Day 4: Abstract Classes & Interfaces (2 Hours)

OOP Course

July 21, 2025

1 Learning Objectives

By the end of today, you will:

- Understand the difference between abstract classes and concrete classes
- Master interfaces and multiple inheritance
- Know when to use abstract classes versus interfaces
- Design systems using abstractions effectively

2 Part 1: Abstract Classes vs Concrete Classes (45 minutes)

2.1 What are Abstract Classes?

Abstract classes are classes that cannot be instantiated directly and are meant to serve as base classes for other classes. They can contain both implemented and abstract (unimplemented) methods.

2.2 Concrete Classes

Concrete classes are fully implemented classes that can be instantiated directly. They provide complete implementations for all their methods.

```
// Abstract class example
public abstract class Vehicle {
    public string Make { get; set; }
    public string Model { get; set; }

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

    // Abstract method - must be implemented by derived classes
    public abstract void StartEngine();

    // Concrete method
    public void DisplayInfo() {
```

```
        Console.WriteLine($"Vehicle: {Make} {Model}");
    }
}

// Concrete class inheriting from abstract class
public class Car : Vehicle {
    public int NumberOfDoors { get; set; }

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

    public override void StartEngine() {
        Console.WriteLine($"{Make} {Model} engine started with key
            ignition.");
    }
}

// Concrete class that cannot be inherited
public sealed class Motorcycle : Vehicle {
    public bool HasSidecar { get; set; }

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

    public override void StartEngine() {
        Console.WriteLine($"{Make} {Model} engine started with
            kick-start.");
    }
}
```

2.3 Key Differences

- Abstract classes cannot be instantiated; concrete classes can
- Abstract classes can have abstract methods; concrete classes must implement all methods
- Abstract classes are designed for inheritance; concrete classes may or may not be

3 Part 2: Interfaces and Multiple Inheritance (45 minutes)

3.1 What are Interfaces?

Interfaces define a contract of methods and properties that implementing classes must provide, without any implementation details.

```
// Interface definitions
public interface IDriveable {
    void Drive();
    void Stop();
    int Speed { get; set; }
}

public interface IFuelable {
    void Refuel();
    double FuelLevel { get; }
}

// Class implementing multiple interfaces
public class Car : Vehicle, IDriveable, IFuelable {
    public int NumberOfDoors { get; set; }
    public int Speed { get; set; }
    public double FuelLevel { get; private set; }

    public Car(string make, string model, int doors)
        : base(make, model) {
        NumberOfDoors = doors;
        FuelLevel = 100.0;
    }

    public override void StartEngine() {
        Console.WriteLine($"{Make} {Model} engine started.");
    }

    public void Drive() {
        Console.WriteLine($"{Make} {Model} is driving at {Speed}
           mph.");
    }

    public void Stop() {
        Console.WriteLine($"{Make} {Model} has stopped.");
        Speed = 0;
    }

    public void Refuel() {
        FuelLevel = 100.0;
        Console.WriteLine($"{Make} {Model} has been refueled.");
    }
}
```

3.2 Multiple Inheritance with Interfaces

C# does not support multiple class inheritance but allows classes to implement multiple interfaces, achieving similar flexibility.

```
class Program {
    static void Main() {
```

```
        Car car = new Car("Toyota", "Camry", 4);
        car.Speed = 60;
        car.DisplayInfo();
        car.StartEngine();
        car.Drive();
        car.Refuel();
        car.Stop();
    }
}
```

4 Part 3: When to Use Abstract Classes vs Interfaces (30 minutes)

4.1 Decision Guide

The following table helps decide when to use an abstract class versus an interface, with an example illustrating the choice.

Table 1: Decision Guide: Abstract Class vs Interface

Criterion	Use Abstract Class When	Use Interface When
Common Code	You have common code to share	You want to define a contract only
Class Relationship	Classes are closely related	Classes may be unrelated
Implementation	You want to provide default implementation	You need multiple inheritance
Members	You need protected members	You want maximum flexibility
Design Control	You control the inheritance hierarchy	You're designing for extensibility

Example: Payment System

For a payment system, use an abstract class `PaymentProcessor` to share common validation logic (e.g., checking amount) across related payment types like `CreditCardProcessor`. Use an interface `INotifiable` for notification behavior that can be implemented by unrelated classes (e.g., `EmailService`, `SMSService`).

```
// Abstract class for shared payment logic
public abstract class PaymentProcessor {
    public decimal Amount { get; set; }

    protected PaymentProcessor(decimal amount) {
        Amount = amount;
    }

    public bool ValidateAmount() {
```

```
        return Amount > 0;
    }

    public abstract bool Process();
}

// Interface for notification
public interface INotifiable {
    void Notify();
}

// Concrete class using both
public class CreditCardProcessor : PaymentProcessor, INotifiable {
    public string CardNumber { get; set; }

    public CreditCardProcessor(decimal amount, string cardNumber)
        : base(amount) {
        CardNumber = cardNumber;
    }

    public override bool Process() {
        if (ValidateAmount()) {
            Console.WriteLine($"Processing ${Amount:F2} via card
                {CardNumber}");
            return true;
        }
        return false;
    }

    public void Notify() {
        Console.WriteLine("Payment confirmation sent.");
    }
}
```

5 Part 4: Practice - Design with Abstractions (30 minutes)

5.1 Exercise: Notification System

Design a notification system with the following requirements:

1. Abstract class `Notification` with properties `Message`, `Recipient`, and abstract method `Send()`
2. Interface `ILoggable` with method `Log()`
3. Concrete classes: `EmailNotification`, `SMSNotification`, `PushNotification`

5.2 Solution

```
// Abstract class
public abstract class Notification {
    public string Message { get; set; }
```

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

    protected Notification(string message, string recipient) {
        Message = message;
        Recipient = recipient;
    }

    public abstract bool Send();
    public virtual void Preview() {
        Console.WriteLine($"Preview: To {Recipient}: {Message}");
    }
}

// Interface
public interface ILoggable {
    void Log();
}

// Concrete classes
public class EmailNotification : Notification, ILoggable {
    public string Subject { get; set; }

    public EmailNotification(string message, string recipient,
       string subject)
        : base(message, recipient) {
        Subject = subject;
    }

    public override bool Send() {
        Console.WriteLine($"Sending email to {Recipient} with
           subject: {Subject}");
        Console.WriteLine($"Message: {Message}");
        return true;
    }

    public void Log() {
        Console.WriteLine($"Logged email notification to
           {Recipient} at {DateTime.Now}");
    }
}

public class SMSNotification : Notification, ILoggable {
    public string PhoneNumber { get; set; }

    public SMSNotification(string message, string phoneNumber)
        : base(message, phoneNumber) {
        PhoneNumber = phoneNumber;
    }

    public override bool Send() {
```

```
        Console.WriteLine($"Sending SMS to {PhoneNumber}: {Message}");
        return true;
    }

    public void Log() {
        Console.WriteLine($"Logged SMS notification to {PhoneNumber} at {DateTime.Now}");
    }
}

public class PushNotification : Notification {
    public string DeviceId { get; set; }

    public PushNotification(string message, string recipient, string deviceId)
        : base(message, recipient) {
        DeviceId = deviceId;
    }

    public override bool Send() {
        Console.WriteLine($"Sending push notification to device {DeviceId} for {Recipient}");
        Console.WriteLine($"Message: {Message}");
        return true;
    }
}

// Test program
class Program {
    static void Main() {
        Notification[] notifications = {
            new EmailNotification("Meeting tomorrow", "user@example.com", "Team Meeting"),
            new SMSNotification("Your code: 123456", "+1234567890"),
            new PushNotification("New message!", "user123", "device789")
        };

        foreach (Notification notification in notifications) {
            notification.Preview();
            notification.Send();
            if (notification is ILoggable loggable) {
                loggable.Log();
            }
            Console.WriteLine();
        }
    }
}
```

6 Day 4 Summary

6.1 What You've Learned

1. Abstract classes provide partial implementations and cannot be instantiated
2. Interfaces define contracts without implementation
3. Multiple interfaces can be implemented for flexible design
4. When to use abstract classes (shared code, IS-A relationships) vs interfaces (unrelated classes, contracts)

6.2 Best Practices

- Use abstract classes for closely related classes with shared implementation
- Use interfaces for loosely coupled behaviors across different class types
- Keep interfaces focused on specific behaviors
- Avoid deep inheritance hierarchies

6.3 Tomorrow's Preview

Day 5 will cover **Advanced Features** including generics, delegates, events, and exception handling.