C#.NET Programming Fundamentals

Pravin Nikam

What You'll Master in This Unit

Project Structure

Create and organize .NET projects using Visual Studio with proper namespaces and class organization

Language Features

Master C# data types, type conversion, assemblies, and the core language constructs that power modern applications

Base Class Library

Leverage .NET's built-in classes for strings, files, collections, and common programming tasks

Professional Skills

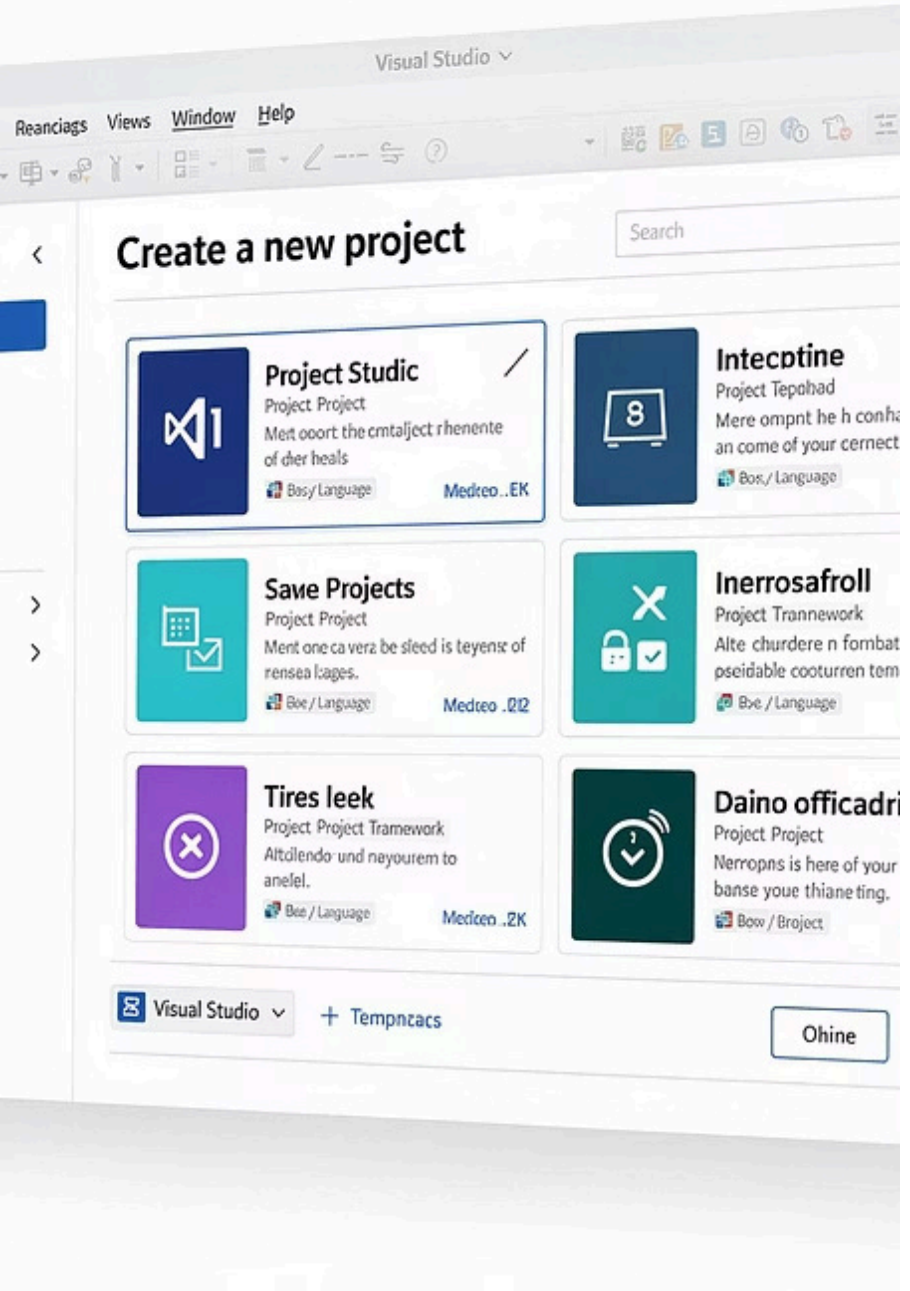
Debug efficiently, handle errors gracefully, and write production-ready code following industry standards

Why This Unit Matters

C#.NET is one of the most widely used frameworks in enterprise software development. Understanding its core features is essential for building robust, scalable applications.

This unit bridges the gap between basic programming concepts and professional development practices. You'll learn not just the syntax, but the architectural thinking that makes .NET applications maintainable and efficient.

The skills covered here form the foundation for web development with ASP.NET, database integration with ADO.NET, and modern cloud applications on Azure.



Creating Your First .NET Project

Every .NET application starts with a project. Think of a project as a container that holds all your code files, resources, and configuration settings. Visual Studio provides several project templates, each designed for different types of applications.

Console Application

Command-line programs that run in a terminal. Perfect for learning, testing concepts, and building utility tools.

Class Library

Reusable code packages that other projects can reference. Think of them as toolboxes other developers can use.

Understanding the .csproj File

The .csproj file is your project's blueprint. It tells Visual Studio and the .NET compiler everything about your project: which .NET version to use, what packages are installed, and how to build your application.

This XML file is automatically generated but understanding its structure helps you troubleshoot issues and configure advanced project settings.

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
 <OutputType>Exe</OutputType>
 <TargetFramework>net6.0</TargetFramework>
 </PropertyGroup>
</Project>
```

The Build and Run Process

Write Code

Author C# source in Visual Studio.

Compile to IL

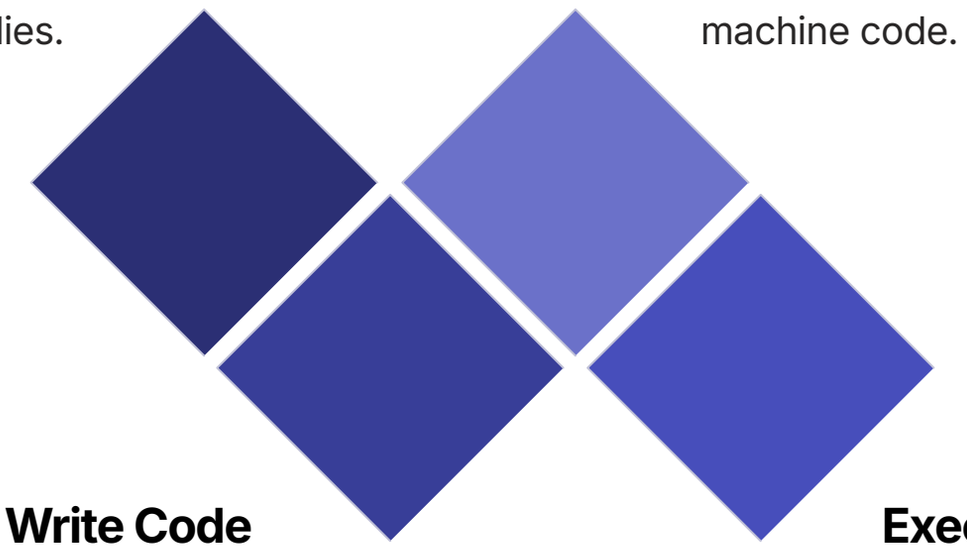
Compiler emits Intermediate Language assemblies.

JIT Compilation

JIT converts IL to native machine code.

Execute on CLR

CLR runs native code with runtime services.



When you press "Run" in Visual Studio, your C# code goes through several transformations. First, the compiler converts your human-readable code into Intermediate Language (IL). Then, at runtime, the Just-In-Time (JIT) compiler converts IL into machine code specific to your computer's processor. This two-step process allows .NET applications to run on any platform that has the .NET runtime installed.

Namespaces: Organizing Your Code

Think of It Like This

Namespaces are like folders on your computer. Just as you organize documents into folders to avoid confusion, namespaces organize classes to prevent naming conflicts.

Without namespaces, if two developers create a class called "Student," the compiler wouldn't know which one to use.

Real-World Example

```
namespace SchoolManagement.Students
{
 public class Student
 {
 public string Name { get; set; }
 public int RollNumber { get; set; }
 }
}

namespace SchoolManagement.Teachers
{
 public class Teacher
 {
 public string Name { get; set; }
 public string Subject { get; set; }
 }
}
```

Using Namespaces in Your Code

The `using` directive tells the compiler where to look for classes you're referencing. Instead of writing the full path every time, you can import the namespace once at the top of your file.

Without Using Directive

```
System.Console.WriteLine("Hello");
System.Collections.Generic.List<int> numbers =
    new System.Collections.Generic.List<int>();
```

With Using Directive

```
using System;
using System.Collections.Generic;

Console.WriteLine("Hello");
List<int> numbers = new List<int>();
```

Classes: The Building Blocks

A class is a blueprint for creating objects. It defines the properties (data) and methods (behaviors) that objects of that class will have. Think of a class as a cookie cutter and objects as the cookies it creates.

Properties

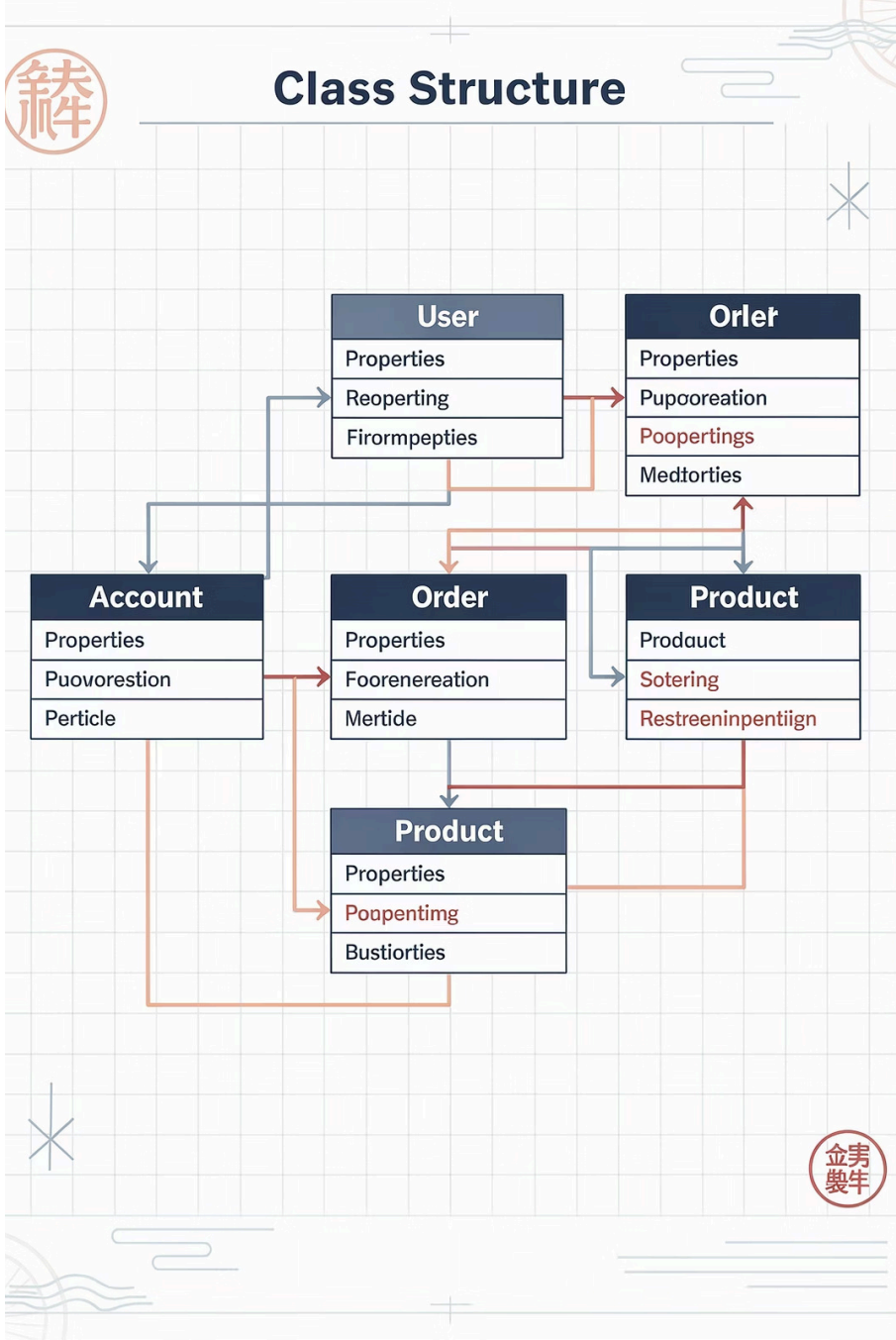
Store data about the object. For a Student class: Name, Age, RollNumber

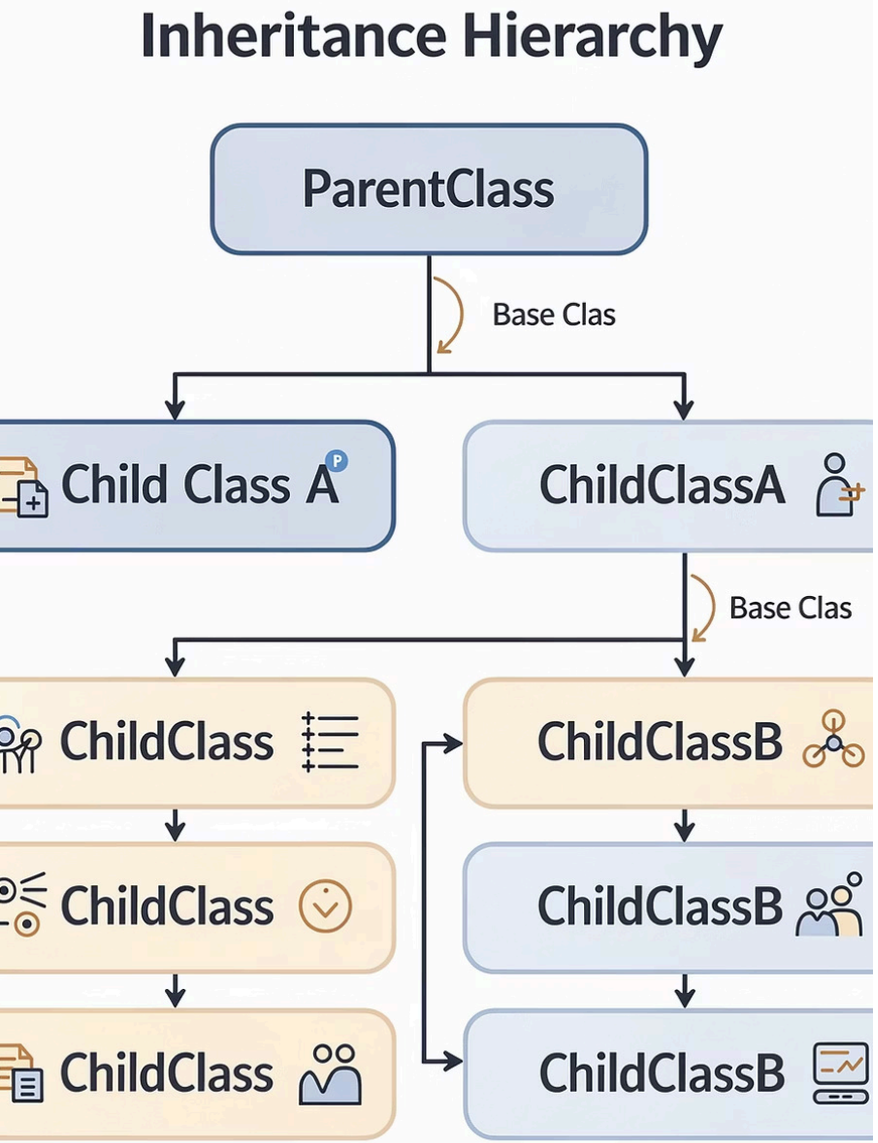
Methods

Define what the object can do. For a Student class: Study(), AttendClass(), TakeExam()

Constructors

Special methods that run when you create a new object. They initialize the object's properties.





Inheritance in Practice

Inheritance allows you to create new classes based on existing ones, inheriting their properties and methods. This promotes code reuse and establishes natural hierarchies in your programs.

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

 public void Introduce()
 {
 Console.WriteLine($"Hi, I'm {Name}");
 }
}

public class Student : Person
{
 public int RollNumber { get; set; }
 public string Major { get; set; }

 public void Study()
 {
 Console.WriteLine($"{Name} is studying {Major}");
 }
}
```

C# is Strongly Typed: What Does That Mean?

C# is a strongly typed language, meaning every variable must have a declared type, and the compiler checks that you're using variables correctly. This catches errors at compile-time rather than runtime.

Think of it like labeling containers in your kitchen. If you label a jar "Sugar," you won't accidentally put salt in it. The compiler acts as your quality control inspector.

Safe

Prevents type-related bugs before your program runs

Clear

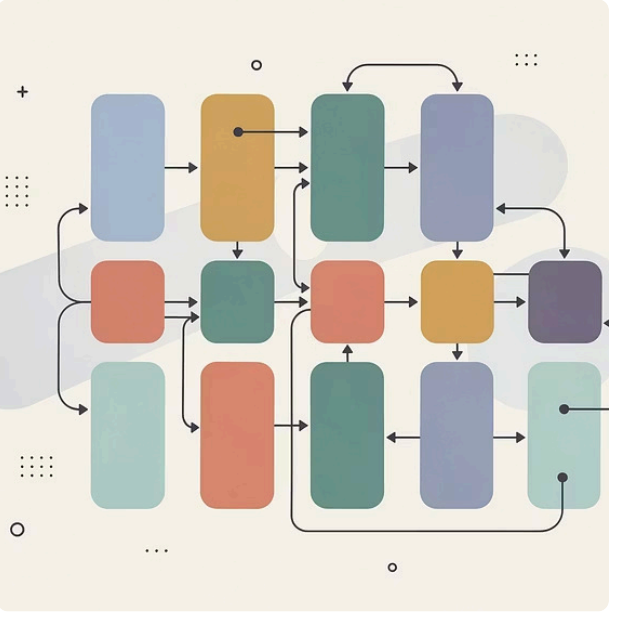
Code is easier to read and understand

Fast

Compiler optimizations make code run faster

Value Types vs Reference Types

Understanding how data is stored in memory is crucial for writing efficient code. C# has two fundamental ways of storing data, each with different behaviors.



Value Types

Stored directly in memory. Include int, float, bool, struct. When you copy a value type, you get a complete duplicate.

Reference Types

Stored as a reference (pointer) to memory. Include classes, arrays, strings. Copying creates another reference to the same data.

Value vs Reference: A Practical Example

Value Type Behavior

```
int a = 10;
int b = a;
b = 20;

Console.WriteLine(a); // 10
Console.WriteLine(b); // 20
```

Changing `b` doesn't affect `a` because they're independent copies.

Reference Type Behavior

```
Student student1 = new Student();
student1.Name = "John";

Student student2 = student1;
student2.Name = "Jane";

Console.WriteLine(student1.Name); // Jane
```

Both variables point to the same object in memory. Changing one affects the other.

Core Data Types in C#

Numeric Types

int: Whole numbers (-2 billion to 2 billion)

float: Decimal numbers, 7 digits precision

double: Decimal numbers, 15 digits precision

decimal: High precision for financial calculations

Text Types

char: A single character ('A', '9', '$')

string: Sequence of characters ("Hello World")

Boolean

bool: True or false values, used for conditions and logic

Special Types

object: Base type for all types in C#

dynamic: Type checking happens at runtime, not compile-time

Type Conversion: Implicit vs Explicit

Sometimes you need to convert data from one type to another. C# supports two types of conversion based on whether data might be lost in the process.

Implicit Casting (Automatic)

Happens automatically when no data will be lost. You're moving from a smaller container to a larger one.

```
int myInt = 100;
double myDouble = myInt; // Safe
long myLong = myInt;     // Safe

Console.WriteLine(myDouble); // 100.0
```

Explicit Casting (Manual)

Required when data might be lost. You must explicitly tell the compiler you understand the risk.

```
double myDouble = 9.78;
int myInt = (int)myDouble; // Loses decimal

Console.WriteLine(myInt); // 9

// Use Convert for safer conversion
int safer = Convert.ToInt32(myDouble); // 10
```

The var Keyword: Type Inference

The `var` keyword lets the compiler automatically determine a variable's type based on the value you assign. This doesn't make C# dynamically typed—the type is still determined at compile time and cannot change.

```
var number = 42;          // int
var name = "Alice";       // string
var price = 99.99;        // double
var isActive = true;      // bool

// Still strongly typed!
number = "text"; // ERROR: Cannot convert string to int
```

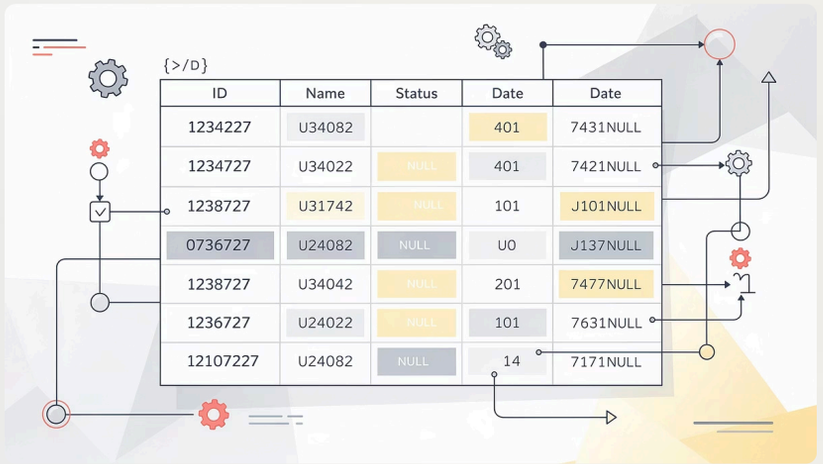
Best Practice: Use `var` when the type is obvious from the right-hand side. Avoid it when clarity would be lost.

When to Use var

- Type is obvious from initialization
- Long type names become cumbersome
- Working with anonymous types

When to Avoid var

- Type isn't immediately clear
- Clarity is more important than brevity



Nullable Types: Handling Missing Data

By default, value types like int and bool must always have a value. But what if you need to represent "no value" or "unknown"? Nullable types solve this problem by allowing value types to be null.

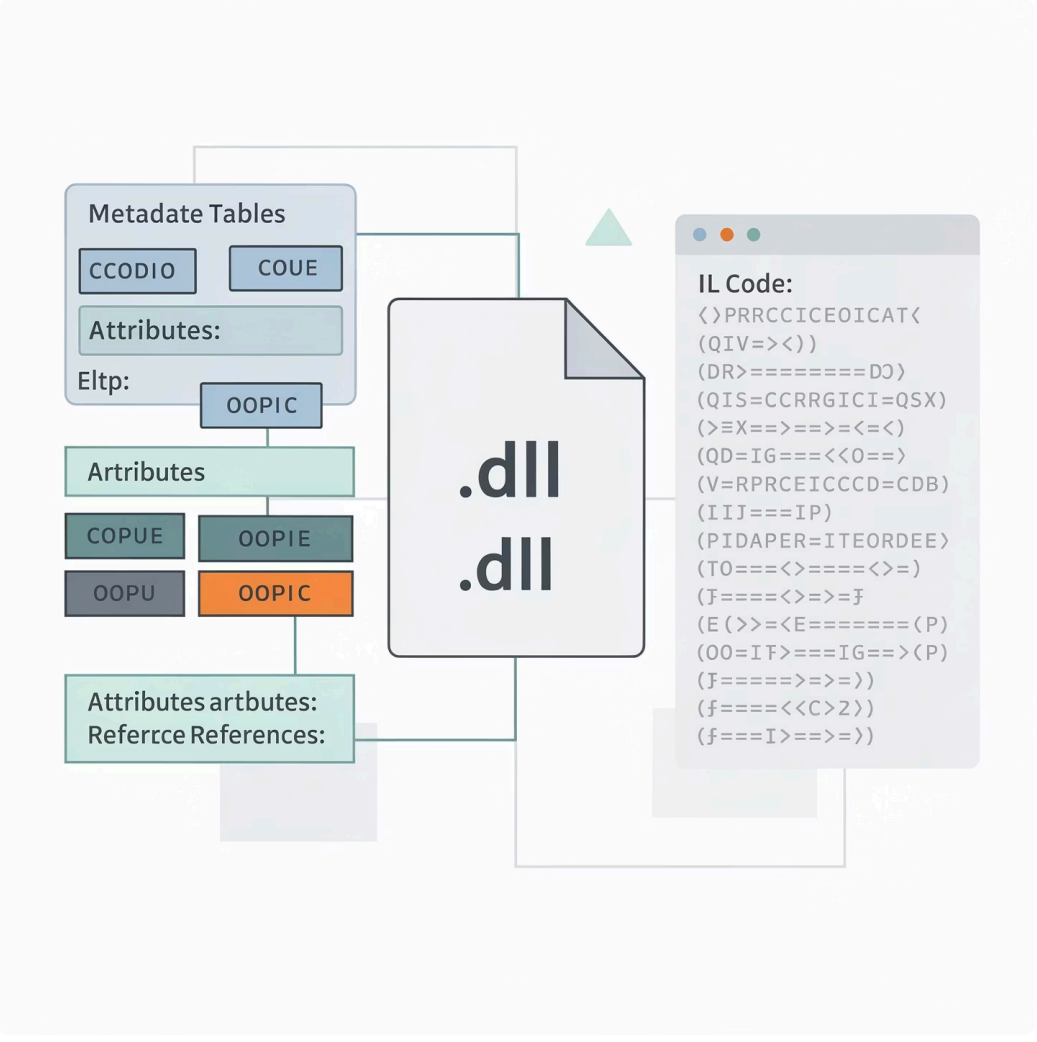
Declaring Nullable Types

```
int? age = null;
double? salary = null;
bool? isApproved = null;

// Check if value exists
if (age.HasValue)
{
    Console.WriteLine($"Age is
{age.Value}");
}
else
{
    Console.WriteLine("Age is
unknown");
}
```

Null-Coalescing Operator

```
int? userInput = null;

// Use default value if null
int result = userInput ?? 0;
Console.WriteLine(result); // 0

// Or chain multiple checks
int value = input1 ?? input2 ?? input3
?? 100;
```

Assemblies: The Building Blocks of .NET



An assembly is a compiled code library that contains your application or reusable components. Think of it as a packaged unit of functionality that can be deployed and versioned independently.

Every .NET application consists of one or more assemblies. When you build your project, Visual Studio creates an assembly file (.exe or .dll) containing your compiled code.

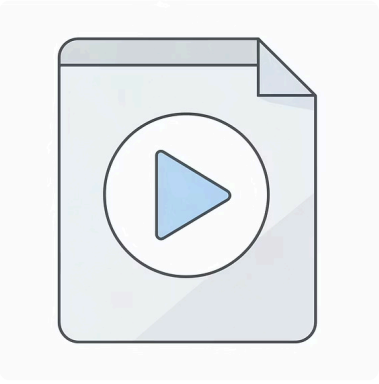
Contains

IL code, metadata, resources, and manifest

Provides

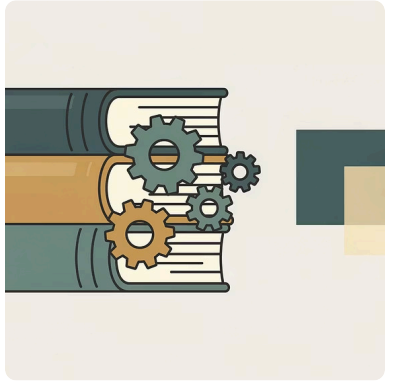
Type information, security boundaries, versioning

DLL vs EXE Assemblies



EXE (Executable)

A standalone program that can run on its own. Contains a Main() entry point where execution begins. Examples: console applications, desktop applications.



DLL (Dynamic Link Library)

A library of code that other programs can use. Cannot run independently—must be referenced by an EXE. Examples: class libraries, shared utilities, third-party packages.

Assembly Metadata and Manifest

Every assembly contains more than just code. It includes metadata that describes the assembly itself, its dependencies, and the types it contains.

1

Manifest

The assembly's identity: name, version number, culture information, and public key. Lists all files in the assembly and references to other assemblies.

2

Type Metadata

Information about every type (class, interface, enum) in the assembly: their members, attributes, and relationships.

3

IL Code

The actual compiled Intermediate Language code that gets executed by the .NET runtime.

4

Resources

Embedded files like images, strings, or configuration data that the application uses.

Exploring the Base Class Library (BCL)

The Base Class Library is a vast collection of pre-written, tested, and optimized code provided by Microsoft. Instead of writing common functionality from scratch, you can leverage thousands of classes that handle everything from file operations to network communication.

Think of the BCL as a massive toolbox. Need to work with dates? Use System.DateTime. Need to read files? Use System.IO. Need to make web requests? Use System.Net. Learning to navigate and use the BCL efficiently is a key skill for .NET developers.

Essential BCL Namespaces

System

Core types like String, Int32, Double, DateTime, Console, Math, and Exception. The foundation of every C# program.

System.Collections

Data structures like lists, dictionaries, queues, and stacks. Generic and non-generic collections for storing and managing data.

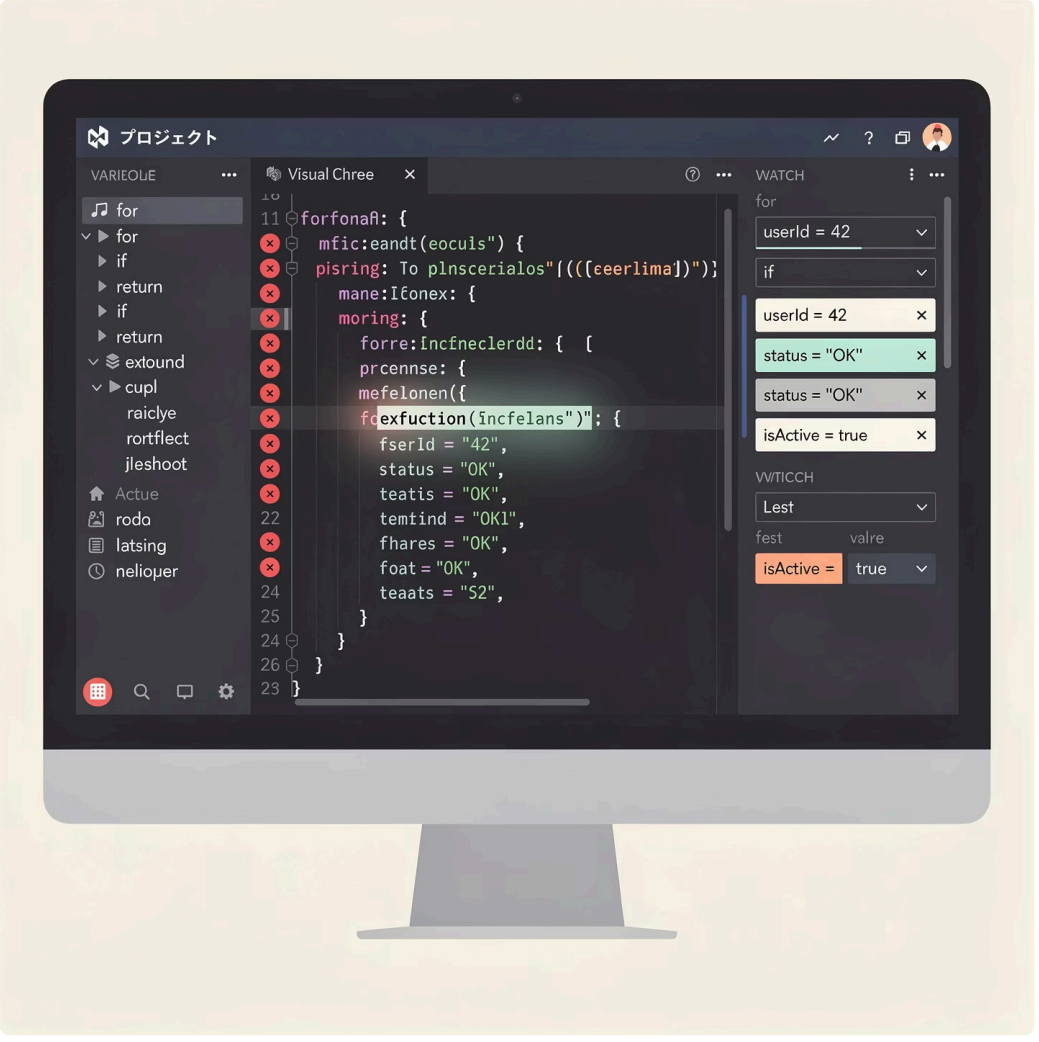
System.IO

File and directory operations: reading, writing, creating, and deleting files. Stream handling for efficient data processing.

System.Math

Mathematical functions and constants: trigonometry, logarithms, rounding, absolute values, and more.

Professional Debugging Techniques



Debugging is the process of finding and fixing errors in your code. Visual Studio provides powerful debugging tools that let you pause execution, inspect variables, and step through code line by line.

Professional developers spend significant time debugging. Mastering these tools makes you dramatically more efficient at understanding code behavior and identifying issues.

Essential Debugging Tools

Breakpoints

Pause execution at a specific line. Click in the left margin or press F9. When code hits a breakpoint, execution stops and you can inspect the current state.

Step Over (F10)

Execute the current line and move to the next line. Treats function calls as a single step—doesn't dive into them.

Step Into (F11)

Execute the current line. If it's a function call, enter that function and start debugging inside it.

Watch Window

Monitor specific variables while debugging. Add any expression and see its value update as you step through code.

Locals Window

Automatically displays all variables in the current scope and their values. Essential for understanding the current state.

Error Handling with Try-Catch-Finally

Errors happen in every program—files don't exist, networks fail, users enter invalid data. Professional applications handle these errors gracefully instead of crashing. The try-catch-finally pattern provides structured error handling.

```
try
{
    // Code that might cause an error
    int[] numbers = { 1, 2, 3 };
    Console.WriteLine(numbers[5]); // Error!
}
catch (IndexOutOfRangeException ex)
{
    // Handle specific error
    Console.WriteLine("Array index out of bounds");
    Console.WriteLine($"Details: {ex.Message}");
}
catch (Exception ex)
{
    // Handle any other error
    Console.WriteLine($"Unexpected error: {ex.Message}");
}
finally
{
    // Always runs, error or not
    Console.WriteLine("Cleanup complete");
}
```

Three Blocks

- **try:** Contains code that might fail
- **catch:** Handles specific error types
- **finally:** Cleanup code that always runs

Best Practice: Catch specific exceptions first, then use a general Exception catch as a fallback.

Creating Custom Exceptions

Sometimes built-in exceptions don't adequately describe your application's error conditions. Custom exceptions let you create meaningful error types specific to your domain.

For example, in a banking application, you might create InsufficientFundsException or InvalidAccountException to clearly communicate what went wrong.

```
public class InvalidAgeException : Exception
{
 public InvalidAgeException() { }

 public InvalidAgeException(string message)
 : base(message) { }

 public InvalidAgeException(string message,
 Exception inner) : base(message, inner) { }
}

// Using the custom exception
if (age < 0 || age > 150)
{
 throw new InvalidAgeException(
 $"Age {age} is not valid. Must be 0-150.");
}
```

String Manipulation Essentials

Strings are fundamental to almost every application—user input, file processing, data formatting, and display output all involve string manipulation. C# provides rich string functionality through the String class.

Case Conversion

ToUpper(), ToLower() change the case of all characters

Substring

Extract portion of string: "Hello".Substring(0, 3) → "Hel"

Replace

Substitute text: "cat".Replace("c", "b") → "bat"

Split

Break into array: "a,b,c".Split(',') → ["a", "b", "c"]

Trim

Remove whitespace from ends: " text ".Trim() → "text"

Contains

Check if substring exists: "Hello".Contains("ell") → true

StringBuilder for Efficient String Building

The Problem with String Concatenation

Strings in C# are immutable—once created, they cannot change. When you concatenate strings in a loop, you're creating new string objects every iteration, wasting memory and CPU.

```
// Inefficient: creates 1000 new strings
string result = "";
for (int i = 0; i < 1000; i++)
{
    result += i.ToString(); // New string each time
}
```

The StringBuilder Solution

StringBuilder is a mutable character buffer optimized for building strings. It modifies the same buffer instead of creating new objects.

```
// Efficient: uses single buffer
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++)
{
    sb.Append(i.ToString()); // Modifies buffer
}
string result = sb.ToString();
```

Rule of thumb: Use StringBuilder when building strings in loops or concatenating more than a few strings.

String Formatting Techniques

Professional applications need to format data for display—currency, dates, numbers with specific decimal places. C# offers several formatting approaches.

String Interpolation (Recommended)

```
string name = "Alice";
int age = 25;
double gpa = 3.856;

string msg = $"Student: {name}, Age: {age}";
string precise = $"GPA: {gpa:F2}"; // 3.86

Console.WriteLine(msg);
```

Composite Formatting

```
string msg = string.Format(
 "Student: {0}, Age: {1}",
 name, age);

// With alignment and format specifiers
string formatted = string.Format(
 "{0,-10} {1,5:C}",
 "Price:", 49.99); // Price: $49.99
```

File Handling: Reading and Writing Data

Most applications need to persist data beyond program execution. The System.IO namespace provides classes for file operations—creating, reading, writing, and deleting files.

File I/O is essential for saving user data, processing large datasets, generating reports, and integrating with other systems. Understanding these operations is crucial for building practical applications.

Simple File Operations

Writing Text to a File

```
using System.IO;

// Write entire string to file
string content = "Hello, World!";
File.WriteAllText("output.txt", content);

// Write lines array
string[] lines = {
    "Line 1",
    "Line 2",
    "Line 3"
};
File.WriteAllLines("data.txt", lines);
```

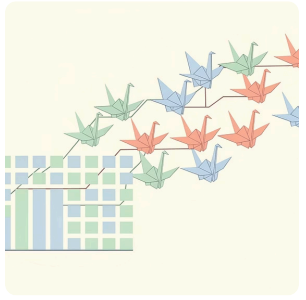
Reading Text from a File

```
using System.IO;

// Read entire file as string
string content = File.ReadAllText("output.txt");
Console.WriteLine(content);

// Read file as array of lines
string[] lines = File.ReadAllLines("data.txt");
foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

Always wrap file operations in try-catch blocks. Files might not exist, permissions might be denied, or disk space might be full.

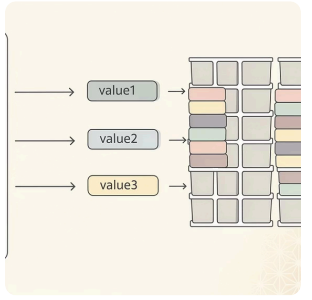
Collections: Managing Dynamic Data

Arrays have a fixed size determined at creation. Collections are dynamic data structures that can grow and shrink as needed. They're essential for managing data when you don't know the size in advance.



List<T>

Generic dynamic array. Add, remove, search items efficiently. Most commonly used collection type.



Dictionary<K,V>

Key-value pairs for fast lookup by key. Perfect for mapping relationships like ID→Student or word→definition.

Queue<T>

First-In-First-Out (FIFO) structure. Add to back, remove from front. Used for task scheduling, message processing.

Stack<T>

Last-In-First-Out (LIFO) structure. Add and remove from top. Used for undo operations, parsing expressions.

Working with Lists and Dictionaries

List<T> Example

```
using System.Collections.Generic;

List<string> students = new List<string>();

// Add items
students.Add("Alice");
students.Add("Bob");
students.Add("Charlie");

// Access items
Console.WriteLine(students[0]); // Alice

// Remove items
students.Remove("Bob");

// Count items
Console.WriteLine(students.Count); // 2

// Iterate
foreach (string student in students)
{
    Console.WriteLine(student);
}
```

Dictionary<K,V> Example

```
using System.Collections.Generic;

Dictionary<int, string> students =
 new Dictionary<int, string>();

// Add key-value pairs
students.Add(101, "Alice");
students.Add(102, "Bob");
students[103] = "Charlie"; // Alternative syntax

// Access by key
Console.WriteLine(students[101]); // Alice

// Check if key exists
if (students.ContainsKey(102))
{
 Console.WriteLine("Student found");
}

// Iterate
foreach (var kvp in students)
{
 Console.WriteLine($"{kvp.Key}: {kvp.Value}");
}
```

Key Takeaways

Project Structure Mastery

You can now create well-organized .NET projects, understand assemblies and namespaces, and structure code for maintainability and reuse.

Language Fundamentals

You understand C#'s type system, the difference between value and reference types, type conversion, and how to work with nullable types safely.

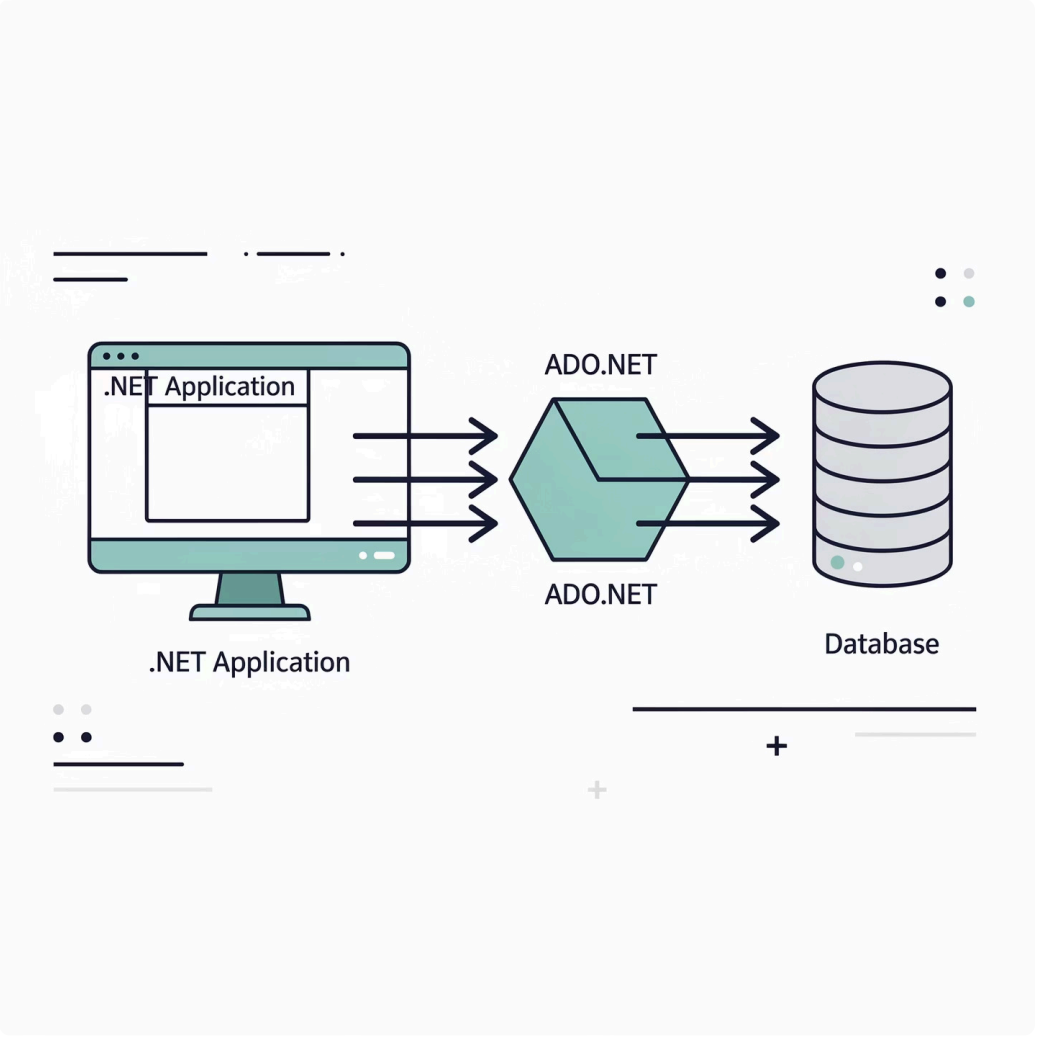
BCL Proficiency

You can leverage the Base Class Library for common tasks, manipulate strings efficiently, read and write files, and manage data with collections.

Professional Development Practices

You know how to debug effectively using Visual Studio tools, handle errors gracefully with structured exception handling, and write robust production-ready code.

What's Next: ADO.NET and Database Integration



In the next unit, you'll learn to connect C# applications to databases using ADO.NET. You'll execute SQL queries, retrieve data, perform CRUD operations, and build data-driven applications.

The foundation you've built in this unit—understanding types, collections, error handling, and file I/O—directly applies to database programming. ADO.NET uses the same patterns with different objects: instead of files, you'll work with databases; instead of text lines, you'll work with data records.

Get ready to bring your applications to life by integrating persistent data storage!