Day 1: C# OOP Foundation & Classes (2 Hours)

Learning Objectives

By the end of today, you will:

- Understand what classes and objects are
- Create your first C# class
- Work with fields and methods
- Understand constructors and object initialization
- Master access modifiers (public, private, protected, internal)

Part 1: Classes, Objects, and Fields (30 minutes)

What is Object-Oriented Programming?

OOP is a programming paradigm based on the concept of "objects" - entities that contain data (fields) and code (methods).

Classes vs Objects

- **Class**: A blueprint or template for creating objects
- **Object**: An instance of a class

Think of it like:

- **Class**: House blueprint
- **Object**: Actual house built from that blueprint

Your First C# Class

```csharp
```

```
// Simple class definition
public class Person
{
    // Fields (data)
    public string Name;
    public int Age;
    public string Email;
}

// Using the class
class Program
{
    static void Main()
    {
        // Creating objects (instances)
        Person person1 = new Person();
        person1.Name = "Alice";
        person1.Age = 25;
        person1.Email = "alice@email.com";

        Person person2 = new Person();
        person2.Name = "Bob";
        person2.Age = 30;
        person2.Email = "bob@email.com";

        Console.WriteLine($"{person1.Name} is {person1.Age} years old");
        Console.WriteLine($"{person2.Name} is {person2.Age} years old");
    }
}
```

Key Points:

- Each object has its own copy of the fields
- Objects are created using the `new` keyword
- Fields store the state/data of an object

Part 2: Constructors and Methods (30 minutes)

Constructors

Constructors are special methods that initialize objects when they're created.

```csharp
public class Person
{
    public string Name;
    public int Age;
    public string Email;

    // Default constructor
    public Person()
    {
        Name = "Unknown";
        Age = 0;
        Email = "";
    }

    // Parameterized constructor
    public Person(string name, int age, string email)
    {
        Name = name;
        Age = age;
        Email = email;
    }

    // Constructor overloading
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
        Email = "";
    }
}
```

using constructor overloading we can tailor the object creation depending on the situation. It's make the code very dynamic and flexible creating multiple entry-points to use it.

Methods

Methods define what objects can do (behavior).

```csharp
```

```csharp
public class Person
{
    public string Name;
    public int Age;
    public string Email;

    // Constructor
    public Person(string name, int age, string email)
    {
        Name = name;
        Age = age;
        Email = email;
    }

    // Methods
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}, Email: {Email}");
    }

    public void HaveBirthday()
    {
        Age++;
        Console.WriteLine($"Happy Birthday {Name}! You are now {Age} years old.");
    }

    public bool IsAdult()
    {
        return Age >= 18;
    }

    public string GetAgeGroup()
    {
        if (Age < 13) return "Child";
        if (Age < 20) return "Teenager";
        if (Age < 65) return "Adult";
        return "Senior";
    }
}
```

Using the Enhanced Class

```csharp
```

```
class Program
{
    static void Main()
    {
        // Using different constructors
        Person person1 = new Person("Alice", 25, "alice@email.com");
        Person person2 = new Person("Bob", 17);
        Person person3 = new Person(); // Uses default constructor

        // Using methods
        person1.DisplayInfo();
        person1.HaveBirthday();

        Console.WriteLine($"Is {person2.Name} an adult? {person2.IsAdult()}");
        Console.WriteLine($"{person2.Name} is in the {person2.GetAgeGroup()} age group");
    }
}
```

Part 3: Access Modifiers (30 minutes)

Access modifiers control who can access your class members.

The Four Access Modifiers:

1. **public**: Accessible from anywhere
2. **private**: Only accessible within the same class
3. **protected**: Accessible within the class and its derived classes
4. **internal**: Accessible within the same assembly

csharp

```
public class BankAccount
{
    // Public - can be accessed from anywhere
    public string AccountHolder;

    // Private - only accessible within this class
    private decimal balance;

    // Protected - accessible in this class and derived classes
    protected string accountNumber;

    // Internal - accessible within the same assembly
    internal string bankCode;

    public BankAccount(string holder, decimal initialBalance)
    {
        AccountHolder = holder;
        balance = initialBalance;
        accountNumber = GenerateAccountNumber();
        bankCode = "BANK001";
    }

    // Public method to access private field
    public decimal GetBalance()
    {
        return balance;
    }

    // Public method to modify private field
    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
            Console.WriteLine($"Deposited ${amount}. New balance: ${balance}");
        }
    }

    public bool Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= balance)
        {
            balance -= amount;
```

```
            Console.WriteLine($"Withdrew ${amount}. New balance: ${balance}");
            return true;
        }
        Console.WriteLine("Insufficient funds or invalid amount");
        return false;
    }

    // Private method - only used internally
    private string GenerateAccountNumber()
    {
        return $"ACC{DateTime.Now.Ticks}";
    }
}
```

Why Use Access Modifiers?

csharp

```
class Program
{
    static void Main()
    {
        BankAccount account = new BankAccount("John Doe", 1000);

        // This works - public field
        Console.WriteLine($"Account holder: {account.AccountHolder}");

        // This works - public method
        Console.WriteLine($"Balance: ${account.GetBalance()}");

        // This would cause an error - private field
        // Console.WriteLine(account.balance); // Error!

        // This works - public method to modify private data
        account.Deposit(500);
        account.Withdraw(200);
    }
}
```

Part 4: Hands-on Lab - Build a Simple Class (30 minutes)

Exercise: Create a Car Class

Create a `Car` class with the following requirements:

1. **Fields**: brand, model, year, mileage (private), isRunning (private)
2. **Constructor**: Takes brand, model, and year
3. **Methods**:
   - `StartEngine()` - sets isRunning to true
   - `StopEngine()` - sets isRunning to false
   - `Drive(int miles)` - adds to mileage if engine is running
   - `GetMileage()` - returns current mileage
   - `GetCarInfo()` - returns formatted string with car details

Solution:

```csharp
```

```
public class Car
{
    // Public fields
    public string Brand;
    public string Model;
    public int Year;

    // Private fields
    private int mileage;
    private bool isRunning;

    // Constructor
    public Car(string brand, string model, int year)
    {
        Brand = brand;
        Model = model;
        Year = year;
        mileage = 0;
        isRunning = false;
    }

    // Public methods
    public void StartEngine()
    {
        isRunning = true;
        Console.WriteLine($"The {Brand} {Model} engine is now running.");
    }

    public void StopEngine()
    {
        isRunning = false;
        Console.WriteLine($"The {Brand} {Model} engine is now stopped.");
    }

    public void Drive(int miles)
    {
        if (isRunning && miles > 0)
        {
            mileage += miles;
            Console.WriteLine($"Drove {miles} miles. Total mileage: {mileage}");
        }
        else if (!isRunning)
        {
```

```
            Console.WriteLine("Cannot drive. Engine is not running!");
        }
        else
        {
            Console.WriteLine("Invalid miles value.");
        }
    }

    public int GetMileage()
    {
        return mileage;
    }

    public string GetCarInfo()
    {
        return $"{Year} {Brand} {Model} - Mileage: {mileage} miles, Engine: {(isRunning ? "Running" : "Stopped")}";
    }
}

// Test the Car class
class Program
{
    static void Main()
    {
        Car myCar = new Car("Toyota", "Camry", 2020);

        Console.WriteLine(myCar.GetCarInfo());

        myCar.StartEngine();
        myCar.Drive(50);
        myCar.Drive(25);

        Console.WriteLine(myCar.GetCarInfo());

        myCar.StopEngine();
        myCar.Drive(10); // Should show error message

        Console.WriteLine($"Final mileage: {myCar.GetMileage()} miles");
    }
}
```

Day 1 Summary

What You've Learned:

1. **Classes and Objects**: Blueprint vs instance concept
2. **Fields**: Store object data/state
3. **Constructors**: Initialize objects when created
4. **Methods**: Define object behavior
5. **Access Modifiers**: Control visibility and access

Key Principles:

- **Encapsulation**: Keep data private and provide public methods to access it
- **Initialization**: Always initialize your objects properly
- **Separation of Concerns**: Methods should have clear, single responsibilities

Tomorrow's Preview:

Day 2 will cover **Properties** - a more sophisticated way to handle data access, and **Static Members** - shared across all instances.

Practice Exercises (Optional)

1. Create a `Student` class with name, ID, and grades list
2. Create a `Rectangle` class with width, height, and methods to calculate area/perimeter
3. Create a `Temperature` class that can convert between Celsius and Fahrenheit

Remember: The key to mastering OOP is practice. Try to think of real-world entities and model them as classes!