Day 2: Encapsulation & Properties (2 Hours)

Learning Objectives

By the end of today, you will:

- Understand the difference between fields and properties
- Master auto-implemented properties
- Learn when and how to use custom getters and setters
- Understand static members and their use cases
- Apply proper encapsulation principles

Part 1: Properties vs Fields (45 minutes)

The Problem with Public Fields

Yesterday we used public fields, but this can cause issues:

csharp

```csharp
public class Person
{
    public string Name;
    public int Age;  // What if someone sets Age to -5?
    public string Email;
}

// Problems:
Person person = new Person();
person.Age = -100;  // This is invalid but allowed!
person.Name = "";   // Empty name might not be desired
```

Properties: A Better Solution

Properties provide controlled access to fields:

csharp

```
public class Person
{
    // Private fields (backing fields)
    private string name;
    private int age;
    private string email;

    // Properties with custom logic
    public string Name
    {
        get { return name; }
        set
        {
            if (!string.IsNullOrWhiteSpace(value))
                name = value;
            else
                throw new ArgumentException("Name cannot be empty");
        }
    }

    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0 && value <= 150)
                age = value;
            else
                throw new ArgumentException("Age must be between 0 and 150");
        }
    }

    public string Email
    {
        get { return email; }
        set
        {
            if (IsValidEmail(value))
                email = value;
            else
                throw new ArgumentException("Invalid email format");
        }
    }
```

```
    // Helper method
    private bool IsValidEmail(string email)
    {
        return email.Contains("@") && email.Contains(".");
    }
}
```

Different Property Types

csharp

```
public class BankAccount
{
    private decimal balance;
    private string accountNumber;
    private DateTime creationDate;

    // Read-write property
    public string AccountHolder { get; set; }

    // Read-only property
    public string AccountNumber
    {
        get { return accountNumber; }
    }

    // Write-only property (rare, but possible)
    public string Pin
    {
        set { /* Store encrypted pin */ }
    }

    // Computed property (no backing field)
    public int AccountAge
    {
        get { return (DateTime.Now - creationDate).Days; }
    }

    // Property with validation
    public decimal Balance
    {
        get { return balance; }
        private set { balance = value; } // Private setter
    }
}
```

Expression-Bodied Properties (C# 6.0+)

csharp

```
public class Circle
{
    private double radius;

    public double Radius
    {
        get => radius;
        set => radius = value > 0 ? value : throw new ArgumentException("Radius must be positive");
    }

    // Read-only expression property
    public double Area => Math.PI * radius * radius;
    public double Circumference => 2 * Math.PI * radius;
}
```

Part 2: Auto-Implemented Properties (30 minutes)

Simple Auto-Properties

When you don't need custom logic, use auto-implemented properties:

csharp

```csharp
public class Product
{
    // Auto-implemented properties
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }

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

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

    public Product(string name, decimal price, string category)
    {
        Name = name;
        Price = price;
        Category = category;
        CreatedDate = DateTime.Now;
    }
}
```

Property Initializers

csharp

```
public class User
{
    public string Username { get; set; }
    public string Email { get; set; }
    public List<string> Roles { get; set; } = new List<string>();
    public DateTime LastLogin { get; set; } = DateTime.Now;
    public bool IsActive { get; set; } = true;
    public int LoginAttempts { get; private set; } = 0;

    public void RecordLoginAttempt()
    {
        LoginAttempts++;
        if (LoginAttempts > 3)
        {
            IsActive = false;
        }
    }
}
```

When to Use Auto-Properties vs Custom Properties

csharp

```
public class Employee
{
    // Auto-property - simple data storage
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime HireDate { get; set; }

    // Custom property - with validation
    private decimal salary;
    public decimal Salary
    {
        get => salary;
        set => salary = value >= 0 ? value : throw new ArgumentException("Salary cannot be negative");
    }

    // Computed property - derived from other properties
    public string FullName => $"{FirstName} {LastName}";
    public int YearsOfService => (DateTime.Now - HireDate).Days / 365;
}
```

Part 3: Static Members (30 minutes)

Understanding Static

Static members belong to the class itself, not to any instance:

```csharp
public class MathHelper
{
    // Static field - shared across all instances
    public static readonly double PI = 3.14159;

    // Static property
    public static DateTime ServerStartTime { get; private set; }

    // Static constructor - runs once when class is first used
    static MathHelper()
    {
        ServerStartTime = DateTime.Now;
        Console.WriteLine("MathHelper class initialized");
    }

    // Static methods
    public static double CalculateCircleArea(double radius)
    {
        return PI * radius * radius;
    }

    public static double CalculateDistance(double x1, double y1, double x2, double y2)
    {
        return Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2));
    }
}

// Usage - no need to create instances
double area = MathHelper.CalculateCircleArea(5.0);
double distance = MathHelper.CalculateDistance(0, 0, 3, 4);
```

Static vs Instance Members

```csharp
```

```
public class Counter
{
    // Static field - shared by all instances
    private static int totalInstances = 0;

    // Instance field - unique to each instance
    private int instanceValue;

    // Static property
    public static int TotalInstances => totalInstances;

    // Instance property
    public int Value
    {
        get => instanceValue;
        set => instanceValue = value;
    }

    // Constructor
    public Counter()
    {
        totalInstances++; // Increment shared counter
        instanceValue = 0; // Initialize instance value
    }

    // Instance method
    public void Increment()
    {
        instanceValue++;
    }

    // Static method
    public static void ResetTotalCount()
    {
        totalInstances = 0;
    }
}

// Usage
Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

```
Console.WriteLine($"Total instances: {Counter.TotalInstances}"); // 3

c1.Increment();
c2.Increment();
c2.Increment();

Console.WriteLine($"c1 value: {c1.Value}"); // 1
Console.WriteLine($"c2 value: {c2.Value}"); // 2
Console.WriteLine($"c3 value: {c3.Value}"); // 0
```

Practical Example: Configuration Class

csharp

```
public static class AppConfig
{
    // Static properties for application settings
    public static string DatabaseConnectionString { get; set; }
    public static int MaxRetryAttempts { get; set; } = 3;
    public static bool EnableLogging { get; set; } = true;

    // Static method to load configuration
    public static void LoadConfig()
    {
        DatabaseConnectionString = "Server=localhost;Database=MyApp;";
        MaxRetryAttempts = 5;
        EnableLogging = true;
    }

    // Static method to validate configuration
    public static bool IsValidConfig()
    {
        return !string.IsNullOrEmpty(DatabaseConnectionString) &&
               MaxRetryAttempts > 0;
    }
}

// Usage throughout your application
AppConfig.LoadConfig();
if (AppConfig.IsValidConfig())
{
    // Use AppConfig.DatabaseConnectionString
}
```

Part 4: Practice - Create a Robust Class (45 minutes)

Exercise: Build a Student Management System

Create a `Student` class with the following requirements:

1. **Properties**:
   - StudentId (read-only, auto-generated)
   - Name (with validation)
   - Age (18-65 range)
   - Email (with basic validation)
   - GPA (0.0-4.0 range, calculated from grades)
   - Grades (List of integers 0-100)
2. **Static Members**:
   - Total number of students created
   - School name
3. **Methods**:
   - AddGrade(int grade)
   - GetAverageGrade()
   - GetLetterGrade()

Solution:

```csharp
```

```
public class Student
{
    // Static members
    private static int nextStudentId = 1000;
    private static int totalStudents = 0;
    public static string SchoolName { get; set; } = "Default University";
    public static int TotalStudents => totalStudents;

    // Private fields
    private string name;
    private int age;
    private string email;
    private List<int> grades;

    // Properties
    public int StudentId { get; private set; }

    public string Name
    {
        get => name;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("Name cannot be empty");
            name = value.Trim();
        }
    }

    public int Age
    {
        get => age;
        set
        {
            if (value < 18 || value > 65)
                throw new ArgumentException("Age must be between 18 and 65");
            age = value;
        }
    }

    public string Email
    {
        get => email;
        set
```

```
        {
            if (string.IsNullOrWhiteSpace(value) || !value.Contains("@"))
                throw new ArgumentException("Invalid email format");
            email = value.Trim().ToLower();
        }
    }

    public double GPA => grades.Count > 0 ? CalculateGPA() : 0.0;

    public List<int> Grades => new List<int>(grades); // Return copy to prevent external modification

    // Constructor
    public Student(string name, int age, string email)
    {
        StudentId = nextStudentId++;
        totalStudents++;
        grades = new List<int>();

        Name = name;
        Age = age;
        Email = email;
    }

    // Methods
    public void AddGrade(int grade)
    {
        if (grade < 0 || grade > 100)
            throw new ArgumentException("Grade must be between 0 and 100");
        grades.Add(grade);
    }

    public double GetAverageGrade()
    {
        return grades.Count > 0 ? grades.Average() : 0.0;
    }

    public string GetLetterGrade()
    {
        double average = GetAverageGrade();
        return average switch
        {
            >= 90 => "A",
            >= 80 => "B",
            >= 70 => "C",
```

```
            >= 60 => "D",
            _ => "F"
        };
    }

    private double CalculateGPA()
    {
        double average = GetAverageGrade();
        return average switch
        {
            >= 97 => 4.0,
            >= 93 => 3.7,
            >= 90 => 3.3,
            >= 87 => 3.0,
            >= 83 => 2.7,
            >= 80 => 2.3,
            >= 77 => 2.0,
            >= 73 => 1.7,
            >= 70 => 1.3,
            >= 67 => 1.0,
            >= 65 => 0.7,
            _ => 0.0
        };
    }

    public override string ToString()
    {
        return $"Student {StudentId}: {Name} (Age: {Age}, Email: {Email}, GPA: {GPA:F2})";
    }

    // Static methods
    public static void DisplaySchoolInfo()
    {
        Console.WriteLine($"School: {SchoolName}");
        Console.WriteLine($"Total Students: {TotalStudents}");
    }
}

// Test the Student class
class Program
{
    static void Main()
    {
        Student.SchoolName = "Tech University";
```

```
        try
        {
            Student student1 = new Student("Alice Johnson", 20, "alice@email.com");
            student1.AddGrade(95);
            student1.AddGrade(87);
            student1.AddGrade(92);

            Student student2 = new Student("Bob Smith", 22, "bob@email.com");
            student2.AddGrade(78);
            student2.AddGrade(85);
            student2.AddGrade(88);

            Console.WriteLine(student1);
            Console.WriteLine($"Average: {student1.GetAverageGrade():F2}");
            Console.WriteLine($"Letter Grade: {student1.GetLetterGrade()}");
            Console.WriteLine();

            Console.WriteLine(student2);
            Console.WriteLine($"Average: {student2.GetAverageGrade():F2}");
            Console.WriteLine($"Letter Grade: {student2.GetLetterGrade()}");
            Console.WriteLine();

            Student.DisplaySchoolInfo();
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

Day 2 Summary

What You've Learned:

1. **Properties**: Better than public fields, provide controlled access
2. **Auto-implemented Properties**: Simple syntax for basic properties
3. **Property Validation**: Ensuring data integrity
4. **Static Members**: Shared across all instances
5. **Encapsulation**: Hiding internal details, exposing controlled interface

Key Principles:

- **Data Validation**: Always validate input in property setters
- **Immutability**: Use private setters or read-only properties when appropriate
- **Static Usage**: Use static for utility functions and shared data
- **Encapsulation**: Private fields + Public properties = Controlled access

Best Practices:

- Use auto-properties for simple data storage
- Add validation in custom property setters
- Make properties read-only when they shouldn't change after initialization
- Use static members for class-level data and utility methods

Tomorrow's Preview:

Day 3 will cover **Inheritance & Polymorphism** - creating class hierarchies and enabling objects to behave differently based on their type.

Additional Practice Exercises

1. Create a `BankAccount` class with proper validation for balance operations
2. Build a `Product` class with pricing rules and discount calculations
3. Design a `Vehicle` class with static methods for vehicle statistics

The key to mastering properties is understanding when to use auto-properties vs custom properties with validation!