Day 3: Inheritance & Polymorphism (2 Hours)

Learning Objectives

By the end of today, you will:

- Understand inheritance and how to create class hierarchies
- Master the virtual, override, and base keywords
- Implement polymorphism effectively
- Know when to use method overriding vs method overloading
- Design proper inheritance relationships

Part 1: Inheritance Fundamentals (45 minutes)

What is Inheritance?

Inheritance allows you to create new classes based on existing classes, inheriting their properties and methods.

```csharp
```

```
// Base class (Parent class)
public class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Species { get; set; }

    public Animal(string name, int age, string species)
    {
        Name = name;
        Age = age;
        Species = species;
    }

    public virtual void MakeSound()
    {
        Console.WriteLine($"{Name} makes a generic animal sound.");
    }

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

    public void Eat(string food)
    {
        Console.WriteLine($"{Name} is eating {food}.");
    }
}

// Derived class (Child class)
public class Dog : Animal
{
    public string Breed { get; set; }

    // Constructor calls base constructor
    public Dog(string name, int age, string breed) : base(name, age, "Dog")
    {
        Breed = breed;
    }

    // Override the virtual method
    public override void MakeSound()
```

```
    {
        Console.WriteLine($"{Name} barks: Woof! Woof!");
    }

    // New method specific to Dog
    public void Fetch()
    {
        Console.WriteLine($"{Name} is fetching the ball!");
    }
}

public class Cat : Animal
{
    public bool IsIndoor { get; set; }

    public Cat(string name, int age, bool isIndoor) : base(name, age, "Cat")
    {
        IsIndoor = isIndoor;
    }

    public override void MakeSound()
    {
        Console.WriteLine($"{Name} meows: Meow! Meow!");
    }

    public void Purr()
    {
        Console.WriteLine($"{Name} is purring contentedly.");
    }
}
```

Using Inheritance

csharp

```
class Program
{
    static void Main()
    {
        // Create instances
        Dog myDog = new Dog("Buddy", 3, "Golden Retriever");
        Cat myCat = new Cat("Whiskers", 2, true);

        // Inherited methods work
        myDog.Sleep();
        myDog.Eat("dog food");

        // Overridden methods use child implementation
        myDog.MakeSound(); // "Buddy barks: Woof! Woof!"
        myCat.MakeSound(); // "Whiskers meows: Meow! Meow!"

        // Child-specific methods
        myDog.Fetch();
        myCat.Purr();

        // Access inherited and new properties
        Console.WriteLine($"Dog breed: {myDog.Breed}");
        Console.WriteLine($"Cat is indoor: {myCat.IsIndoor}");
    }
}
```

The `base` Keyword

Use `base` to access parent class members:

csharp

```
public class Vehicle
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }

    public Vehicle(string make, string model, int year)
    {
        Make = make;
        Model = model;
        Year = year;
    }

    public virtual void Start()
    {
        Console.WriteLine($"The {Year} {Make} {Model} is starting...");
    }

    public virtual void DisplayInfo()
    {
        Console.WriteLine($"Vehicle: {Year} {Make} {Model}");
    }
}

public class Car : Vehicle
{
    public int NumberOfDoors { get; set; }

    public Car(string make, string model, int year, int doors)
        : base(make, model, year)
    {
        NumberOfDoors = doors;
    }

    public override void Start()
    {
        // Call parent method first
        base.Start();
        Console.WriteLine("Engine started. Ready to drive!");
    }

    public override void DisplayInfo()
    {
```

```
        // Call parent method and extend it
        base.DisplayInfo();
        Console.WriteLine($"Doors: {NumberOfDoors}");
    }
}

public class Motorcycle : Vehicle
{
    public bool HasSidecar { get; set; }

    public Motorcycle(string make, string model, int year, bool hasSidecar)
        : base(make, model, year)
    {
        HasSidecar = hasSidecar;
    }

    public override void Start()
    {
        base.Start();
        Console.WriteLine("Kick-started! Ready to ride!");
    }
}
```

Part 2: Polymorphism in Action (30 minutes)

What is Polymorphism?

Polymorphism allows objects of different types to be treated as objects of a common base type, but still behave according to their actual type.

csharp

```
public abstract class Shape
{
    public string Name { get; set; }

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

    // Virtual method - can be overridden
    public virtual double CalculateArea()
    {
        return 0;
    }

    // Abstract method - must be overridden
    public abstract double CalculatePerimeter();

    public virtual void Display()
    {
        Console.WriteLine($"Shape: {Name}");
        Console.WriteLine($"Area: {CalculateArea():F2}");
        Console.WriteLine($"Perimeter: {CalculatePerimeter():F2}");
    }
}

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

    public Circle(double radius) : base("Circle")
    {
        Radius = radius;
    }

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

    public override double CalculatePerimeter()
    {
        return 2 * Math.PI * Radius;
```

```
    }
}

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

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

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

    public override double CalculatePerimeter()
    {
        return 2 * (Width + Height);
    }
}

public class Triangle : Shape
{
    public double Base { get; set; }
    public double Height { get; set; }
    public double SideA { get; set; }
    public double SideB { get; set; }

    public Triangle(double baseLength, double height, double sideA, double sideB)
        : base("Triangle")
    {
        Base = baseLength;
        Height = height;
        SideA = sideA;
        SideB = sideB;
    }

    public override double CalculateArea()
    {
        return 0.5 * Base * Height;
```

```
    }

    public override double CalculatePerimeter()
    {
        return Base + SideA + SideB;
    }
}
```

Polymorphism in Action

csharp

```
class Program
{
    static void Main()
    {
        // Create array of different shapes
        Shape[] shapes = {
            new Circle(5),
            new Rectangle(4, 6),
            new Triangle(3, 4, 5, 5),
            new Circle(2.5),
            new Rectangle(2, 8)
        };

        // Polymorphism: Each shape behaves according to its actual type
        Console.WriteLine("=== All Shapes ===");
        foreach (Shape shape in shapes)
        {
            shape.Display(); // Calls the correct override for each type
            Console.WriteLine();
        }

        // Calculate total area
        double totalArea = 0;
        foreach (Shape shape in shapes)
        {
            totalArea += shape.CalculateArea();
        }
        Console.WriteLine($"Total area of all shapes: {totalArea:F2}");

        // Find largest shape
        Shape largest = shapes[0];
        foreach (Shape shape in shapes)
        {
            if (shape.CalculateArea() > largest.CalculateArea())
            {
                largest = shape;
            }
        }
        Console.WriteLine($"Largest shape: {largest.Name} with area {largest.CalculateArea():F2}");
    }
}
```

Method Overloading vs Method Overriding

csharp

```
public class Calculator
{
    // Method Overloading - same name, different parameters
    public int Add(int a, int b)
    {
        return a + b;
    }

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

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

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

    // Virtual method for overriding
    public virtual double Calculate(double value)
    {
        return value;
    }
}

public class ScientificCalculator : Calculator
{
    // Method Overriding - same signature, different implementation
    public override double Calculate(double value)
    {
        // Add scientific computation
        return Math.Pow(value, 2);
    }

    // Additional overloaded methods
    public double Add(double a, double b, double c)
    {
        return a + b + c;
```

```
    }
}
```

Part 3: Advanced Inheritance Concepts (30 minutes)

Protected Access Modifier

csharp

```
public class Employee
{
    public string Name { get; set; }
    protected decimal baseSalary; // Accessible by derived classes
    private string ssn; // Only accessible within this class

    public Employee(string name, decimal baseSalary, string ssn)
    {
        Name = name;
        this.baseSalary = baseSalary;
        this.ssn = ssn;
    }

    public virtual decimal CalculateSalary()
    {
        return baseSalary;
    }

    protected virtual decimal CalculateBonus()
    {
        return baseSalary * 0.1m; // 10% bonus
    }
}

public class Manager : Employee
{
    public int TeamSize { get; set; }

    public Manager(string name, decimal baseSalary, string ssn, int teamSize)
        : base(name, baseSalary, ssn)
    {
        TeamSize = teamSize;
    }

    public override decimal CalculateSalary()
    {
        // Can access protected members
        return baseSalary + CalculateBonus() + (TeamSize * 1000);
    }

    protected override decimal CalculateBonus()
    {
        // Managers get 20% bonus
```

```
        return baseSalary * 0.2m;
    }
}

public class Developer : Employee
{
    public string ProgrammingLanguage { get; set; }

    public Developer(string name, decimal baseSalary, string ssn, string language)
        : base(name, baseSalary, ssn)
    {
        ProgrammingLanguage = language;
    }

    public override decimal CalculateSalary()
    {
        decimal techBonus = ProgrammingLanguage.ToLower() switch
        {
            "c#" => 5000,
            "python" => 4000,
            "javascript" => 3000,
            _ => 2000
        };

        return baseSalary + CalculateBonus() + techBonus;
    }
}
```

Sealed Classes and Methods

csharp

```
public class Animal
{
    public virtual void Move()
    {
        Console.WriteLine("Animal moves");
    }
}

public class Bird : Animal
{
    // Sealed method - cannot be overridden further
    public sealed override void Move()
    {
        Console.WriteLine("Bird flies");
    }

    public virtual void Sing()
    {
        Console.WriteLine("Bird sings");
    }
}

// Sealed class - cannot be inherited
public sealed class Penguin : Bird
{
    public override void Move()
    {
        // Error! Cannot override sealed method
        // Console.WriteLine("Penguin waddles");
    }

    public override void Sing()
    {
        Console.WriteLine("Penguin doesn't sing much");
    }
}

// This would cause an error - cannot inherit from sealed class
// public class EmperorPenguin : Penguin { }
```

Part 4: Hands-on Lab - Build an Inheritance Hierarchy (45 minutes)

Exercise: Library Management System

Create a library management system with the following hierarchy:

1. **Base class**: `LibraryItem`
   - Properties: Title, Author, ISBN, IsAvailable
   - Methods: CheckOut(), Return(), DisplayInfo()
2. **Derived classes**: `Book`, `DVD`, `Magazine`
   - Each with specific properties and overridden methods
3. **Further inheritance**: `EBook` inherits from `Book`

Solution:

```csharp
```

```
// Base class
public abstract class LibraryItem
{
    public string Title { get; set; }
    public string Author { get; set; }
    public string ISBN { get; set; }
    public bool IsAvailable { get; protected set; }
    public DateTime CheckoutDate { get; protected set; }
    public DateTime DueDate { get; protected set; }

    protected LibraryItem(string title, string author, string isbn)
    {
        Title = title;
        Author = author;
        ISBN = isbn;
        IsAvailable = true;
    }

    public virtual bool CheckOut()
    {
        if (IsAvailable)
        {
            IsAvailable = false;
            CheckoutDate = DateTime.Now;
            DueDate = DateTime.Now.AddDays(GetLoanPeriodDays());
            Console.WriteLine($"{Title} has been checked out. Due: {DueDate:MM/dd/yyyy}");
            return true;
        }
        Console.WriteLine($"{Title} is not available for checkout.");
        return false;
    }

    public virtual void Return()
    {
        if (!IsAvailable)
        {
            IsAvailable = true;
            Console.WriteLine($"{Title} has been returned.");

            if (DateTime.Now > DueDate)
            {
                int daysLate = (DateTime.Now - DueDate).Days;
                Console.WriteLine($"Item was {daysLate} days late. Late fee: ${CalculateLateFee(daysLate):F2}");
```

```
        }
    }
    else
    {
        Console.WriteLine($"{Title} was not checked out.");
    }
}

public virtual void DisplayInfo()
{
    Console.WriteLine($"Title: {Title}");
    Console.WriteLine($"Author: {Author}");
    Console.WriteLine($"ISBN: {ISBN}");
    Console.WriteLine($"Status: {(IsAvailable ? "Available" : "Checked Out")}");
    if (!IsAvailable)
    {
        Console.WriteLine($"Due Date: {DueDate:MM/dd/yyyy}");
    }
}

protected abstract int GetLoanPeriodDays();

protected virtual decimal CalculateLateFee(int daysLate)
{
    return daysLate * 0.50m; // 50 cents per day
}
}

// Book class
public class Book : LibraryItem
{
public int Pages { get; set; }
public string Genre { get; set; }

public Book(string title, string author, string isbn, int pages, string genre)
    : base(title, author, isbn)
{
    Pages = pages;
    Genre = genre;
}

protected override int GetLoanPeriodDays()
{
    return 14; // 2 weeks for books
```

```
    }

    public override void DisplayInfo()
    {
        base.DisplayInfo();
        Console.WriteLine($"Pages: {Pages}");
        Console.WriteLine($"Genre: {Genre}");
        Console.WriteLine($"Type: Book");
    }
}

// DVD class
public class DVD : LibraryItem
{
    public int RuntimeMinutes { get; set; }
    public string Rating { get; set; }

    public DVD(string title, string director, string isbn, int runtime, string rating)
        : base(title, director, isbn)
    {
        RuntimeMinutes = runtime;
        Rating = rating;
    }

    protected override int GetLoanPeriodDays()
    {
        return 3; // 3 days for DVDs
    }

    protected override decimal CalculateLateFee(int daysLate)
    {
        return daysLate * 2.00m; // $2 per day for DVDs
    }

    public override void DisplayInfo()
    {
        base.DisplayInfo();
        Console.WriteLine($"Director: {Author}"); // Author field used for Director
        Console.WriteLine($"Runtime: {RuntimeMinutes} minutes");
        Console.WriteLine($"Rating: {Rating}");
        Console.WriteLine($"Type: DVD");
    }
}
```

```
// Magazine class
public class Magazine : LibraryItem
{
    public string IssueNumber { get; set; }
    public DateTime PublishDate { get; set; }

    public Magazine(string title, string publisher, string isbn, string issueNumber, DateTime publishDate)
        : base(title, publisher, isbn)
    {
        IssueNumber = issueNumber;
        PublishDate = publishDate;
    }

    protected override int GetLoanPeriodDays()
    {
        return 7; // 1 week for magazines
    }

    public override void DisplayInfo()
    {
        base.DisplayInfo();
        Console.WriteLine($"Publisher: {Author}"); // Author field used for Publisher
        Console.WriteLine($"Issue: {IssueNumber}");
        Console.WriteLine($"Publish Date: {PublishDate:MM/dd/yyyy}");
        Console.WriteLine($"Type: Magazine");
    }
}

// EBook inherits from Book
public class EBook : Book
{
    public string FileFormat { get; set; }
    public double FileSizeMB { get; set; }

    public EBook(string title, string author, string isbn, int pages, string genre,
                 string fileFormat, double fileSizeMB)
        : base(title, author, isbn, pages, genre)
    {
        FileFormat = fileFormat;
        FileSizeMB = fileSizeMB;
    }

    protected override int GetLoanPeriodDays()
    {
```

```
        return 21; // 3 weeks for eBooks
    }

    public override bool CheckOut()
    {
        // EBooks are always available (digital copies)
        Console.WriteLine($"EBook '{Title}' has been downloaded. Access expires in {GetLoanPeriodDays()} days.");
        return true;
    }

    public override void Return()
    {
        Console.WriteLine($"EBook '{Title}' access has expired automatically.");
    }

    public override void DisplayInfo()
    {
        base.DisplayInfo();
        Console.WriteLine($"File Format: {FileFormat}");
        Console.WriteLine($"File Size: {FileSizeMB:F2} MB");
        Console.WriteLine($"Type: EBook");
    }
}

// Library management class
public class Library
{
    private List<LibraryItem> items;

    public Library()
    {
        items = new List<LibraryItem>();
    }

    public void AddItem(LibraryItem item)
    {
        items.Add(item);
        Console.WriteLine($"Added '{item.Title}' to the library.");
    }

    public void DisplayAllItems()
    {
        Console.WriteLine("\n=== Library Inventory ===");
        foreach (var item in items)
```

```
        {
            item.DisplayInfo();
            Console.WriteLine(new string('-', 30));
        }
    }

    public void CheckOutItem(string title)
    {
        var item = items.FirstOrDefault(i => i.Title.Equals(title, StringComparison.OrdinalIgnoreCase));
        if (item != null)
        {
            item.CheckOut();
        }
        else
        {
            Console.WriteLine($"Item '{title}' not found in library.");
        }
    }

    public void ReturnItem(string title)
    {
        var item = items.FirstOrDefault(i => i.Title.Equals(title, StringComparison.OrdinalIgnoreCase));
        if (item != null)
        {
            item.Return();
        }
        else
        {
            Console.WriteLine($"Item '{title}' not found in library.");
        }
    }
}

// Test the system
class Program
{
    static void Main()
    {
        Library library = new Library();

        // Add different types of items
        library.AddItem(new Book("The Great Gatsby", "F. Scott Fitzgerald", "978-0743273565", 180, "Fiction"));
        library.AddItem(new DVD("Inception", "Christopher Nolan", "B002ZG981E", 148, "PG-13"));
        library.AddItem(new Magazine("National Geographic", "National Geographic Society", "NG202301", "January 2023
```

```
        library.AddItem(new EBook("Clean Code", "Robert C. Martin", "978-0132350884", 464, "Technology", "PDF", 5.2));

        // Display all items
        library.DisplayAllItems();

        // Test checkout and return
        Console.WriteLine("\n=== Testing Checkout/Return ===");
        library.CheckOutItem("The Great Gatsby");
        library.CheckOutItem("Inception");
        library.CheckOutItem("Clean Code");

        Console.WriteLine("\n=== After Checkouts ===");
        library.DisplayAllItems();

        Console.WriteLine("\n=== Testing Returns ===");
        library.ReturnItem("The Great Gatsby");
        library.ReturnItem("Inception");
    }
}
```

Day 3 Summary

What You've Learned:

1. **Inheritance**: Creating class hierarchies with `:` syntax
2. **Method Overriding**: Using `virtual` and `override` keywords
3. **Polymorphism**: Different objects behaving according to their actual type
4. **The `base` keyword**: Accessing parent class members
5. **Protected access**: Sharing with derived classes but hiding from others
6. **Abstract classes**: Defining contracts for derived classes

Key Principles:

- **IS-A Relationship**: Use inheritance when there's a clear "is-a" relationship
- **Polymorphism**: Write code that works with base types but behaves according to actual types
- **Method Overriding**: Customize behavior in derived classes
- **Proper Access Control**: Use protected for derived class access

Best Practices:

- Use inheritance sparingly - favor composition when possible
- Make base class methods virtual if they should be overridable
- Use abstract classes when you want to enforce implementation of certain methods
- Always call base constructors when needed

Tomorrow's Preview:

Day 4 will cover **Abstract Classes & Interfaces** - defining contracts and achieving multiple inheritance through interfaces.

Additional Practice Exercises

1. Create a `Vehicle` hierarchy with `Car`, `Truck`, and `Motorcycle`
2. Build a `Media Player` system with different file types
3. Design an `Employee` payroll system with different employee types

The key to mastering inheritance is understanding when to use it and how to design proper class hierarchies!