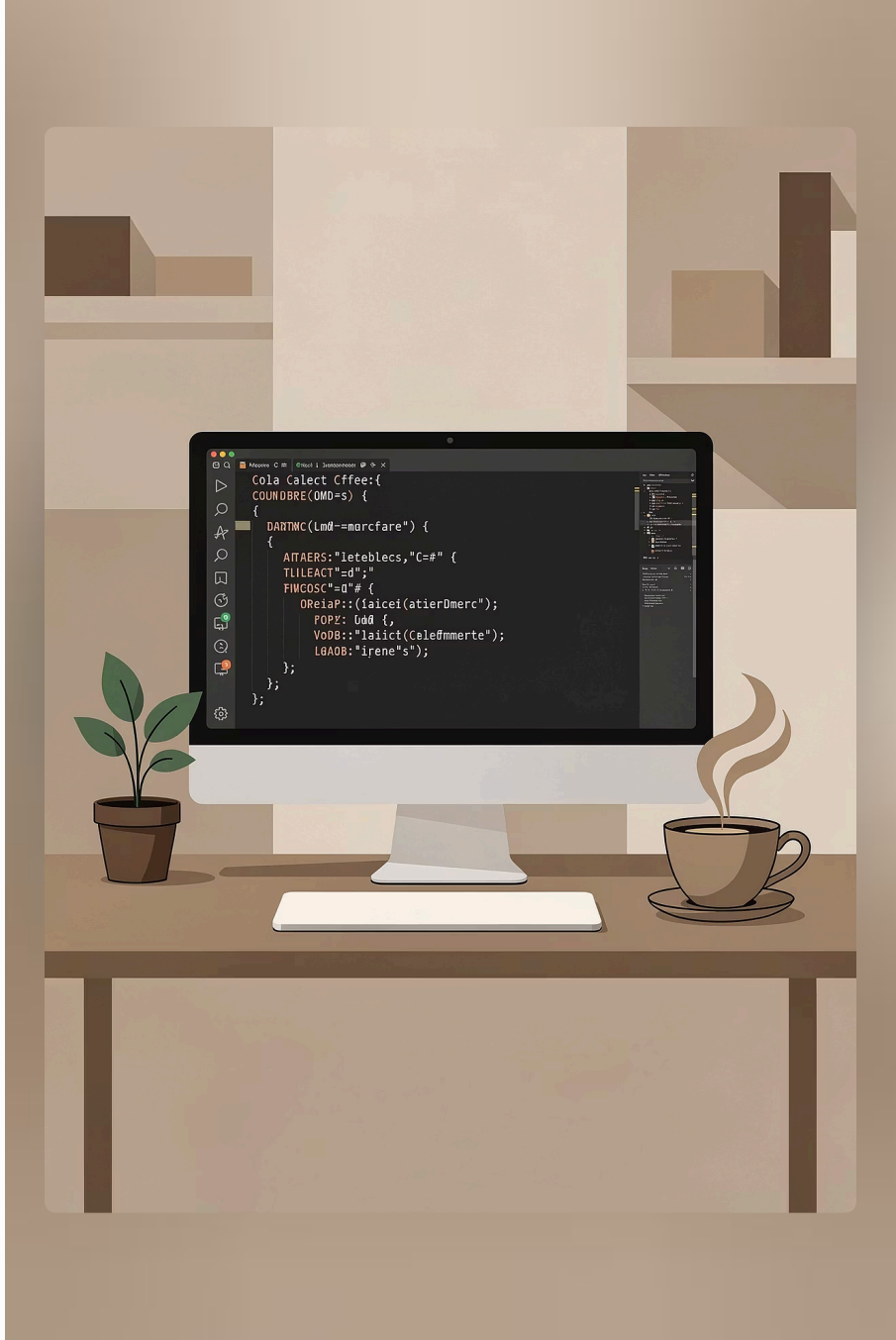
Introduction to C#

Pravin Nikam



What You'll Learn in This Unit

1

C# Fundamentals

Master the core concepts of C# programming language and the .NET ecosystem

2

Object-Oriented Programming

Apply OOP principles including constructors, inheritance, and polymorphism

3

Advanced Features

Work with properties, indexers, attributes, and reflection for robust applications

4

Console Applications

Build interactive programs with effective input and output handling

What is C#?

C# (pronounced "C-Sharp") is a modern, object-oriented programming language developed by Microsoft. It combines the power of C++ with the simplicity of Visual Basic, making it ideal for building a wide range of applications.

Key characteristics of C#:

- Type-safe and memory-managed
- Part of the .NET framework ecosystem
- Supports multiple programming paradigms
- Cross-platform compatibility
- Rich standard library and tools

Why Learn C#?

C# is widely used for developing desktop applications, web services, games, mobile apps, and enterprise software. Its versatility makes it one of the most in-demand programming languages in the industry.

The .NET Ecosystem

C# operates within the .NET framework, a comprehensive development platform that provides tools, libraries, and runtime environments for building various types of applications.

.NET Framework

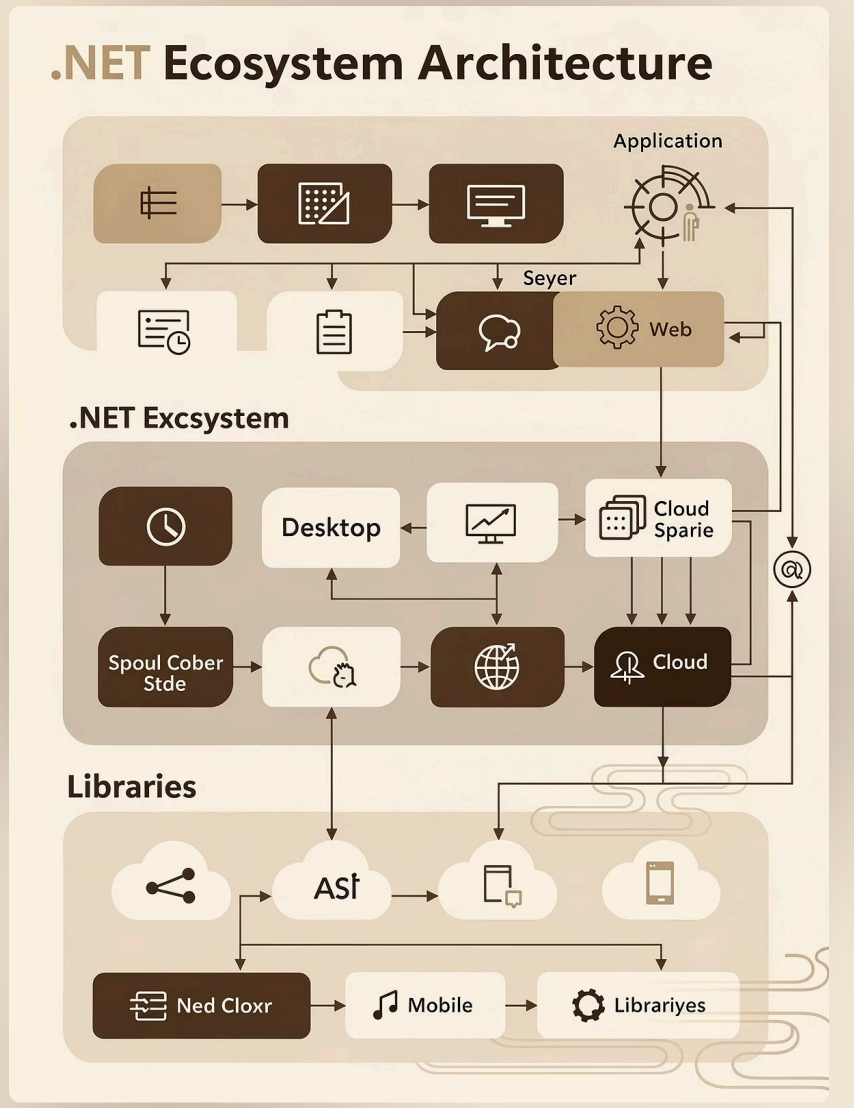
Traditional framework for Windows desktop applications and web services

.NET Core

Cross-platform, open-source framework for modern cloud-based applications

Xamarin

Platform for building native mobile applications for iOS and Android



When to Use Console Applications

Console applications are text-based programs that run in a command-line interface. They're perfect for learning programming fundamentals and building utility tools.

Ideal Use Cases

- Learning programming basics
- Automation scripts and batch processing
- System administration tools
- Quick prototyping and testing
- Command-line utilities
- Background services and daemons

Key Benefits

- Simple and straightforward development
- Fast execution and minimal overhead
- Easy to debug and test
- No complex UI considerations
- Portable across different systems

Think of console applications as the foundation: just like learning to walk before running, mastering console apps helps you understand core programming concepts before moving to GUI applications.

Structure of a C# Program

Every C# program follows a specific structure that organizes code into logical units. Understanding this structure is essential for writing clean, maintainable code.

01

Using Directives

Import namespaces to access predefined classes and methods

02

Namespace Declaration

Define a container for organizing related classes and types

03

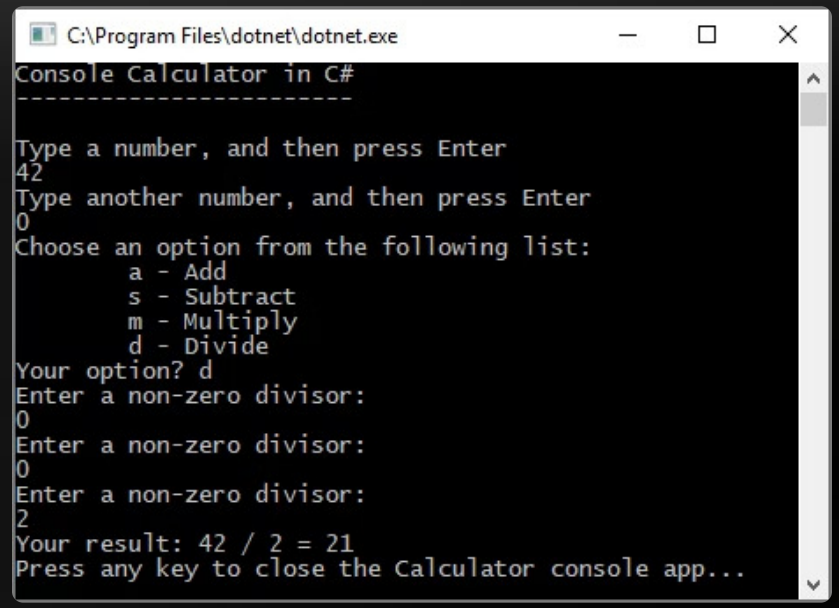
Class Definition

Create a blueprint that contains the program's methods and properties

04

Main Method

Entry point where program execution begins



Your First C# Program

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Breaking It Down

`using System;` imports the System namespace containing Console class

`namespace HelloWorld` organizes your code into a logical grouping

`class Program` defines a container for the Main method

`Main()` is where execution starts—every program needs one

`Console.WriteLine()` displays text to the console window

Understanding Namespaces

Namespaces are organizational containers that group related classes, preventing naming conflicts and making code more manageable. Think of them as folders in a file system—they help organize your code into logical categories.

System Namespace

Contains fundamental classes like Console, String, and Math

System.Collections

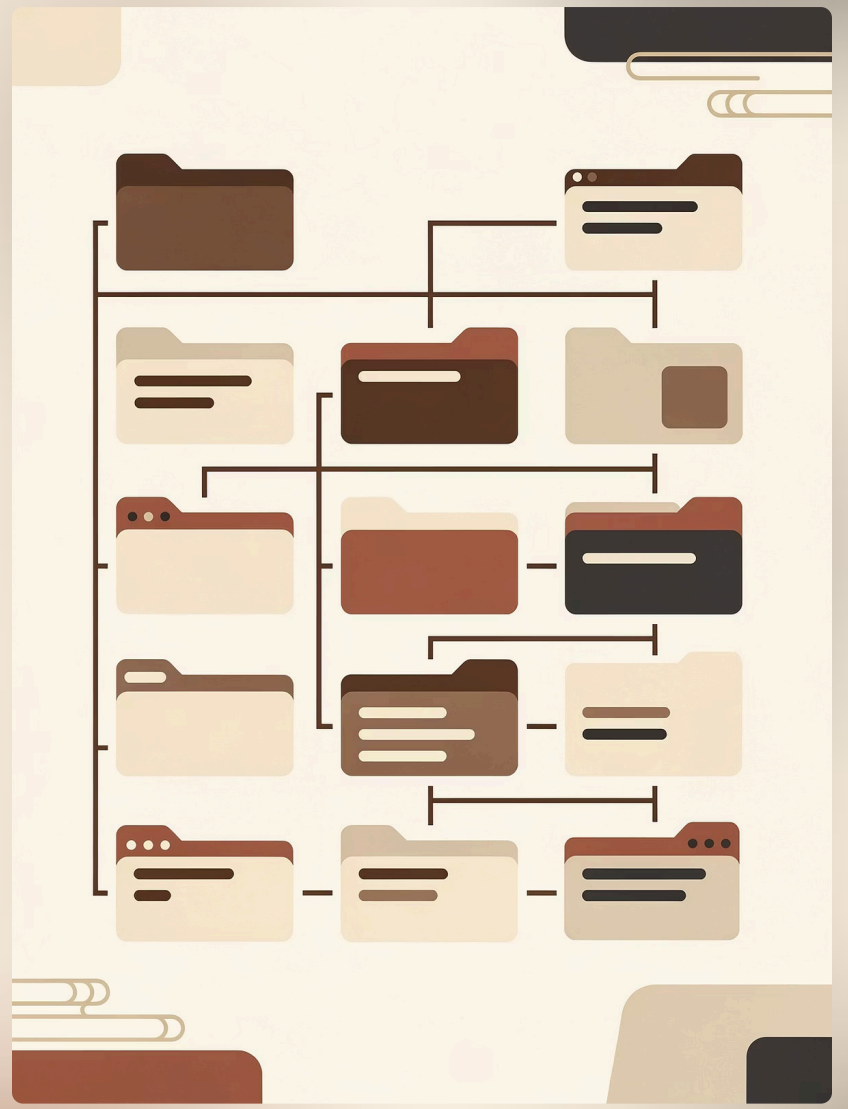
Provides data structures like ArrayList and Hashtable

System.IO

Handles file and directory operations

Custom Namespaces

Create your own to organize project-specific code



The Main() Method: Your Program's Entry Point

The Main method is where program execution begins. Every console application must have exactly one Main method that serves as the starting point for the runtime.

```
static void Main(string[] args)
{
    // Program starts here
    // args contains command-line arguments
}
```

static

Allows the method to be called without creating an instance of the class

void

Indicates the method doesn't return a value (can also return int for exit codes)

string[] args

Array of command-line arguments passed to the program

Console Output Methods

C# provides several methods for displaying information to the console. Choosing the right method depends on your formatting needs and whether you want a new line after output.

Console.WriteLine()

Outputs text and moves to a new line

```
Console.WriteLine("Hello");
Console.WriteLine("World");
// Output:
// Hello
// World
```

Console.Write()

Outputs text without a new line

```
Console.Write("Hello ");
Console.Write("World");
// Output:
// Hello World
```

Use WriteLine for separate lines of output, and Write when you want to keep output on the same line—like building a sentence piece by piece.

String Formatting in Console Output

C# offers multiple ways to format strings for display, making it easy to combine text with variables and create professional-looking output.

```
string name = "Alice";
int age = 20;

// String concatenation
Console.WriteLine("Name: " + name + ", Age: " + age);

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

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

Best Practice: Use string interpolation (`$""`) for readability and maintainability. It's the most modern and readable approach for combining strings with variables.

Processing Console Input

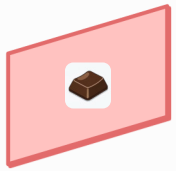
Interactive programs need to accept user input. C# provides methods to read data from the console, allowing your programs to respond to user interactions.

Console.ReadLine()

Reads an entire line of text input as a string until the user presses Enter

Console.Read()

Reads a single character and returns its integer ASCII value



Console.ReadKey()

Reads a single key press without displaying it, useful for menu navigation

Hands-On: Taking User Input

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter your name: ");
        string name = Console.ReadLine();

        Console.Write("Enter your country: ");
        string country = Console.ReadLine();

        Console.WriteLine($"\nHello {name} from {country}!");
    }
}
```

Converting Input to Numbers

ReadLine() always returns a string, so you need to convert it for numeric operations:

```
Console.Write("Enter age: ");
string input = Console.ReadLine();
int age = int.Parse(input);

// Or use TryParse for safer conversion
if (int.TryParse(input, out int age))
{
 Console.WriteLine($"Age: {age}");
}
else
{
 Console.WriteLine("Invalid number");
}
```

Building a Simple Calculator

Let's apply what we've learned by creating a basic calculator that performs arithmetic operations based on user input.

```
using System;

class Calculator
{
 static void Main()
 {
 Console.WriteLine("=== Simple Calculator ===");

 Console.Write("Enter first number: ");
 double num1 = double.Parse(Console.ReadLine());

 Console.Write("Enter operator (+, -, *, /): ");
 char op = Console.ReadLine()[0];

 Console.Write("Enter second number: ");
 double num2 = double.Parse(Console.ReadLine());

 double result = 0;

 if (op == '+') result = num1 + num2;
 else if (op == '-') result = num1 - num2;
 else if (op == '*') result = num1 * num2;
 else if (op == '/') result = num1 / num2;

 Console.WriteLine($"\nResult: {num1} {op} {num2} = {result}");
 }
}
```

Takeaways

Program Structure

Every C# program has using directives, namespace, class, and Main method

Console Output

Use WriteLine for new lines, Write for continuous output, and string interpolation for formatting

User Input

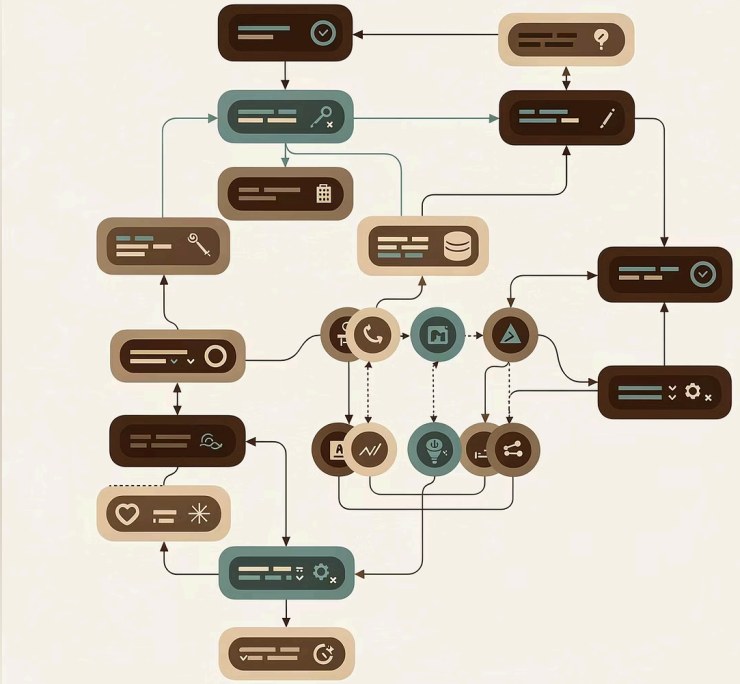
ReadLine() captures text input; convert to numbers using Parse or TryParse

Namespaces

Organize code and access functionality through System and other namespace libraries

Object-Oriented Programming in C#

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects rather than functions. In C#, OOP principles help create modular, reusable, and maintainable code.



Understanding Constructors

A constructor is a special method that initializes objects when they're created. It has the same name as the class and runs automatically when you instantiate a new object. Think of it as setting up a new house—before you move in, you need to set up utilities and furniture.

Default Constructor

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

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

Parameterized Constructor

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

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

You can have multiple constructors in a class—this is called constructor overloading.

```
class Program
{
    static void Main(string[] args)
    {
        var p = new Person("John", "Smith", 30);
    }
}

class Person
{
    public Person(string name, string surname)
    {
        Name=name;
        Surname=surname;
    }

    public string Name { get; set; }
    public string Surname { get; set; }
}
```

Working with Destructors

A destructor is a special method that performs cleanup operations before an object is destroyed. It has the same name as the class but prefixed with a tilde (~). Destructors are called automatically by the garbage collector.

```
class FileHandler
{
    private string fileName;

    public FileHandler(string name)
    {
        fileName = name;
        Console.WriteLine($"File {fileName} opened");
    }

    // Destructor
    ~FileHandler()
    {
        Console.WriteLine($"File {fileName} closed and resources released");
    }
}
```

In modern C#, you rarely need to write destructors manually. The garbage collector handles memory management automatically. Use destructors only when working with unmanaged resources like file handles or database connections.

Constructor vs Destructor

Constructor

- Called when object is created
- Initializes object state
- Can have parameters
- Can be overloaded
- Same name as class

Destructor

- Called when object is destroyed
- Performs cleanup operations
- Cannot have parameters
- Cannot be overloaded
- Same name with ~ prefix

Function Overloading (Compile-Time Polymorphism)

Function overloading allows you to define multiple methods with the same name but different parameters. The compiler determines which method to call based on the arguments provided. This is like having different-sized screwdrivers in a toolkit—same purpose, different sizes for different needs.

```
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;
 }
}

// Usage
Calculator calc = new Calculator();
Console.WriteLine(calc.Add(5, 10)); // Calls first method: 15
Console.WriteLine(calc.Add(5, 10, 15)); // Calls second method: 30
Console.WriteLine(calc.Add(5.5, 10.5)); // Calls third method: 16.0
```

```
static void Print(string text)
{
    Console.WriteLine(text);
}

static void Print(int number)
{
    Console.WriteLine(number);
}

static void Print(string text, int number)
{
    Console.WriteLine(text + " " +number);
}
```

Rules for Function Overloading

For methods to be considered overloaded, they must differ in their parameter list. Simply changing the return type is not enough.

Number of Parameters

Methods can have different numbers of parameters

`Add(int a, int b)` vs `Add(int a, int b, int c)`

Type of Parameters

Methods can have different parameter types

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

Order of Parameters

Methods can have different parameter orders

`Display(int a, string b)` vs `Display(string b, int a)`

Common mistake: You cannot overload methods by only changing the return type. The compiler uses the parameter list, not the return type, to distinguish between overloaded methods.

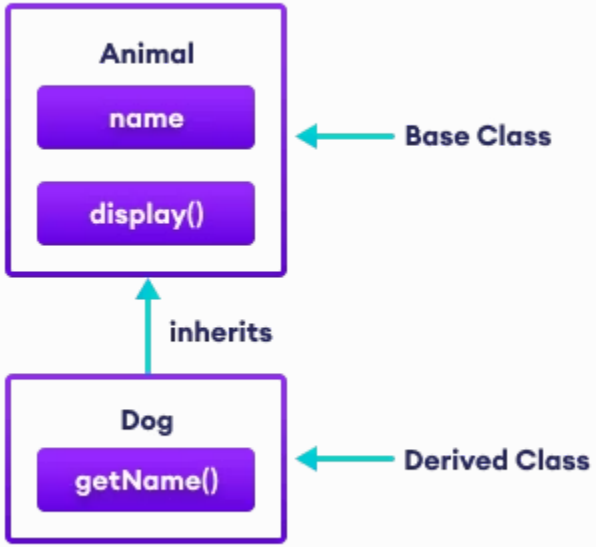
Operator Overloading

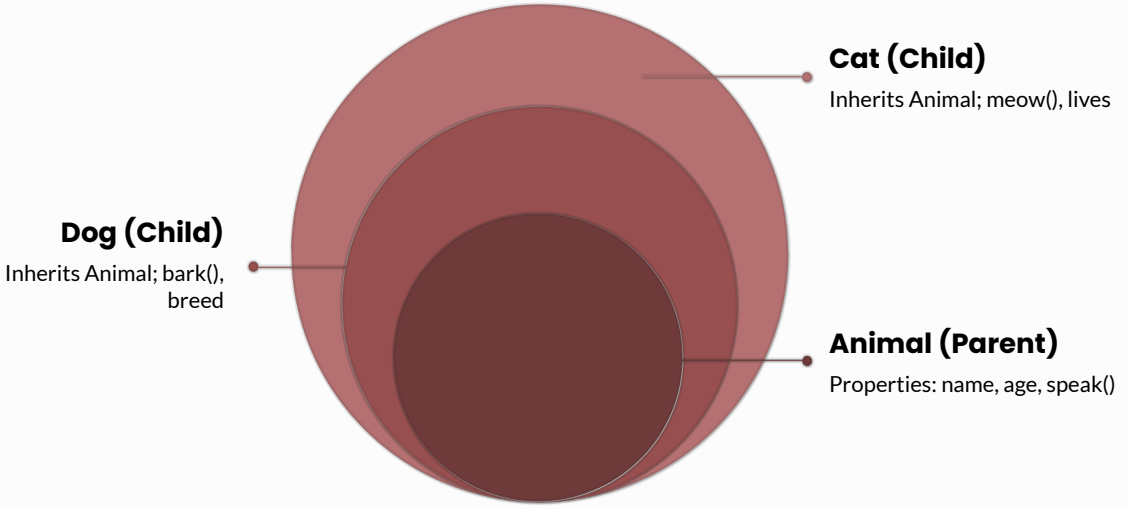
Operator overloading allows you to define how operators like +, -, *, / work with your custom classes. This makes your classes behave more naturally and intuitively.

```
class ComplexNumber
{
 public double Real { get; set; }
 public double Imaginary { get; set; }

 public ComplexNumber(double real, double imaginary)
 {
 Real = real;
 Imaginary = imaginary;
 }

 // Overload + operator
 public static ComplexNumber operator +(ComplexNumber c1, ComplexNumber c2)
 {
 return new ComplexNumber(
 c1.Real + c2.Real,
 c1.Imaginary + c2.Imaginary
 );
 }
}

// Usage
ComplexNumber c1 = new ComplexNumber(3, 4);
ComplexNumber c2 = new ComplexNumber(1, 2);
ComplexNumber result = c1 + c2; // Uses overloaded + operator
Console.WriteLine($"{result.Real} + {result.Imaginary}i"); // Output: 4 + 6i
```

Understanding Inheritance

Inheritance is a fundamental OOP concept where a new class (derived/child class) inherits properties and methods from an existing class (base/parent class). This promotes code reuse and establishes relationships between classes.





Implementing Inheritance in C#

Parent Class (Base Class)

```
class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }

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

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

Child Class (Derived Class)

```
class Dog : Animal
{
    public string Breed { get; set; }

    public void Bark()
    {
        Console.WriteLine($"{Name} says: Woof!");
    }
}

// Usage
Dog myDog = new Dog();
myDog.Name = "Buddy";
myDog.Age = 3;
myDog.Breed = "Labrador";
myDog.Eat();    // Inherited method
myDog.Bark();   // Own method
```

The colon (:) symbol indicates inheritance. Dog inherits all public and protected members from Animal.

Types of Inheritance in C#

Single Inheritance

One class inherits from one parent class

`class Dog : Animal`

Multilevel Inheritance

A class inherits from a derived class

`Animal → Dog → Puppy`

Hierarchical Inheritance

Multiple classes inherit from one parent

`Animal → Dog, Cat, Bird`

C# does NOT support multiple inheritance (one class inheriting from multiple classes) to avoid complexity and ambiguity. However, you can implement multiple interfaces, which we'll cover in later sessions.

OOP Concepts Recap

1

Constructors

Initialize objects when created; can be default or parameterized

2

Destructors

Clean up resources when objects are destroyed; called automatically

3

Function Overloading

Multiple methods with same name but different parameters; compile-time polymorphism

4

Operator Overloading

Define custom behavior for operators with your classes

5

Inheritance

Derive new classes from existing ones to promote code reuse

Access Modifiers and Encapsulation

Access modifiers control the visibility and accessibility of classes, methods, and properties. They're like security levels in a building—some areas are public, others are restricted to certain people.

Types of Access Modifiers

public

Accessible from anywhere—no restrictions. Use for APIs and interfaces that need to be widely available.

private

Accessible only within the same class. Use to hide implementation details and protect data.

protected

Accessible within the same class and derived classes. Use for inheritance scenarios.

internal

Accessible within the same assembly (project). Use for internal implementation that shouldn't be exposed externally.

Encapsulation: Protecting Your Data

Encapsulation is the practice of hiding internal data and providing controlled access through public methods or properties. It's like a bank vault—you can't directly access the money, but you can deposit or withdraw through controlled procedures.

❌ Without Encapsulation

```
class BankAccount
{
    public double balance;
}

// Problem: Direct access
BankAccount acc = new BankAccount();
acc.balance = -1000;  // Dangerous!
```

✅ With Encapsulation

```
class BankAccount
{
 private double balance;

 public void Deposit(double amount)
 {
 if (amount > 0)
 balance += amount;
 }

 public double GetBalance()
 {
 return balance;
 }
}
```

Properties in C#

Properties provide a flexible mechanism to read, write, or compute the values of private fields. They look like fields from the outside but have built-in logic for validation and control.

```
class Employee
{
 private string name;
 private int age;

 // Property with full control
 public string Name
 {
 get { return name; }
 set
 {
 if (!string.IsNullOrEmpty(value))
 name = value;
 }
 }

 // Property with validation
 public int Age
 {
 get { return age; }
 set
 {
 if (value >= 18 && value <= 65)
 age = value;
 else
 throw new ArgumentException("Age must be between 18 and 65");
 }
 }

 // Auto-implemented property (shorthand)
 public string Department { get; set; }
}
```

Read-Only and Write-Only Properties

Sometimes you want to restrict access to only reading or only writing a property. C# allows you to create properties with only get or only set accessors.

Read-Only Property

```
class Circle
{
    private double radius;

    public Circle(double r)
    {
        radius = r;
    }

    // Read-only property
    public double Area
    {
        get
        {
            return Math.PI * radius * radius;
        }
    }
}

// Usage
Circle c = new Circle(5);
Console.WriteLine(c.Area);  // OK
// c.Area = 100;  // Error: no set accessor
```

Write-Only Property

```
class User
{
 private string password;

 // Write-only property
 public string Password
 {
 set
 {
 // Hash the password before storing
 password = HashPassword(value);
 }
 }

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

// Usage
User u = new User();
u.Password = "secret123"; // OK
// string p = u.Password; // Error: no get
```

Understanding Indexers

Indexers allow objects to be indexed like arrays using the bracket notation. They're useful when your class represents a collection or container of items. Think of it as creating a custom array-like behavior for your class.

```
class StudentCollection
{
 private string[] students = new string[5];

 // Indexer
 public string 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;
 else
 throw new IndexOutOfRangeException();
 }
 }
}

// Usage
StudentCollection sc = new StudentCollection();
sc[0] = "Alice";
sc[1] = "Bob";
Console.WriteLine(sc[0]); // Output: Alice
```

Practical Example: Employee Management System

Let's combine properties, encapsulation, and validation in a real-world scenario.

```
class Employee
{
 private string name;
 private double salary;
 private string email;

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

 // Property with range validation
 public double Salary
 {
 get { return salary; }
 set
 {
 if (value >= 0)
 salary = value;
 else
 throw new ArgumentException("Salary cannot be negative");
 }
 }

 // Property with format validation
 public string Email
 {
 get { return email; }
 set
 {
 if (value.Contains("@"))
 email = value;
 else
 throw new ArgumentException("Invalid email format");
 }
 }
}
```

☆ BEST PRACTICES

Property and Encapsulation Best Practices

Always Use Private Fields

Keep fields private and expose them through properties for better control and validation

Validate in Property Setters

Add validation logic in set accessors to ensure data integrity and prevent invalid states

Use Auto-Properties When Simple

For properties without validation, use auto-implemented properties for cleaner code

Provide Clear Error Messages

When throwing exceptions, include descriptive messages explaining what went wrong

Attributes and Reflection API

Attributes provide a way to add metadata to your code elements. Reflection allows you to inspect this metadata at runtime. Together, they enable powerful features like serialization, validation frameworks, and dependency injection.

What Are Attributes?

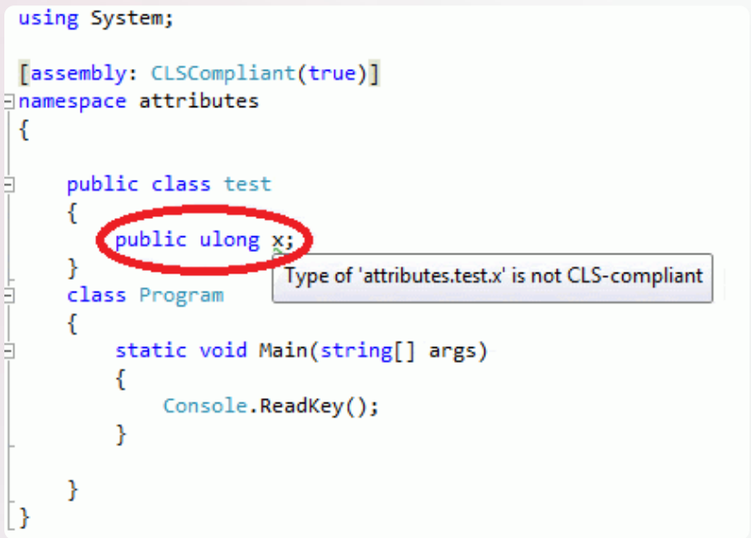
Attributes are declarative tags that add metadata to code elements like classes, methods, and properties. They don't change how code executes, but they provide information that can be used by the compiler, runtime, or other tools.

Built-in Attributes

```
// Mark method as obsolete
[Obsolete("Use NewMethod instead")]
public void OldMethod()
{
    // Implementation
}

// Serialize class to JSON/XML
[Serializable]
public class Student
{
    public string Name { get; set; }
}

// Conditional compilation
[Conditional("DEBUG")]
public void DebugMethod()
{
    Console.WriteLine("Debug mode");
}
```

Common Built-in Attributes:

- `[Obsolete]` - Marks code as deprecated
- `[Serializable]` - Enables serialization
- `[DllImport]` - Calls unmanaged code
- `[Conditional]` - Conditional compilation
- `[AttributeUsage]` - Defines attribute usage rules



Creating Custom Attributes

You can define your own attributes by creating a class that inherits from `System.Attribute`. Custom attributes are useful for framework development, validation, and adding domain-specific metadata.

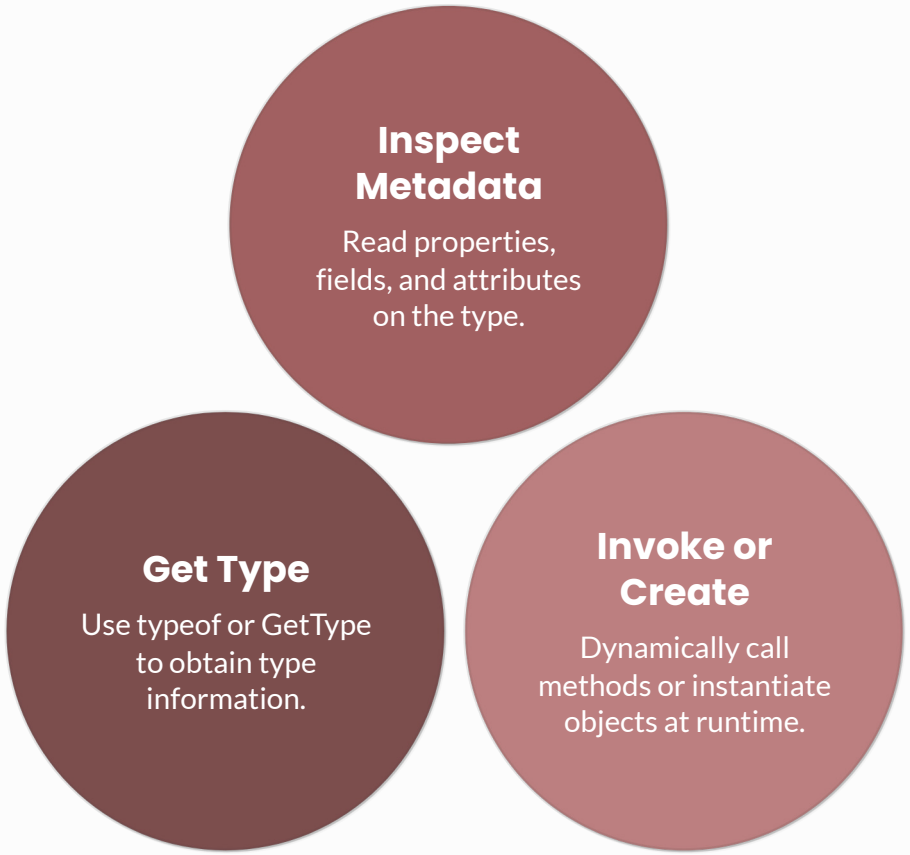
```
// Define custom attribute
[AttributeUsage(AttributeTargets.Property)]
public class RequiredAttribute : Attribute
{
    public string ErrorMessage { get; set; }

    public RequiredAttribute(string errorMessage)
    {
        ErrorMessage = errorMessage;
    }
}

// Use custom attribute
public class Student
{
    [Required("Name is required")]
    public string Name { get; set; }

    [Required("Email is required")]
    public string Email { get; set; }

    public int Age { get; set; }
}
```

The `[AttributeUsage]` attribute specifies where your custom attribute can be applied—classes, methods, properties, etc.

Introduction to Reflection API

Reflection is a powerful feature that allows you to inspect and manipulate code at runtime. You can examine types, read attributes, invoke methods, and even create objects dynamically. It's like having X-ray vision into your code.

```csharp
using System;
using System.Reflection;

namespace Reflection_CSharp
{
    3 references
    public class Student
    {
        2 references
        public int StudentID { get; set; }
        2 references
        public string Name { get; set; }
        2 references
        public int Age { get; set; }
        0 references
        public Student()
        {
            this.StudentID = 0;
            this.Name = String.Empty;
            this.Age = 0;
        }
        0 references
        public Student(int id, string name, int age)
        {
            this.StudentID = id;
            this.Name = name;
            this.Age = age;
        }
        0 references
        class Program
        {
            0 references
            static void Main(string[] args)
            {
                Type _type = typeof(Student);
                System.Reflection.MethodInfo[] methodInfo = _type.GetMethods();
                System.Reflection.MemberInfo[] memberInfo = _type.GetMembers();
                Console.WriteLine("============Methods============");
                foreach (MethodInfo mi in methodInfo)
                {
                    Console.WriteLine(mi.Name);
                }
                Console.WriteLine("============Members============");
                foreach (MemberInfo mi in memberInfo)
                {
                    Console.WriteLine(mi.Name);
                }
                Console.ReadKey();
            }
        }
    }
}
```



Using Reflection to Read Metadata

```
using System;
using System.Reflection;

public class Student
{
 [Required("Name is required")]
 public string Name { get; set; }

 [Required("Email is required")]
 public string Email { get; set; }

 public int Age { get; set; }
}

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

 Console.WriteLine($"Class Name: {studentType.Name}");
 Console.WriteLine("\nProperties:");

 foreach (PropertyInfo prop in studentType.GetProperties())
 {
 Console.WriteLine($"- {prop.Name} ({prop.PropertyType.Name})");

 // Check for custom attributes
 var attributes = prop.GetCustomAttributes(typeof(RequiredAttribute), false);
 if (attributes.Length > 0)
 {
 var reqAttr = (RequiredAttribute)attributes[0];
 Console.WriteLine($" Validation: {reqAttr.ErrorMessage}");
 }
 }
 }
}
```

Key Takeaways & What's Next

Console Mastery

You can now create interactive console applications with proper input/output handling

OOP Fundamentals

You understand constructors, inheritance, overloading, and polymorphism

Data Protection

You can implement encapsulation using access modifiers and properties

Advanced Features

You've learned about attributes and reflection for metadata inspection

---

What's Next?

In the next unit, you'll explore advanced C# concepts including interfaces, abstract classes, exception handling, collections, LINQ, and file operations. You'll also dive into Windows Forms for building graphical user interfaces.

Keep practicing! The best way to master C# is through hands-on coding.