT (directive)
using System.IO;        // <- Using STATEMENT (directive)
using MyAlias = System.Collections.Generic.List<string>; // <- Using
STATEMENT (alias)

public class UsingComparison
{
    public static void Main()
    {
        Console.WriteLine("=== Using Block vs Using Statement Comparison
===\n");

        // Demonstrate using statements (directives)
        UsingStatementDemo();

        // Demonstrate using blocks
        UsingBlockDemo();

        // Demonstrate using declarations (C# 8+)
        UsingDeclarationDemo();

        // Show practical differences
        PracticalDifferencesDemo();
    }

    public static void UsingStatementDemo()
    {
        Console.WriteLine("1. USING STATEMENTS (Directives):");
        Console.WriteLine("   - Located at the top of source files");
        Console.WriteLine("   - Import namespaces or create aliases");
```

0

```
        Console.WriteLine("   - Compile-time feature");
        Console.WriteLine("   - Examples:");
        Console.WriteLine("     using System;");
        Console.WriteLine("     using System.IO;");
        Console.WriteLine("     using MyList =
System.Collections.Generic.List<int>;");
        Console.WriteLine();

        // Using an alias defined with using statement
        MyAlias stringList = new MyAlias { "Item1", "Item2", "Item3" };
        Console.WriteLine($"   Using alias: MyAlias contains
{stringList.Count} items");
        Console.WriteLine();
    }

    public static void UsingBlockDemo()
    {
        Console.WriteLine("2. USING BLOCKS:");
        Console.WriteLine("   - Used within method bodies");
        Console.WriteLine("   - Ensures automatic disposal of
resources");
        Console.WriteLine("   - Runtime feature");
        Console.WriteLine("   - Example:");

        string fileName = "using_block_example.txt";

        // This is a USING BLOCK
        using (FileStream fileStream = new FileStream(fileName,
FileMode.Create))
        using (StreamWriter writer = new StreamWriter(fileStream))
        {
            writer.WriteLine("This demonstrates a using BLOCK");
            writer.WriteLine("Resources are automatically disposed");
            Console.WriteLine("   âœ“ File written using using block");
        } // <- Resources automatically disposed here

        // Cleanup
        if (File.Exists(fileName))
        {
            File.Delete(fileName);
0
```

```
            Console.WriteLine("   âœ“ Temporary file cleaned up");
        }

        Console.WriteLine();
    }

    public static void UsingDeclarationDemo()
    {
        Console.WriteLine("3. USING DECLARATIONS (C# 8.0+):");
        Console.WriteLine("   - Simplified syntax without explicit
block");
        Console.WriteLine("   - Resources disposed at end of enclosing
scope");
        Console.WriteLine("   - Example:");

        string fileName = "using_declaration_example.txt";

        // This is a USING DECLARATION (no explicit block)
        using var fileStream = new FileStream(fileName, FileMode.Create);
        using var writer = new StreamWriter(fileStream);

        writer.WriteLine("This demonstrates a using DECLARATION");
        writer.WriteLine("No explicit block needed");
        Console.WriteLine("   âœ“ File written using using declaration");

        // Resources automatically disposed at end of method

        // Cleanup
        if (File.Exists(fileName))
        {
            File.Delete(fileName);
            Console.WriteLine("   âœ“ Temporary file cleaned up");
        }

        Console.WriteLine();
    }

    public static void PracticalDifferencesDemo()
    {
        Console.WriteLine("4. PRACTICAL DIFFERENCES:");
```

```
        DemonstrateNamespaceImport();
        DemonstrateAliasCreation();
        DemonstrateResourceManagement();
        DemonstrateGlobalUsings();
    }

    public static void DemonstrateNamespaceImport()
    {
        Console.WriteLine("   a) Namespace Import with Using
Statements:");

        // Without using statement, you would need to write:
        // System.DateTime now = System.DateTime.Now;
        // System.Console.WriteLine("Full namespace required");

        // With using statement at top of file:
        DateTime now = DateTime.Now; // 'using System;' allows this
        Console.WriteLine($"   Current time: {now:HH:mm:ss}");

        // Using alias for long namespace names
        var list = new MyAlias(); // MyAlias = List<string> defined at
top
        list.Add("Demonstration");
        Console.WriteLine($"   Alias usage: {list[0]}");
        Console.WriteLine();
    }

    public static void DemonstrateAliasCreation()
    {
        Console.WriteLine("   b) Alias Creation with Using Statements:");

        // Example of using statement for alias
        // using StringDictionary =
System.Collections.Generic.Dictionary<string, string>;

        var dict = new System.Collections.Generic.Dictionary<string,
string>
        {
            {"Key1", "Value1"},
0
```

```
            {"Key2", "Value2"}
        };

        Console.WriteLine($"   Dictionary contains {dict.Count} items");
        Console.WriteLine("   (Would be simpler with alias: using
StringDict = Dictionary<string, string>)");
        Console.WriteLine();
    }

    public static void DemonstrateResourceManagement()
    {
        Console.WriteLine("   c) Resource Management with Using
Blocks:");

        // Demonstrate proper resource disposal
        string tempFile = "resource_demo.tmp";

        try
        {
            // Using block ensures disposal
            using (var resource = new DisposableResource("ResourceA"))
            {
                resource.UseResource();
                Console.WriteLine("   âœ“ Resource used within using
block");

                // Even if exception occurs, resource is disposed
                if (DateTime.Now.Millisecond > 900) // Rare condition
                {
                    throw new Exception("Simulated exception");
                }
            } // Resource automatically disposed here

            Console.WriteLine("   âœ“ Resource automatically disposed");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"   Exception handled: {ex.Message}");
            Console.WriteLine("   âœ“ Resource still disposed despite
exception");
```

```
        }

        Console.WriteLine();
    }

    public static void DemonstrateGlobalUsings()
    {
        Console.WriteLine("   d) Global Using Statements (C# 10+):");
        Console.WriteLine("   - Can use 'global using' for project-wide
imports");
        Console.WriteLine("   - Example: global using System;");
        Console.WriteLine("   - Available in all files in the project");
        Console.WriteLine("   - Reduces repetitive using statements");
        Console.WriteLine();
    }
}

// Custom disposable class for demonstration
public class DisposableResource : IDisposable
{
    private string resourceName;
    private bool disposed = false;

    public DisposableResource(string name)
    {
        resourceName = name;
        Console.WriteLine($"     {resourceName} created");
    }

    public void UseResource()
    {
        if (disposed)
            throw new ObjectDisposedException(resourceName);

        Console.WriteLine($"     {resourceName} is being used");
    }

    public void Dispose()
    {
        Dispose(true);
```

```
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            Console.WriteLine($"     {resourceName} disposed");
            disposed = true;
        }
    }
}

// Demonstrate different using scenarios
public class AdvancedUsingScenarios
{
    // Using statements at different levels
    using System.Text; // <- Using statement (namespace import)

    public static void ComprehensiveDemo()
    {
        Console.WriteLine("5. COMPREHENSIVE USING SCENARIOS:");

        // Scenario 1: Multiple using blocks
        Console.WriteLine("   Scenario 1: Multiple using blocks");
        using (var resource1 = new DisposableResource("Resource1"))
        using (var resource2 = new DisposableResource("Resource2"))
        {
            resource1.UseResource();
            resource2.UseResource();
            // Both disposed in reverse order
        }

        // Scenario 2: Nested using blocks
        Console.WriteLine("\n   Scenario 2: Nested using blocks");
        using (var outer = new DisposableResource("Outer"))
        {
            outer.UseResource();

            using (var inner = new DisposableResource("Inner"))
0
```

```
            {
                inner.UseResource();
                // Inner disposed first
            }
            // Outer disposed after inner
        }

        // Scenario 3: Using declarations
        Console.WriteLine("\n   Scenario 3: Using declarations");
        using var declaration1 = new DisposableResource("Declaration1");
        using var declaration2 = new DisposableResource("Declaration2");

        declaration1.UseResource();
        declaration2.UseResource();

        // Both disposed at end of method in reverse order
        Console.WriteLine("   End of method - declarations will be
disposed");
    }
}
```

Detailed Comparison Table:

Aspect	Using Statement (Directive)	Using Block	Using Declaration
Location	Top of source file	Inside method body	Inside method body
Purpose	Import namespaces/create aliases	Resource management	Resource management
Syntax	`using System;`	`using (var x = ...) { }`	`using var x = ...;`
When Applied	Compile time	Runtime	Runtime
Scope	Entire file	Block only	Method/scope
Disposal	N/A	Automatic at end of block	Automatic at end of scope
C# Version	C# 1.0+	C# 1.0+	C# 8.0+

Real Examples:

0

```
// USING STATEMENTS (at top of file)
using System;                    // Namespace import
using System.IO;                 // Namespace import
using FileStream = System.IO.FileStream;  // Alias
global using System.Collections.Generic;  // Global using (C# 10+)

public class Examples
{
    public void DemonstrateAll()
    {
        // USING BLOCK
        using (var stream = new FileStream("file.txt", FileMode.Create))
        {
            // Use stream
        } // stream.Dispose() called automatically

        // USING DECLARATION (C# 8+)
        using var reader = new StreamReader("file.txt");
        // Use reader
        // reader.Dispose() called at end of method
    }
}
```

Key Takeaways:

1. **Using Statement**: Compile-time directive for namespace imports and aliases
2. **Using Block**: Runtime construct for automatic resource disposal with explicit scope
3. **Using Declaration**: Modern syntax for resource disposal without explicit blocks
4. **Different Purposes**: Statements are for namespaces, blocks/declarations are for resource management
5. **Choose Based on Need**: Use statements for imports, blocks/declarations for IDisposable objects

The confusion often arises because both use the `using` keyword, but they serve completely different purposes in the language.

Question 28: Explain following terms: Assembly, Namespace, Class, Property, Field, Methods.

0

Fundamental C# Concepts Explained

Let me explain each of these important C# concepts with practical examples:

```
// NAMESPACE - A logical grouping of types
namespace MyCompany.EmployeeManagement  // <- This is a NAMESPACE
{
    using System;
    using System.Collections.Generic;

    // CLASS - A blueprint for creating objects
    public class Employee  // <- This is a CLASS
    {
        // FIELDS - Private data storage (variables)
        private int employeeId;           // <- This is a FIELD
        private string firstName;         // <- This is a FIELD
        private string lastName;          // <- This is a FIELD
        private decimal salary;           // <- This is a FIELD
        private DateTime hireDate;        // <- This is a FIELD
        private List<string> skills;      // <- This is a FIELD

        // PROPERTIES - Public interface to access private fields
        public int EmployeeId             // <- This is a PROPERTY
        {
            get { return employeeId; }
            private set
            {
                if (value > 0)
                    employeeId = value;
                else
                    throw new ArgumentException("Employee ID must be
positive");
            }
        }

        public string FirstName           // <- This is a PROPERTY
        {
            get { return firstName; }
            set
            {
0
```

```
                if (!string.IsNullOrWhiteSpace(value))
                    firstName = value.Trim();
                else
                    throw new ArgumentException("First name cannot be
empty");
            }
        }

        public string LastName            // <- This is a PROPERTY
        {
            get { return lastName; }
            set
            {
                if (!string.IsNullOrWhiteSpace(value))
                    lastName = value.Trim();
                else
                    throw new ArgumentException("Last name cannot be
empty");
            }
        }

        public decimal Salary             // <- This is a PROPERTY
        {
            get { return salary; }
            set
            {
                if (value >= 0)
                    salary = value;
                else
                    throw new ArgumentException("Salary cannot be
negative");
            }
        }

        public DateTime HireDate          // <- This is a PROPERTY
        {
            get { return hireDate; }
            set { hireDate = value; }
        }
```

0

```
        // Auto-implemented property (compiler creates backing field)
        public string Department { get; set; }  // <- This is a PROPERTY

        // Read-only property
        public string FullName            // <- This is a PROPERTY
        {
            get { return $"{firstName} {lastName}"; }
        }

        // Read-only property with calculation
        public int YearsOfService         // <- This is a PROPERTY
        {
            get { return DateTime.Now.Year - hireDate.Year; }
        }

        // METHODS - Functions that define behavior

        // Constructor METHOD
        public Employee(int id, string first, string last, decimal sal,
DateTime hire)  // <- This is a METHOD
        {
            EmployeeId = id;
            FirstName = first;
            LastName = last;
            Salary = sal;
            HireDate = hire;
            skills = new List<string>();
        }

        // Instance METHOD
        public void AddSkill(string skill)   // <- This is a METHOD
        {
            if (!string.IsNullOrWhiteSpace(skill) &&
!skills.Contains(skill))
            {
                skills.Add(skill);
                Console.WriteLine($"Added skill '{skill}' to
{FullName}");
            }
        }
0
```

```
        // Instance METHOD with return value
        public List<string> GetSkills()      // <- This is a METHOD
        {
            return new List<string>(skills); // Return copy to maintain
encapsulation
        }

        // Instance METHOD with parameters
        public void GiveRaise(decimal percentage)  // <- This is a METHOD
        {
            if (percentage > 0 && percentage <= 50)
            {
                decimal oldSalary = salary;
                salary += salary * (percentage / 100);
                Console.WriteLine($"{FullName} received a {percentage}%
raise");
                Console.WriteLine($"Salary increased from ${oldSalary:F2}
to ${salary:F2}");
            }
            else
            {
                throw new ArgumentException("Raise percentage must be
between 0 and 50");
            }
        }

        // Static METHOD (belongs to class, not instance)
        public static Employee CreateTemporaryEmployee(string first,
string last)  // <- This is a METHOD
        {
            return new Employee(0, first, last, 0, DateTime.Now);
        }

        // Virtual METHOD (can be overridden)
        public virtual void DisplayInfo()    // <- This is a METHOD
        {
            Console.WriteLine($"=== Employee Information ===");
            Console.WriteLine($"ID: {EmployeeId}");
            Console.WriteLine($"Name: {FullName}");
```

```
            Console.WriteLine($"Department: {Department ?? "Not
Assigned"}");
            Console.WriteLine($"Salary: ${Salary:F2}");
            Console.WriteLine($"Hire Date: {HireDate:yyyy-MM-dd}");
            Console.WriteLine($"Years of Service: {YearsOfService}");
            Console.WriteLine($"Skills: {(skills.Count > 0 ?
string.Join(", ", skills) : "None")}");
            Console.WriteLine($"============================");
        }

        // Override Object.ToString() METHOD
        public override string ToString()    // <- This is a METHOD
        {
            return $"Employee {EmployeeId}: {FullName} ({Department})";
        }
    }

    // Another CLASS demonstrating inheritance
    public class Manager : Employee      // <- This is a CLASS inheriting
from Employee
    {
        // Additional FIELDS for Manager
        private List<Employee> subordinates;  // <- This is a FIELD

        // Additional PROPERTIES for Manager
        public int TeamSize               // <- This is a PROPERTY
        {
            get { return subordinates.Count; }
        }

        public decimal BonusBudget { get; set; }  // <- This is a
PROPERTY

        // Constructor METHOD
        public Manager(int id, string first, string last, decimal sal,
DateTime hire, decimal bonus)
            : base(id, first, last, sal, hire)  // <- This is a METHOD
(constructor)
        {
            subordinates = new List<Employee>();
0
```

```
            BonusBudget = bonus;
        }

        // Manager-specific METHODS
        public void AddSubordinate(Employee employee)  // <- This is a
METHOD
        {
            if (employee != null && !subordinates.Contains(employee))
            {
                subordinates.Add(employee);
                Console.WriteLine($"{employee.FullName} added to
{FullName}'s team");
            }
        }

        public void RemoveSubordinate(Employee employee)  // <- This is a
METHOD
        {
            if (subordinates.Remove(employee))
            {
                Console.WriteLine($"{employee.FullName} removed from
{FullName}'s team");
            }
        }

        public List<Employee> GetTeamMembers()  // <- This is a METHOD
        {
            return new List<Employee>(subordinates);
        }

        // Override parent METHOD
        public override void DisplayInfo()   // <- This is a METHOD
(overridden)
        {
            base.DisplayInfo(); // Call parent method
            Console.WriteLine($"Team Size: {TeamSize}");
            Console.WriteLine($"Bonus Budget: ${BonusBudget:F2}");
            if (subordinates.Count > 0)
            {
                Console.WriteLine("Team Members:");
0
```

```
                foreach (var emp in subordinates)
                {
                    Console.WriteLine($"  - {emp.FullName}");
                }
            }
        }
    }
}

// Demonstration program
namespace MyCompany.Demo  // <- Another NAMESPACE
{
    using System;
    using MyCompany.EmployeeManagement;

    public class Program     // <- This is a CLASS
    {
        // Static METHOD - entry point
        public static void Main()  // <- This is a METHOD
        {
            Console.WriteLine("=== C# Concepts Demonstration ===\n");

            DemonstrateAssembly();
            DemonstrateNamespace();
            DemonstrateClassAndObjects();
            DemonstrateFieldsAndProperties();
            DemonstrateMethods();
        }

        public static void DemonstrateAssembly()  // <- This is a METHOD
        {
            Console.WriteLine("1. ASSEMBLY:");
            Console.WriteLine("   - This entire compiled program is an
ASSEMBLY");
            Console.WriteLine("   - Assembly contains all compiled code
(.exe or .dll)");
            Console.WriteLine("   - Provides security and versioning
boundary");
            Console.WriteLine($"   - Current assembly:
{System.Reflection.Assembly.GetExecutingAssembly().GetName().Name}");
```

0

```
            Console.WriteLine();
        }

        public static void DemonstrateNamespace()  // <- This is a METHOD
        {
            Console.WriteLine("2. NAMESPACE:");
            Console.WriteLine("   - MyCompany.EmployeeManagement contains
Employee and Manager classes");
            Console.WriteLine("   - MyCompany.Demo contains this Program
class");
            Console.WriteLine("   - Namespaces prevent name conflicts");
            Console.WriteLine("   - Organize related classes logically");
            Console.WriteLine();
        }

        public static void DemonstrateClassAndObjects()  // <- This is a
METHOD
        {
            Console.WriteLine("3. CLASSES AND OBJECTS:");

            // Create objects from classes
            Employee emp1 = new Employee(101, "John", "Doe", 50000, new
DateTime(2020, 3, 15));
            Employee emp2 = new Employee(102, "Jane", "Smith", 55000, new
DateTime(2019, 7, 22));
            Manager mgr1 = new Manager(201, "Bob", "Johnson", 75000, new
DateTime(2018, 1, 10), 10000);

            Console.WriteLine("   âœ“ Created Employee and Manager
objects from classes");
            Console.WriteLine($"   - {emp1}");
            Console.WriteLine($"   - {emp2}");
            Console.WriteLine($"   - {mgr1}");
            Console.WriteLine();
        }

        public static void DemonstrateFieldsAndProperties()  // <- This
is a METHOD
        {
            Console.WriteLine("4. FIELDS vs PROPERTIES:");
0
```

```
            Employee employee = new Employee(103, "Alice", "Brown",
60000, new DateTime(2021, 5, 1));

            Console.WriteLine("   FIELDS (private, internal storage):");
            Console.WriteLine("   - employeeId, firstName, lastName,
salary, etc.");
            Console.WriteLine("   - Not directly accessible from outside
the class");
            Console.WriteLine();

            Console.WriteLine("   PROPERTIES (public interface):");
            Console.WriteLine($"   - EmployeeId: {employee.EmployeeId}");
            Console.WriteLine($"   - FirstName: {employee.FirstName}");
            Console.WriteLine($"   - LastName: {employee.LastName}");
            Console.WriteLine($"   - FullName: {employee.FullName} (read-
only)");
            Console.WriteLine($"   - YearsOfService:
{employee.YearsOfService} (calculated)");

            // Demonstrate property validation
            try
            {
                employee.Salary = 65000;  // Valid
                Console.WriteLine($"   - Salary updated to:
${employee.Salary:F2}");

                // employee.Salary = -1000;  // Would throw exception
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine($"   - Property validation:
{ex.Message}");
            }

            Console.WriteLine();
        }

        public static void DemonstrateMethods()  // <- This is a METHOD
        {
0
```

```
            Console.WriteLine("5. METHODS:");

            // Create objects
            Employee emp = new Employee(104, "Charlie", "Wilson", 58000,
new DateTime(2020, 8, 12));
            Manager mgr = new Manager(202, "Diana", "Taylor", 80000, new
DateTime(2017, 4, 3), 15000);

            Console.WriteLine("   Instance Methods:");

            // Call instance methods
            emp.AddSkill("C#");
            emp.AddSkill("SQL");
            emp.AddSkill("JavaScript");

            emp.GiveRaise(5);

            Console.WriteLine();
            Console.WriteLine("   Method with return value:");
            var skills = emp.GetSkills();
            Console.WriteLine($"   {emp.FullName} has {skills.Count}
skills: {string.Join(", ", skills)}");

            Console.WriteLine();
            Console.WriteLine("   Static Method:");
            Employee tempEmp = Employee.CreateTemporaryEmployee("Temp",
"Worker");
            Console.WriteLine($"   Created temporary employee:
{tempEmp}");

            Console.WriteLine();
            Console.WriteLine("   Virtual Method Override:");
            mgr.AddSubordinate(emp);
            mgr.DisplayInfo();  // Calls overridden version

            Console.WriteLine();
        }
    }
}
```

0

Summary of Concepts:

Concept	Definition	Example	Purpose
Assembly	Compiled unit (.exe/.dll)	Entire application	Security, versioning, deployment
Namespace	Logical grouping of types	`MyCompany.EmployeeManagement`	Organization, prevent conflicts
Class	Blueprint for objects	`Employee`, `Manager`	Define structure and behavior
Field	Private data storage	`private string firstName`	Internal state storage
Property	Public interface to data	`public string FirstName { get; set; }`	Controlled access to data
Method	Function that defines behavior	`public void AddSkill(string skill)`	Define what objects can do

Key Relationships:

1. **Assembly** contains multiple **Namespaces**
2. **Namespace** contains multiple **Classes**
3. **Class** contains **Fields**, **Properties**, and **Methods**
4. **Properties** often provide access to **Fields**
5. **Methods** operate on **Fields** and **Properties**

Hierarchy Example:

```
Assembly: MyApplication.exe
|-- Namespace: MyCompany.EmployeeManagement
|   |-- Class: Employee
|   |   |-- Fields: employeeId, firstName, lastName
|   |   |-- Properties: EmployeeId, FirstName, LastName
|   |   |-- Methods: AddSkill(), GiveRaise(), DisplayInfo()
|   |-- Class: Manager (inherits from Employee)
|-- Namespace: MyCompany.Demo
    |-- Class: Program
        |-- Methods: Main(), DemonstrateAssembly()
```

This structure provides organization, encapsulation, and reusability in C# applications.

0

Exam Tips

1. **Practice Code Examples**: Type out the examples yourself
2. **Understand Patterns**: Know when to use abstract vs interface, DataReader vs DataSet
3. **Remember Architecture**: Know the layers and how they interact
4. **Security**: Understand authentication vs authorization
5. **Performance**: Know when to use async, parallel processing, proper data access

Your comprehensive exam preparation notes are now complete! This document covers all the topics your teacher specified with practical examples, comparisons, and detailed explanations. You can now compile this into a PDF for printing and study.

0