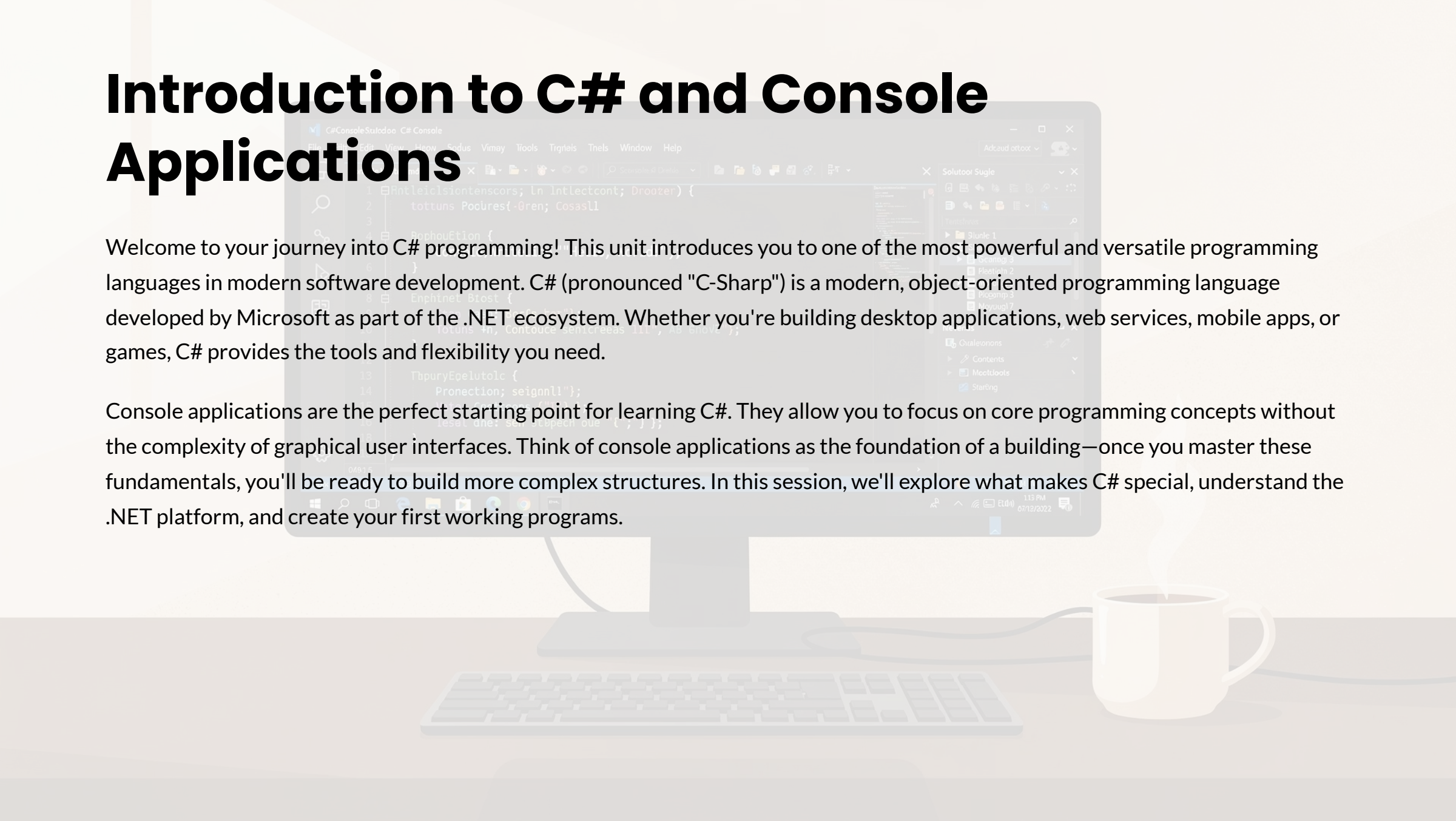
Introduction to C# and Console Applications



Welcome to your journey into C# programming! This unit introduces you to one of the most powerful and versatile programming languages in modern software development. C# (pronounced "C-Sharp") is a modern, object-oriented programming language developed by Microsoft as part of the .NET ecosystem. Whether you're building desktop applications, web services, mobile apps, or games, C# provides the tools and flexibility you need.

Console applications are the perfect starting point for learning C#. They allow you to focus on core programming concepts without the complexity of graphical user interfaces. Think of console applications as the foundation of a building—once you master these fundamentals, you'll be ready to build more complex structures. In this session, we'll explore what makes C# special, understand the .NET platform, and create your first working programs.

Understanding C# and the .NET Ecosystem

What is C#?

A modern, type-safe, object-oriented language designed for building robust applications on the .NET platform

The .NET Platform

A comprehensive development framework providing libraries, runtime, and tools for building various applications

Why Console Apps?

Perfect for learning fundamentals, automation scripts, utilities, and backend processing tasks

C# was introduced by Microsoft in 2000 as part of the .NET initiative, designed by Anders Hejlsberg. It combines the power of C++ with the simplicity of Visual Basic, making it an ideal language for both beginners and experienced developers. The language has evolved significantly over the years, with each version adding powerful features while maintaining backward compatibility.

The .NET ecosystem consists of several components working together. The Common Language Runtime (CLR) manages program execution, handling memory management, security, and exception handling. The Base Class Library (BCL) provides thousands of pre-built classes for common tasks like file operations, networking, and data manipulation. When you write C# code, it's compiled into Intermediate Language (IL), which the CLR then converts to machine code at runtime—this process is called Just-In-Time (JIT) compilation.

Console applications are command-line programs that interact with users through text input and output. While they might seem simple compared to modern graphical applications, they're incredibly powerful for automation, batch processing, system utilities, and learning programming fundamentals. Many professional developers use console applications daily for tasks like data processing, file manipulation, and system administration.

Structure of a C# Console Program

Every C# program follows a specific structure that might seem formal at first, but each part serves an important purpose. Let's break down the anatomy of a C# console application step by step, understanding why each component exists and how they work together.

At the top of most C# files, you'll find `using` directives. Think of these as telling C# which toolboxes you want to access. Just like a carpenter doesn't carry every tool to every job site, your program doesn't load every .NET library—you specify which ones you need. For example, `using System;` gives you access to fundamental classes like Console, String, and DateTime. The System namespace is like the main toolbox containing the most commonly used tools.

Next comes the namespace declaration. Namespaces organize your code into logical groups, preventing naming conflicts. Imagine you and your classmate both create a class called "Student"—namespaces ensure these don't clash. It's like having folders on your computer: you can have a file named "report.doc" in multiple folders without confusion. A typical namespace might look like `namespace MyFirstApp`, grouping all related classes together.

Key Components

- Using directives - Import necessary namespaces
- Namespace - Organize and group related code
- Class - Container for methods and data
- Main method - Entry point of execution
- Statements - Instructions for the computer

Execution Flow

When you run a C# console application, the CLR searches for the Main() method and begins execution there. Everything inside Main() executes sequentially from top to bottom unless you use control structures like loops or conditionals.

The class is a blueprint for creating objects and contains methods and properties. Every console application must have at least one class. Inside your class, the `Main()` method is special—it's the entry point where your program begins execution. When you click "Run" in Visual Studio, the operating system looks for Main() and starts executing code from there. The signature `static void Main(string[] args)` tells us several things: it's static (can be called without creating an object), returns void (doesn't return a value), and accepts string arguments (command-line parameters).

Your First C# Program: Hello World

```
using System;

namespace MyFirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Display output to console
            Console.WriteLine("Hello, World!");
            Console.WriteLine("Welcome to C# Programming!");

            // Keep console window open
            Console.ReadKey();
        }
    }
}
```

Let's analyze this classic "Hello World" program line by line. The first line, `using System;`, imports the System namespace, which contains the Console class we'll use for output. Without this, we'd have to write `System.Console.WriteLine()` every time—the using directive saves us that repetition.

The `namespace MyFirstApp` declares our own namespace to organize this code. Inside, we define a class called Program (you can name it anything, but Program is conventional for console apps). The class contains our Main method where execution begins.

01

Program Starts

CLR finds and invokes the Main() method

02

Execute Statements

WriteLine() outputs text to the console window

03

Wait for Input

ReadKey() pauses until user presses a key

04

Program Ends

Control returns to operating system

`Console.WriteLine()` outputs text followed by a newline character, moving the cursor to the next line. If you want to print without moving to a new line, use `Console.Write()` instead. The text inside quotation marks is called a string literal—the exact text that appears on screen.

The `Console.ReadKey()` at the end is important. Without it, the program would execute and immediately close the console window, and you wouldn't see the output. This method waits for the user to press any key before the program terminates, giving you time to read what's displayed.

Working with Console Output

The Console class provides several methods for displaying information to users. Understanding these methods and their differences is crucial for creating effective console applications. Let's explore the primary output methods and formatting techniques.

`Console.WriteLine()` is the most commonly used method. It displays text and automatically moves to the next line. Think of it like pressing Enter after typing—the cursor moves down. In contrast, `Console.Write()` displays text but keeps the cursor on the same line, useful for creating prompts or building output piece by piece.

String Formatting Techniques

C# offers multiple ways to format output, from simple concatenation to modern string interpolation

```
// Concatenation
Console.WriteLine("Hello " + name);

// Composite formatting
Console.WriteLine("Name: {0}, Age: {1}",
    name, age);

// String interpolation (modern)
Console.WriteLine($"Name: {name}, Age: {age}");
```

String concatenation using the + operator is straightforward but can become messy with multiple variables. Composite formatting uses placeholders like {0} and {1}, where numbers correspond to the order of arguments. String interpolation, introduced in C# 6, uses the $ symbol before the string and allows you to embed expressions directly inside curly braces—it's more readable and less error-prone.

You can also control formatting for numbers and dates. For example, `Console.WriteLine($"{price:C}")` formats a number as currency, and `Console.WriteLine($"{date:yyyy-MM-dd}")` formats dates. The Console class also provides `ForegroundColor` and `BackgroundColor` properties to change text colors, making your output more visually distinctive.

Processing Console Input

```
using System;

namespace InputExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get user's name
            Console.Write("Enter your name: ");
            string name = Console.ReadLine();

            // Get user's age
            Console.Write("Enter your age: ");
            int age = Convert.ToInt32(Console.ReadLine());

            // Display personalized message
            Console.WriteLine($"\nHello {name}!");
            Console.WriteLine($"You are {age} years old.");

            Console.ReadKey();
        }
    }
}
```

Reading user input is fundamental to interactive console applications. The `Console.ReadLine()` method reads everything the user types until they press Enter, returning it as a string. This is like asking someone a question and waiting for their complete answer before continuing.

However, there's an important gotcha: `ReadLine()` always returns a string, even if the user types numbers. If you need numeric input, you must convert the string to the appropriate type. The `Convert` class provides methods like `ToInt32()`, `ToDouble()`, and `ToBoolean()` for this purpose.

ReadLine() Method

Reads a complete line of text input from the user, returning it as a string. Waits for Enter key press.

Type Conversion

Use Convert class or Parse methods to transform string input into numbers, booleans, or other types.

Input Validation

Always validate user input to prevent crashes from invalid data. Use TryParse for safer conversion.

For more robust input handling, use `TryParse()` methods instead of Convert. For example: `int.TryParse(Console.ReadLine(), out int age)` attempts the conversion and returns true if successful, false otherwise. This prevents your program from crashing when users enter invalid data like "twenty" instead of "20". The `out` keyword allows the method to return multiple values—the boolean success indicator and the converted value.

Building a Simple Calculator

Let's apply everything we've learned to create a practical console calculator. This project demonstrates input/output, type conversion, and basic arithmetic operations—all fundamental programming skills.

```
using System;

namespace SimpleCalculator
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("=== Simple Calculator ===\n");

            // Get first number
            Console.Write("Enter first number: ");
            double num1 = Convert.ToDouble(Console.ReadLine());

            // Get operator
            Console.Write("Enter operator (+, -, *, /): ");
            char operation = Convert.ToChar(Console.ReadLine());

            // Get second number
            Console.Write("Enter second number: ");
            double num2 = Convert.ToDouble(Console.ReadLine());

            // Perform calculation
            double result = 0;

            switch(operation)
            {
                case '+':
                    result = num1 + num2;
                    break;
                case '-':
                    result = num1 - num2;
                    break;
                case '*':
                    result = num1 * num2;
                    break;
                case '/':
                    if(num2 != 0)
                        result = num1 / num2;
                    else
                        Console.WriteLine("Error: Division by zero!");
                    break;
                default:
                    Console.WriteLine("Invalid operator!");
                    break;
            }

            // Display result
            Console.WriteLine($"\n{num1} {operation} {num2} = {result}");

            Console.ReadKey();
        }
    }
}
```

This calculator demonstrates several important concepts. We use `double` for numbers to handle decimal values. The switch statement efficiently handles different operations based on the operator character. Notice the division-by-zero check—this is a critical validation that prevents runtime errors.

Input Collection

Gather two numbers and an operator from user

Operation Selection

Use switch statement to choose calculation

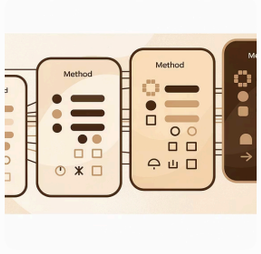
Result Display

Show formatted calculation result to user

Object-Oriented Programming Fundamentals

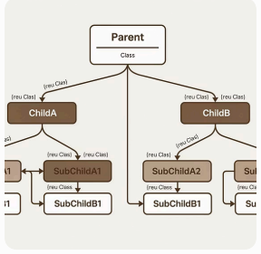
Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects rather than functions and logic. Think of it like building with LEGO blocks—each block (object) has specific characteristics (properties) and things it can do (methods). Objects are instances of classes, and classes are blueprints that define what those objects look like and how they behave.

Why use OOP? Imagine you're building a student management system. Without OOP, you'd have separate variables for each student's name, age, grade, etc., quickly becoming chaotic with dozens of students. With OOP, you create a Student class once, then create individual student objects as needed. This approach makes code more organized, reusable, and easier to maintain.



Encapsulation

Bundling data and methods together while hiding internal details, protecting data from direct access

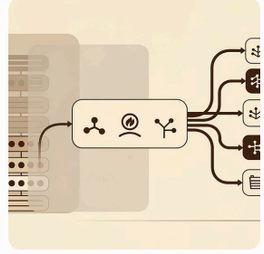


Inheritance

Creating new classes based on existing ones, inheriting properties and behaviors to promote code reuse

Polymorphism

Ability for objects to take multiple forms, allowing same operation to behave differently on different classes



Abstraction

Hiding complex implementation details and showing only essential features, simplifying interaction with objects

The four pillars of OOP—Encapsulation, Inheritance, Polymorphism, and Abstraction—work together to create robust, maintainable applications. Encapsulation protects your data by controlling access through methods. Inheritance allows you to build on existing code without rewriting it. Polymorphism lets you write flexible code that works with different object types. Abstraction hides complexity, letting you focus on what an object does rather than how it does it.

Constructors and Destructors

Constructors and destructors manage the lifecycle of objects—constructors initialize objects when they're created, while destructors clean up when objects are destroyed. Understanding these special methods is crucial for proper resource management in your applications.

A constructor is a special method that runs automatically when you create an object. It has the same name as the class and no return type. Think of it as the setup crew before a concert—it prepares everything needed for the object to function properly. Constructors initialize fields, allocate resources, and ensure the object starts in a valid state.

Default Constructor

```
public class Student
{
    public string Name;
    public int Age;

    // Default constructor
    public Student()
    {
        Name = "Unknown";
        Age = 0;
    }
}
```

Parameterized Constructor

```
public class Student
{
    public string Name;
    public int Age;

    // Parameterized constructor
    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

C# supports constructor overloading—you can have multiple constructors with different parameters. A default constructor takes no parameters and sets default values. A parameterized constructor accepts values to customize object creation. You might have both: a default constructor for when you don't have initial values, and a parameterized one for when you do.

Destructors, marked with a tilde (~) before the class name, clean up resources when an object is destroyed. However, in C# you rarely write destructors because the garbage collector automatically handles memory management. Destructors are mainly needed when working with unmanaged resources like file handles or database connections. The syntax looks like `~ClassName() { }`, and you cannot call destructors explicitly—the garbage collector does it for you.

```
public class FileHandler
{
 private FileStream file;

 // Constructor - opens file
 public FileHandler(string path)
 {
 file = new FileStream(path, FileMode.Open);
 }

 // Destructor - closes file
 ~FileHandler()
 {
 if(file != null)
 {
 file.Close();
 }
 }
}
```

Function and Operator Overloading

Overloading is a form of compile-time polymorphism that allows you to have multiple methods with the same name but different parameters. It's like having multiple tools with the same name but designed for different tasks—a "cutter" might be scissors for paper, a knife for food, or a saw for wood.

Function overloading, also called method overloading, lets you create several methods with identical names as long as they have different parameter lists. The compiler determines which version to call based on the arguments you provide. This makes your code more intuitive—instead of having AddIntegers(), AddDoubles(), and AddStrings(), you can simply have Add() for all types.

Different Parameter Count

```
void Print(string msg)
void Print(string msg, int
count)
```

Different Parameter Types

```
void Add(int a, int b)
void Add(double a, double b)
```

Different Parameter Order

```
void Display(int x, string y)
void Display(string y, int x)
```

Here's a practical example of method overloading with a Calculator class:

```
public class Calculator
{
    // Add two integers
    public int Add(int a, int b)
    {
        return a + b;
    }

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

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

    // Add array of integers
    public int Add(params int[] numbers)
    {
        int sum = 0;
        foreach(int num in numbers)
            sum += num;
        return sum;
    }
}
```

Operator overloading allows you to redefine how operators work with your custom classes. Want to add two Student objects to create a study group? Or subtract dates to get duration? Operator overloading makes this possible. You define operator methods using the `operator` keyword followed by the operator symbol.

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    // Overload + operator to add points
    public static Point operator +(Point p1, Point p2)
    {
        return new Point
        {
            X = p1.X + p2.X,
            Y = p1.Y + p2.Y
        };
    }

    // Overload == operator
    public static bool operator ==(Point p1, Point p2)
    {
        return p1.X == p2.X && p1.Y == p2.Y;
    }
}
```

When overloading operators, remember they must be static methods. Most operators come in pairs—if you overload ==, you should also overload !=. Operator overloading makes your classes feel like built-in types, improving code readability and naturalness.

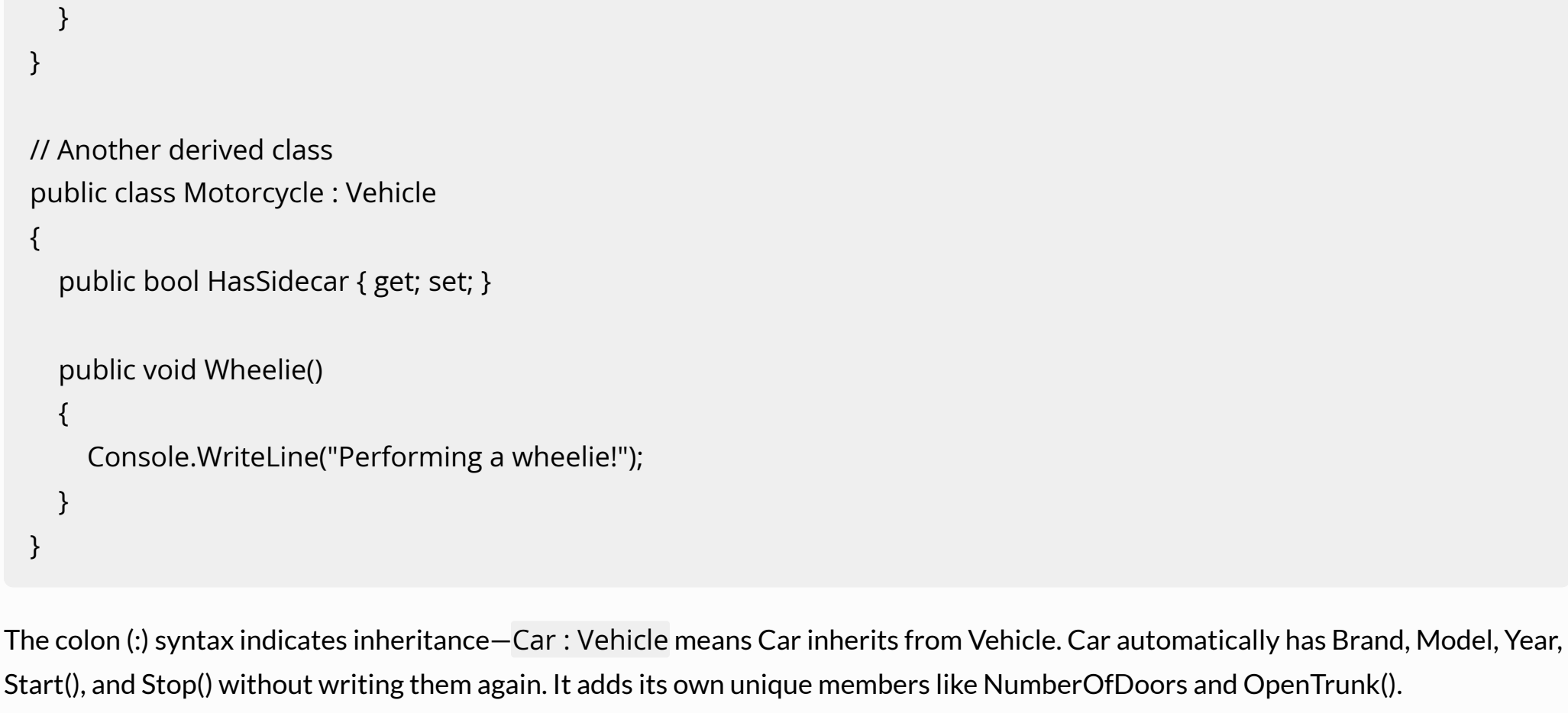
Inheritance in C#

Inheritance is a mechanism where a new class (derived class or child class) inherits properties and methods from an existing class (base class or parent class). It's like genetic inheritance—children inherit characteristics from parents but can also have their own unique traits. This promotes code reuse and establishes natural hierarchies in your programs.

Imagine you're building a vehicle management system. All vehicles share common properties like color, model, and speed. Instead of duplicating these properties in Car, Truck, and Motorcycle classes, you create a base Vehicle class and have others inherit from it. Each derived class gets the base functionality for free and adds its own specialized features.

```
// Base class
public class Vehicle
{
    public string Brand { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }

    public void Start()
    {
        Console.WriteLine($"{Brand} {Model} is starting...");
    }

    public void Stop()
    {
        Console.WriteLine($"{Brand} {Model} has stopped.");
    }
}

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

    public void OpenTrunk()
    {
        Console.WriteLine("Trunk opened.");
    }
}

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

    public void Wheelie()
    {
        Console.WriteLine("Performing a wheelie!");
    }
}
```

The colon (:) syntax indicates inheritance—`Car : Vehicle` means Car inherits from Vehicle. Car automatically has Brand, Model, Year, Start(), and Stop() without writing them again. It adds its own unique members like NumberOfDoors and OpenTrunk().



The `base` keyword lets derived classes access base class members. For example, if Car has its own constructor, you can call the base constructor: `public Car(string brand) : base(brand) { }`. You can also use `base.MethodName()` to call base class methods from derived classes.

C# supports single inheritance (a class can inherit from only one base class) but multiple interface implementation. You cannot have `class Car : Vehicle, Machine`, but you can implement multiple interfaces. The sealed keyword prevents a class from being inherited: `sealed class FinalClass { }`, useful when you want to prevent further derivation for security or design reasons.

Access Modifiers and Encapsulation

Access modifiers control the visibility and accessibility of classes, methods, and properties. They're like security clearance levels in a building—some areas are public (anyone can enter), others are private (restricted access), and some are protected (only family members allowed). Proper use of access modifiers is fundamental to encapsulation, one of OOP's core principles.

public

Accessible from anywhere in the program. Use for methods and properties that should be available to all code.

private

Accessible only within the same class. Use for internal implementation details that should be hidden.

protected

Accessible within the class and derived classes. Use for members that child classes need but others don't.

internal

Accessible within the same assembly. Use for functionality shared within your project but not exposed externally.

Encapsulation means bundling data and methods together while restricting direct access to some components. It's like a car's engine—you interact with the gas pedal and steering wheel (public interface), but the internal combustion process is hidden (private implementation). This protects data integrity and allows internal changes without breaking external code.

```
public class BankAccount
{
    // Private fields - hidden from outside
    private string accountNumber;
    private decimal balance;
    private string pin;

    // Public properties - controlled access
    public string AccountNumber
    {
        get { return accountNumber; }
    }

    public decimal Balance
    {
        get { return balance; }
        private set { balance = value; }
    }

    // Public methods - interface for interaction
    public bool Deposit(decimal amount)
    {
        if(amount > 0)
        {
            balance += amount;
            return true;
        }
        return false;
    }

    public bool Withdraw(decimal amount, string enteredPin)
    {
        if(enteredPin == pin && amount > 0 && amount <= balance)
        {
            balance -= amount;
            return true;
        }
        return false;
    }
}
```

In this example, sensitive data like balance and pin are private. You can't directly modify them from outside the class. Instead, you use public methods like Deposit() and Withdraw() which include validation logic. This prevents invalid operations like negative deposits or unauthorized withdrawals.

Best practices: Make fields private by default, expose them through properties when needed. Keep implementation details private. Use public only for methods that form your class's intended interface. Protected is useful in inheritance hierarchies where derived classes need access to base class internals. Internal is valuable in large projects where you want collaboration within your assembly but controlled exposure externally.

Properties and Indexers

Properties provide controlled access to private fields, combining the security of methods with the simplicity of fields. Instead of writing separate getter and setter methods like in Java, C# uses property syntax that looks like field access but runs method code behind the scenes. Think of properties as smart fields—they look like simple variables but can contain logic.

```
public class Student
{
    // Private backing field
    private string name;
    private int age;

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

    // Auto-implemented property
    public string Email { get; set; }

    // Read-only property (no setter)
    public int Age
    {
        get { return age; }
    }

    // Write-only property (rare, but possible)
    private string password;
    public string Password
    {
        set { password = HashPassword(value); }
    }

    // Property with private setter
    public DateTime EnrollmentDate { get; private set; }

    private string HashPassword(string pwd)
    {
        // Hashing logic here
        return pwd; // Simplified
    }
}
```

The `get` accessor returns the property value, while `set` assigns it. Inside set, the `value` keyword represents the assigned value. This lets you add validation—checking if a name is empty, ensuring age is positive, or formatting data before storage.

Auto-Implemented Properties

When you don't need validation logic, use auto-implemented properties: `public string Email { get; set; }`. The compiler automatically creates a hidden backing field.

Property Access Levels

- Read-only: Only get accessor (calculated or initialized values)
- Write-only: Only set accessor (passwords, configuration)
- Read-write: Both accessors (normal editable data)
- Mixed access: Public get, private set (externally readable, internally writable)

Indexers allow objects to be indexed like arrays. They're useful when your class represents a collection or has naturally indexed data. An indexer uses `this[index]` syntax:

```
public class StudentGroup
{
    private Student[] students = new Student[50];

    // Indexer using integer index
    public Student this[int index]
    {
        get
        {
            if(index >= 0 && index < students.Length)
                return students[index];
            throw new IndexOutOfRangeException();
        }
        set
        {
            if(index >= 0 && index < students.Length)
                students[index] = value;
        }
    }

    // Indexer using string key
    public Student this[string name]
    {
        get
        {
            foreach(var student in students)
            {
                if(student != null && student.Name == name)
                    return student;
            }
            return null;
        }
    }
}

// Usage:
StudentGroup group = new StudentGroup();
group[0] = new Student { Name = "Alice" };
Student s = group[0];  // Get by index
Student alice = group["Alice"];  // Get by name
```

Indexers can be overloaded with different parameter types, allowing flexible access patterns. They make your custom collections feel like built-in arrays or dictionaries, improving code readability and intuitiveness.

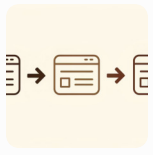
Attributes and Metadata

Attributes are special tags you attach to code elements (classes, methods, properties) to provide metadata—information about the code that doesn't affect its execution but can be read by other code or tools. Think of attributes like sticky notes on files: they don't change the content but provide additional information about it.

Attributes appear in square brackets above the element they describe. C# includes many built-in attributes, and you can create custom ones. They're used extensively in frameworks like ASP.NET for routing, Entity Framework for database mapping, and serialization libraries for controlling JSON/XML output.

Obsolete

Marks code as deprecated, warning developers not to use it and suggesting alternatives



Serializable

Indicates a class can be serialized (converted to byte stream for storage or transmission)

Conditional

Makes methods execute only when specific compilation symbols are defined

```
// Built-in attributes examples
[Obsolete("Use NewMethod instead", false)]
public void OldMethod()
{
    // This method will trigger a warning
}

[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

[Conditional("DEBUG")]
public void DebugLog(string message)
{
    Console.WriteLine($"Debug: {message}");
}
```

Creating custom attributes is straightforward—inherit from the `Attribute` class and mark your class with `[AttributeUsage]` to specify where it can be applied:

```
// Custom attribute definition
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DeveloperInfoAttribute : Attribute
{
    public string Name { get; set; }
    public string Date { get; set; }

    public DeveloperInfoAttribute(string name)
    {
        Name = name;
        Date = DateTime.Now.ToString();
    }
}

// Using custom attribute
[DeveloperInfo("John Doe")]
public class Calculator
{
    [DeveloperInfo("Jane Smith")]
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

Attributes can have constructor parameters (positional parameters) and properties (named parameters). You set properties using named syntax: `[DeveloperInfo("John", Date = "2024-01-15")]`. This flexibility makes attributes powerful for documenting code, configuring behavior, and providing metadata for frameworks and tools.

Reflection API: Inspecting Code at Runtime

Reflection is a powerful feature that allows programs to examine and manipulate their own structure at runtime. It's like giving your program a mirror to look at itself—it can discover what classes exist, what methods they have, what attributes are attached, and even create objects or invoke methods dynamically. While typically used in advanced scenarios, understanding reflection basics opens doors to powerful programming techniques.

The `System.Reflection` namespace contains classes for reflection operations. The most important is the `Type` class, which represents type information. You can get a Type object in several ways and then use it to explore the type's structure.

```
using System;
using System.Reflection;

public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Display()
    {
        Console.WriteLine($"{Name}, {Age} years old");
    }
}

class Program
{
    static void Main()
    {
        // Get Type object
        Type studentType = typeof(Student);

        // Or from an instance
        Student s = new Student();
        Type type2 = s.GetType();

        // Display type information
        Console.WriteLine($"Class Name: {studentType.Name}");
        Console.WriteLine($"Namespace: {studentType.Namespace}");
        Console.WriteLine($"Is Class: {studentType.IsClass}");

        Console.WriteLine("\nProperties:");
        PropertyInfo[] properties = studentType.GetProperties();
        foreach(PropertyInfo prop in properties)
        {
            Console.WriteLine($"  {prop.Name} ({prop.PropertyType.Name})");
        }

        Console.WriteLine("\nMethods:");
        MethodInfo[] methods = studentType.GetMethods();
        foreach(MethodInfo method in methods)
        {
            Console.WriteLine($"  {method.Name}");
        }
    }
}
```

This code retrieves and displays metadata about the Student class—its properties, methods, and other characteristics. The `GetProperties()` and `GetMethods()` methods return arrays of information objects that describe each member.

Type Discovery	Dynamic Invocation	Attribute Reading
Find what types exist in an assembly and examine their structure	Call methods and access properties without knowing them at compile time	Retrieve and process custom attributes attached to code elements

Reflection combined with attributes is extremely powerful. Here's how to read custom attributes at runtime:

```
[DeveloperInfo("Alice Johnson")]
public class Calculator
{
    public int Add(int a, int b) { return a + b; }
}

// Reading attributes with reflection
Type calcType = typeof(Calculator);
object[] attributes = calcType.GetCustomAttributes(typeof(DeveloperInfoAttribute), false);

foreach(DeveloperInfoAttribute attr in attributes)
{
    Console.WriteLine($"Developed by: {attr.Name}");
    Console.WriteLine($"Date: {attr.Date}");
}
```

Common reflection use cases include dependency injection frameworks (dynamically creating objects based on configuration), serialization libraries (reading properties to convert objects to JSON/XML), testing frameworks (discovering test methods), and plugin architectures (loading and using code from external assemblies). While powerful, reflection has performance overhead and should be used judiciously—prefer compile-time solutions when possible, reserving reflection for scenarios that genuinely require runtime flexibility.