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

0

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.

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.

0

Key Concepts

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

0

Feature	.NET Framework	.NET Core	.NET 5+
Performance	Good	Better	Best
Deployment	Framework dependent	Self-contained options	Self-contained options

Architecture Diagram

```
â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€€
â”,        Your Application           â”,
â”œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€€
â”,    Base Class Library (BCL)       â”,
â”œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€€
â”,  Common Language Runtime (CLR)    â”,
â”œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€€
â”,       Operating System            â”,
â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€€
```

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

0

- **Git Integration**: Version control support

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
```

0

```
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)
```

0

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

// Using Convert class
string numberString = "123";
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})";
    }
}

// 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

Feature	struct	enum
Mutability	Should be immutable	Immutable

Collections

Generic Collections (Recommended)

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
```

0

```
};

// 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");
}

// 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"
```

0

```
// 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

```
// 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}");
}

// 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

0

- () - 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);

// 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);
```

0

```
    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)
{
    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);
```

0

```
// 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)
{
    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;
    }
```

0

```
    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");
    }
}
```

0

```
// 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");
    }
}

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
{
```

0

```
    // 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");
    }
}

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;
    }
}
```

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}");
    }
```

0

```
    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" } };
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

0

- 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();
    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;
```

0

```
    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
    }

    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

0

```
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();
}

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)
```

0

```
    {
        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");
    }

    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");
```

0

```
    }

    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
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

0

- 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}");
    }
}

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()
    {
```

0

```
        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

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;
```

0

```
        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}");
    }
```

0

```
    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");
    }
}

// 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.

0 **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; }
        // 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
    }
```

0

```
    // 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;
    }

    // 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)
        {
            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

0

```
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; }

    // 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
```

0

```
    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)
        {
```

0

```
            // 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);

            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}");
        }
```

0

```
        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
{
    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:
0
```

```
{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;
    }

    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.");
        }
```

0

```
        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
        }
    }

    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);
        }
    }
```

0

```
    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($"File content length:
{content.Length}");
                // fileStream and reader are automatically disposed
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"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);
```

0

```
            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
        {
            // 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
0
```

```
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);
```

0

```
            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();
    }
}
```

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 =>
        {
            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}");
```

0

```
    }

    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
        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;
    }
}
```

Thread-Safe Collections

```
public class ThreadSafeCollectionsExample
{
    public static void ConcurrentCollectionsExample()
    {
        // ConcurrentBag - Thread-safe collection of objects
```

0

```
        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");

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

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

0

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

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

0

```
    }
}
```

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}"); //
Should be 10000
    }

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

    public static async Task SemaphoreExample()
    {
```

0

```
        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

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

DataReader vs DataSet

DataReader (Connected Architecture)

0

```
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");
```

0

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

0

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

0

```
statement
            }
            catch (SqlException ex)
            {
                Console.WriteLine($"SQL Error: {ex.Message}");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"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))
                {
0
```

```
                    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")
                            };
                        }
                    }
                }
            }
            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);
```

0

```
                    return command.ExecuteNonQuery() > 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

0

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

    [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:");
            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

0

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

```
<!-- 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"
                    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();
```

0

```
        }

        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
        }
    }
0
```

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

0

```
    <StackPanel Margin="20">

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

        <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>
```

0

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

        // Set DataContext
        DataContext = new Person
        {
            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"
```

0

```
                    FontWeight="Bold"
                    FontSize="16"/>

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

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

    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);
0       SearchCommand = new RelayCommand(Search);
```

```
    }

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

    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)
0
```

```
    {
        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();
```

0

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

    private void LoadEmployees()
    {
        // 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

0

```
â”Œâ”€â”€â”€â
```

```
    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; }
    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
        {
```

0

```
            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)
    {
        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);
```

0

```
        }
        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)
    {
        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
```

0

```
    [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)
        {
            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;
```

0

```
        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
        {
            // 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

0

```
// 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)
    .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
0
```

```
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

// Custom middleware
app.UseRequestLogging();

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

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; }
```

0

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

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

<h1>Products</h1>

<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>
```

0

```
            <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>
                    <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();
        }

        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;
```

0

```
    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)
        };
```

0

```
        // 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)
        };

        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)
    {
        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

0

```
# 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

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:
```

0

```
    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

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"
```

0

```
<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++;
    }
}
```

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; }
```

0

```
    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; }
    [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
0
```

```
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)
    {
        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

0

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>
```

0

```
        <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>
            </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; } = "";
    }
}
```

0

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" />
        <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
    }

    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; }
0
```

```
    }
}
```

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; }
    }
}
```

```
// 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>
    }
```

0

```
    @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"
              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();
```

0

```
        }
    }

    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();
}

// 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;
    }
}
```

0

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

```
// 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();
    }
```

0

```
    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)
    {
        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;
            OnPropertyChanged();
        }
    }

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

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

0

```
    }

    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()
    {
        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;
        }
    }
```

0

```
    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);
        }
    }

    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

0

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>

    <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"
0
```

```
                            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; }

    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; }
```

0

```
    [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
"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>()
```

0

```
            .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) ||
0
```

```
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("..");
    }
}

// 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
{
0
```

```
    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);
    }

    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));
```

0

```
    }
}
```

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
- 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

0

- 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 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