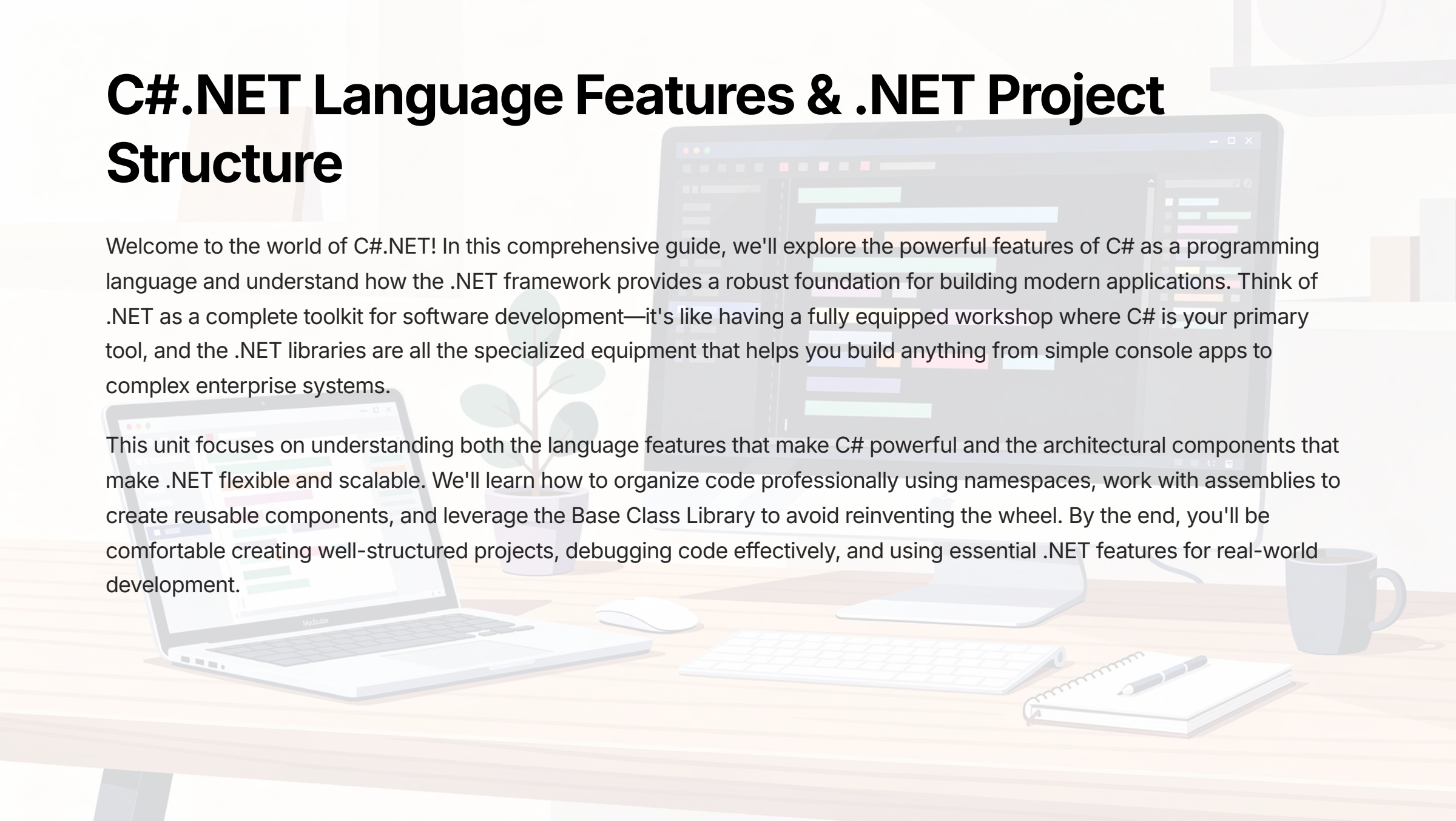
C#.NET Language Features & .NET Project Structure



Welcome to the world of C#.NET! In this comprehensive guide, we'll explore the powerful features of C# as a programming language and understand how the .NET framework provides a robust foundation for building modern applications. Think of .NET as a complete toolkit for software development—it's like having a fully equipped workshop where C# is your primary tool, and the .NET libraries are all the specialized equipment that helps you build anything from simple console apps to complex enterprise systems.

This unit focuses on understanding both the language features that make C# powerful and the architectural components that make .NET flexible and scalable. We'll learn how to organize code professionally using namespaces, work with assemblies to create reusable components, and leverage the Base Class Library to avoid reinventing the wheel. By the end, you'll be comfortable creating well-structured projects, debugging code effectively, and using essential .NET features for real-world development.

Creating Your First .NET Project

Creating a .NET project is the foundation of all C# development. When you create a project in Visual Studio, you're essentially setting up a structured workspace where all your code, resources, and configuration files will live. There are different project types available, but the two most fundamental ones are Console Applications and Class Libraries. A Console Application is a program that runs in a command-line window—perfect for learning and building utility tools. A Class Library, on the other hand, creates reusable code components (DLL files) that other applications can reference and use.

Console Application

Creates an executable (.exe) program that runs in the terminal. Used for command-line tools, utilities, and learning projects. Has a Main() method as the entry point.

Class Library

Creates a reusable library (.dll) containing classes and methods. Cannot run independently—must be referenced by other projects. Used for shared code and component development.

Every .NET project contains a project file with the .csproj extension. Think of this file as the blueprint for your project—it tells the build system what files to include, which .NET version to target, what dependencies you need, and other important configuration details. When you click "Build" in Visual Studio, the compiler reads this file, compiles your C# code into Intermediate Language (IL), and packages everything into an assembly (either a .exe or .dll file). This build process is what transforms your human-readable code into something the .NET runtime can execute.

Project Structure

- Program.cs - Main entry point
- ProjectName.csproj - Project configuration
- bin/ - Compiled output files
- obj/ - Intermediate build files
- Properties/ - Assembly metadata

Build Process

1. Compiler reads .csproj file
2. Compiles .cs files to IL code
3. Creates assembly with metadata
4. Outputs to bin/Debug or bin/Release
5. Ready to execute via .NET runtime

```
// Basic Console Application Structure
using System;

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

The code above shows the minimal structure of a C# console application. The `using System;` statement imports the System namespace, giving you access to fundamental classes like Console. The `namespace` declaration organizes your code, and the `Main()` method is where program execution begins. Every console application must have exactly one Main() method—it's the doorway through which the .NET runtime enters your application.

Namespaces and Classes: Organizing Your Code

As projects grow larger, organizing code becomes critical. Imagine trying to find a specific book in a library where thousands of books are piled randomly on the floor—it would be chaos! Namespaces solve this problem in C# by providing a hierarchical organization system for your classes, similar to how folders organize files on your computer. A namespace groups related classes together and prevents naming conflicts when you have multiple classes with the same name in different parts of your application.

For example, you might have a `Student` class in a university management system, but you might also need a `Student` class in an online course platform. Without namespaces, these would conflict. By placing one in `UniversitySystem.Models` and another in `OnlineCourses.Entities`, you can use both without any problems. The full name becomes `UniversitySystem.Models.Student` and `OnlineCourses.Entities.Student`—completely distinct identities.

Why Use Namespaces?

- Prevent naming conflicts between classes
- Logically group related functionality
- Make code easier to navigate and maintain
- Control access and visibility of types

Namespace Naming Convention

- Use PascalCase (each word capitalized)
- Follow pattern: Company.Product.Feature
- Example: Microsoft.AspNetCore.Mvc
- Be descriptive but concise

```
// File: Models/Student.cs
namespace UniversitySystem.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
    }
}

// File: Services/StudentService.cs
using UniversitySystem.Models; // Import the namespace

namespace UniversitySystem.Services
{
    public class StudentService
    {
        public void EnrollStudent(Student student)
        {
            // Use Student class from Models namespace
            Console.WriteLine($"Enrolling {student.Name}");
        }
    }
}
```

The `using` directive at the top of a file allows you to reference classes from other namespaces without typing the full namespace path every time. Instead of writing `UniversitySystem.Models.Student` repeatedly, you can just write `Student` after adding `using UniversitySystem.Models;` at the top. This keeps your code clean and readable while maintaining the organizational benefits of namespaces.

Best Practice: Organize your files in folders that mirror your namespace structure. If you have a namespace called `MyApp.Data.Repositories`, create folders Data/Repositories/ and place the corresponding class files there. This makes your project intuitive to navigate.

Inheritance: Building on Existing Classes

Inheritance is one of the fundamental pillars of object-oriented programming, allowing you to create new classes based on existing ones. Think of inheritance like genetic inheritance in biology—a child inherits characteristics from their parents but can also have their own unique features. In C#, when a class inherits from another class, it automatically gets all the public and protected members (fields, properties, methods) of the parent class, and can add its own specialized functionality.

This concept is incredibly powerful for code reuse and creating hierarchical relationships. For example, you might have a base `Person` class with common properties like Name and Age. Then you can create specialized classes like `Student` and `Teacher` that inherit from Person, automatically getting those basic properties while adding their own specific ones like StudentId or Subject.

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

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

// Derived Class
public class Student : Person
{
    public string StudentId { get; set; }
    public double GPA { get; set; }

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

Inheritance Benefits

- Eliminates code duplication
- Establishes IS-A relationships
- Enables polymorphism
- Improves maintainability

Key Points

- Use colon (:) to inherit
- C# supports single inheritance only
- Child class gets all public/protected members
- Can override virtual methods

```
// Using the inherited classes
class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
        student.Name = "Sarah";       // Inherited from Person
        student.Age = 20;             // Inherited from Person
        student.StudentId = "S12345";  // Student's own property
        student.GPA = 3.8;            // Student's own property

        student.Introduce();  // Inherited method
        student.Study();      // Student's own method
    }
}
```

When designing class hierarchies, follow the "IS-A" rule: inheritance should only be used when the derived class truly IS-A type of the base class. A Student IS-A Person, so inheritance makes sense. But a Car is not a Person, so inheriting Car from Person would be incorrect. In C#, a class can only inherit from one base class (single inheritance), but can implement multiple interfaces, giving you flexibility in design while maintaining simplicity.

1

Base Class

Defines common behavior

2

Derived Class

Inherits and extends

3

Specialized Use

Polymorphic behavior

Understanding Data Types in C#

C# is a strongly typed language, which means every variable must have a specific data type declared when it's created, and that type cannot change. This is like labeling storage containers in a warehouse—once you designate a container for "electronics," you can't suddenly start storing food in it. This strictness helps prevent errors and makes your code more predictable and maintainable. Understanding data types is crucial because choosing the right type affects memory usage, performance, and what operations you can perform on your data.

Data types in C# fall into two major categories: value types and reference types. Value types store their data directly in memory (on the stack), while reference types store a reference (memory address) to where the actual data lives (on the heap). This distinction affects how variables are copied, compared, and managed in memory. Let's explore the most commonly used data types you'll work with in everyday programming.

int

Integer numbers from -2,147,483,648 to 2,147,483,647. Perfect for counting, IDs, ages.

`int age = 25;`

double

Decimal numbers with high precision. Use for scientific calculations, measurements.

`double price = 99.99;`

string

Text data, sequence of characters. Use for names, messages, any text content.

`string name = "John";`

bool

True or false values. Perfect for flags, conditions, yes/no scenarios.

`bool isActive = true;`

char

Single character enclosed in single quotes. Stores one Unicode character.

`char grade = 'A';`

decimal

High-precision decimal for money. More accurate than double for financial calculations.

`decimal salary = 50000.50m;`

```
// Data Type Examples with Common Operations
class DataTypeDemo
{
    static void Main()
    {
        // Integer types
        int studentCount = 150;
        long worldPopulation = 8000000000L;

        // Floating-point types
        double pi = 3.14159;
        float temperature = 98.6f;
        decimal accountBalance = 1250.75m;

        // Boolean
        bool isPassed = true;
        bool isRaining = false;

        // Character and String
        char initial = 'J';
        string fullName = "John Doe";

        // Performing operations
        int total = studentCount + 50;
        double area = pi * 5 * 5;
        string greeting = "Hello, " + fullName;

        Console.WriteLine($"Total students: {total}");
        Console.WriteLine($"Circle area: {area}");
        Console.WriteLine(greeting);
    }
}
```

Value Types

- int, double, float, decimal
- bool, char
- struct, enum
- Stored on stack
- Copying creates independent copy

Reference Types

- string, object
- class, interface
- arrays, delegates
- Stored on heap
- Copying shares same reference

C# also provides the `var` keyword for implicit typing, where the compiler automatically determines the type based on the assigned value. For example, `var number = 10;` creates an int, and `var name = "Alice";` creates a string. This doesn't make C# dynamically typed—the type is still fixed at compile time; you're just letting the compiler figure it out. Use `var` when the type is obvious from the assignment to keep code concise, but avoid it when clarity would suffer.

Nullable Types: By default, value types cannot be null. If you need to represent "no value," use nullable types with the ? syntax: `int? age = null;`. This is essential when working with databases where fields might be empty.

Type Conversion: Implicit vs Explicit Casting

Type conversion is the process of converting a value from one data type to another. This happens frequently in programming—you might need to convert user input from a string to a number, or combine an integer with a decimal in a calculation. C# provides two types of conversion: implicit (automatic) and explicit (manual). Understanding when each type occurs and how to use them correctly is essential for writing robust code that handles data transformations safely.

Implicit conversion happens automatically when there's no risk of data loss. Think of it like pouring water from a small cup into a large bucket—everything fits without problems. For example, converting an int to a double happens implicitly because every integer value can be represented as a double. The compiler does this conversion for you automatically, no special syntax required. However, going the other direction requires explicit conversion because you might lose the decimal portion.

01

Implicit Conversion

Safe conversions that happen automatically without data loss. From smaller to larger types.

02

Explicit Conversion (Casting)

Manual conversions where data loss is possible. Requires cast operator with parentheses.

03

Conversion Methods

Using Convert class or Parse methods for string-to-type conversions with error handling.

```
// Implicit Conversion Examples (Automatic)
int wholeNumber = 100;
double decimalNumber = wholeNumber;  // int → double (safe)
Console.WriteLine(decimalNumber);    // Output: 100.0

float smallDecimal = 5.5f;
double largeDecimal = smallDecimal;  // float → double (safe)

// Explicit Conversion (Manual Casting)
double price = 99.99;
int roundedPrice = (int)price;       // double → int (loses .99)
Console.WriteLine(roundedPrice);     // Output: 99

long bigNumber = 5000000000L;
int smallerNumber = (int)bigNumber;  // Risky: might overflow

// Using Convert Class (Recommended for safety)
string numberText = "42";
int number = Convert.ToInt32(numberText);
double piValue = Convert.ToDouble("3.14159");
bool isTrue = Convert.ToBoolean("true");

// Using Parse Methods
string ageText = "25";
int age = int.Parse(ageText);
double temperature = double.Parse("98.6");

// TryParse for Safe Conversion (handles errors gracefully)
string input = "not a number";
int result;
if (int.TryParse(input, out result))
{
    Console.WriteLine($"Conversion successful: {result}");
}
else
{
    Console.WriteLine("Invalid input, conversion failed");
}
```

When to Use Each Method

Implicit: Happens automatically, no action needed

Casting: When you're certain about the conversion and accept potential data loss

Convert class: Converting between different types, especially from strings

TryParse: User input or external data where validity is uncertain

Common Conversion Scenarios

- User input (string) → int/double
- Math operations mixing int and double
- Database values → C# types
- API responses → typed objects
- Displaying numbers as formatted strings

The `TryParse` method is particularly useful when converting user input or data from external sources. Instead of throwing an exception when conversion fails (like `Parse` does), it returns a boolean indicating success or failure and outputs the converted value through an `out` parameter. This pattern lets you handle invalid input gracefully without crashing your program—essential for building user-friendly applications.

Common Mistake: Trying to assign a double to an int without casting will cause a compilation error. Always use explicit casting with `(int)` when converting from larger to smaller types, and be aware you'll lose the decimal portion.

Assemblies: The Building Blocks of .NET Applications

An assembly is the fundamental unit of deployment in .NET—it's the compiled output of your code packaged into a reusable format. Think of an assembly as a shipping container for your code: it packages everything needed to run your program into a single file (or set of files) that can be deployed, versioned, and executed. When you build a C# project, the compiler produces an assembly—either an executable (.exe) that can run independently, or a library (.dll) that other programs can use.

Every assembly contains four key components: the code itself (compiled into Intermediate Language or IL), metadata about the types defined in the assembly, a manifest that describes the assembly's identity and dependencies, and optional resources like images or text files. The manifest is particularly important—it's like a shipping label that tells the .NET runtime what this assembly is called, what version it is, what other assemblies it depends on, and what security permissions it needs.

Executable Assembly (.exe)

Contains a Main() entry point and can run independently. Used for applications that users directly execute.

Library Assembly (.dll)

Contains reusable code but cannot run on its own. Must be referenced by executable projects to be used.

Assembly Manifest

Metadata describing the assembly's identity, version, dependencies, and security requirements.

```
// Viewing Assembly Information in Code
using System;
using System.Reflection;

class AssemblyDemo
{
    static void Main()
    {
        // Get information about the current assembly
        Assembly currentAssembly = Assembly.GetExecutingAssembly();

        Console.WriteLine("Assembly Name: " + currentAssembly.FullName);
        Console.WriteLine("Location: " + currentAssembly.Location);
        Console.WriteLine("Version: " + currentAssembly.GetName().Version);

        // List all types (classes) in the assembly
        Type[] types = currentAssembly.GetTypes();
        Console.WriteLine("\nTypes in this assembly:");
        foreach (Type type in types)
        {
            Console.WriteLine("  - " + type.Name);
        }
    }
}
```

Assembly Components

- **IL Code:** Compiled intermediate language
- **Metadata:** Type definitions and members
- **Manifest:** Assembly identity and version
- **Resources:** Embedded files and data

Why Assemblies Matter

- Enable code reuse across projects
- Support versioning and updates
- Provide deployment units
- Enforce security boundaries
- Enable side-by-side execution

Assemblies can be private or shared. Private assemblies are copied into each application's directory and used only by that application—this is the default and simplest approach. Shared assemblies (also called strong-named assemblies) are installed in the Global Assembly Cache (GAC) and can be used by multiple applications simultaneously. Shared assemblies require a strong name (created with a cryptographic key) to ensure versioning and prevent tampering.

When you reference another assembly in your project, you're telling the compiler "I need to use code from this other assembly." The manifest in your assembly will record this dependency, and at runtime, the .NET runtime will locate and load the referenced assemblies automatically. This dependency system allows you to build modular applications where different assemblies handle different responsibilities—one for data access, another for business logic, another for user interface, all working together seamlessly.

Key Concept: The metadata in assemblies is what makes .NET's reflection capabilities possible. At runtime, you can examine assemblies to discover what types they contain, what methods those types have, and even invoke methods dynamically without knowing about them at compile time.

Namespaces: Organizing Large Projects

As your .NET projects grow from simple programs to complex applications with hundreds or thousands of classes, organization becomes critical. Namespaces provide a hierarchical structure for grouping related code, much like how a file system uses folders to organize documents. Without namespaces, all your classes would exist in a flat global space, leading to naming conflicts and confusion. With namespaces, you can have multiple classes with the same name as long as they're in different namespaces—they become distinct entities with unique fully qualified names.

The .NET Framework itself is organized using a comprehensive namespace hierarchy. The root namespace is `System`, and underneath it are hundreds of sub-namespaces like `System.IO` for file operations, `System.Collections` for data structures, and `System.Net` for networking. This hierarchical naming follows a pattern: the more specific the functionality, the deeper the namespace. For example, `System.Collections.Generic` contains generic collection types, which is more specific than just `System.Collections`.

This hierarchical organization helps developers quickly locate functionality and understand the relationships between different components.

```
// Defining and using namespaces across multiple files

// File: DataAccess/Repository.cs
namespace MyCompany.ProjectName.DataAccess
{
    public class Repository
    {
        public void SaveData()
        {
            Console.WriteLine("Saving data to database");
        }
    }
}

// File: Business/OrderProcessor.cs
namespace MyCompany.ProjectName.Business
{
    // Import the DataAccess namespace
    using MyCompany.ProjectName.DataAccess;

    public class OrderProcessor
    {
        private Repository _repository = new Repository();

        public void ProcessOrder()
        {
            Console.WriteLine("Processing order");
            _repository.SaveData();
        }
    }
}

// File: Program.cs
using MyCompany.ProjectName.Business;

class Program
{
    static void Main()
    {
        OrderProcessor processor = new OrderProcessor();
        processor.ProcessOrder();
    }
}
```

1	2	3
Prevent Conflicts	**Logical Grouping**	**Access Control**
Multiple classes can have the same name if in different namespaces	Related classes organized together for easy discovery	Control visibility across project boundaries

Namespace Best Practices

- Follow Company.Product.Component pattern
- Use PascalCase naming convention
- Mirror folder structure in namespaces
- Keep namespace depth reasonable (3-5 levels)
- Avoid namespace names matching class names

Using Directive Features

```
// Standard using
using System;

// Alias for long namespace
using Proj = MyCompany.ProjectName;

// Static using (C# 6.0+)
using static System.Console;
WriteLine("No need for Console.");

// Global using (C# 10.0+)
global using System.Collections;
```

The `using` directive makes your code cleaner by allowing you to reference types without their full namespace path. Instead of writing `System.Collections.Generic.List<int>` throughout your code, you can add `using System.Collections.Generic;` at the top and just write `List<int>`. Modern C# also supports static using (to import static members) and global using (to apply a using statement across all files in a project), further reducing repetitive code.

Common Convention: The default namespace for a project matches the project name. When you create a new class file in Visual Studio, it automatically uses the appropriate namespace based on the folder location within your project structure.

Exploring the Base Class Library (BCL)

The Base Class Library (BCL) is .NET's treasure chest of pre-built functionality—thousands of classes that handle common programming tasks so you don't have to reinvent the wheel. Imagine building a house: you could forge your own nails, cut your own lumber, and create every component from scratch, or you could use manufactured materials and focus on the actual construction. The BCL provides those "manufactured materials" for software development—ready-to-use, tested, and optimized code for everything from reading files to making HTTP requests.

The BCL is organized into namespaces that group related functionality. Some of the most frequently used namespaces include `System` (fundamental types and basic operations), `System.IO` (file and stream operations), `System.Collections.Generic` (data structures like lists and dictionaries), `System.Text` (string manipulation and encoding), `System.Linq` (querying collections), and `System.Net` (networking operations). Learning to navigate and use the BCL effectively is one of the most valuable skills for a .NET developer.

System.IO

File and directory operations, streams, readers, and writers. Essential for reading/writing data to disk, managing files, and working with different data formats.

System.Collections.Generic

Type-safe data structures like List, Dictionary, Queue, and Stack. Use these instead of arrays for dynamic, resizable collections with rich functionality.

System.Text

String manipulation, StringBuilder for efficient string operations, and encoding/decoding for different character sets. Critical for processing text data.

System.Math

Mathematical operations like trigonometric functions, logarithms, power, rounding, and constants like Pi. Used in scientific and financial calculations.

System.Net

Network communication including HTTP requests, TCP/IP sockets, and email. Build web clients, REST API consumers, and networked applications.

System.Linq

Language Integrated Query for filtering, sorting, and transforming collections. Write expressive, SQL-like queries on any enumerable data source.

```
// BCL Usage Examples - Common Scenarios

using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;

class BCLDemo
{
    static void Main()
    {
        // System.Math - Mathematical operations
        double result = Math.Sqrt(16);
        double power = Math.Pow(2, 10);
        Console.WriteLine($"Square root: {result}, Power: {power}");

        // System.IO - File operations
        string filePath = "example.txt";
        File.WriteAllText(filePath, "Hello from BCL!");
        string content = File.ReadAllText(filePath);

        // System.Collections.Generic - Lists
        List students = new List
        {
            "Alice", "Bob", "Charlie", "Diana"
        };
        students.Add("Edward");

        // System.Linq - Querying collections
        var aStudents = students.Where(s => s.StartsWith("A"));

        // System.Text - StringBuilder for efficient concatenation
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < students.Count; i++)
        {
            builder.AppendLine($"{i + 1}. {students[i]}");
        }
        Console.WriteLine(builder.ToString());
    }
}
```

One of the most powerful aspects of the BCL is that it's consistent across different types of .NET applications. Whether you're building a console app, a web application, a mobile app, or a Windows desktop program, the same BCL classes work the same way. This consistency dramatically reduces the learning curve—once you learn how to use `List<T>` or `File` or `HttpClient`, you can apply that knowledge across any .NET project type.

Pro Tip: Before writing code to solve a common problem, check if the BCL already has a solution. Need to sort a list? Use `List.Sort()`. Need to format dates? Use `DateTime.ToString()`. The BCL documentation (docs.microsoft.com) is comprehensive and includes examples for almost every class and method.

Debugging and Error Handling in C#

Debugging is the process of finding and fixing errors in your code—it's an essential skill that separates professional developers from beginners. Visual Studio provides powerful debugging tools that let you pause execution, inspect variable values, step through code line by line, and understand exactly what's happening inside your program. Think of debugging like being a detective: you gather clues (variable values), follow leads (step through code paths), and eventually solve the mystery (find the bug).

The most fundamental debugging tool is the breakpoint—a marker you place on a line of code where you want execution to pause. When your program reaches a breakpoint during debugging, it stops completely, giving you a chance to examine the current state. You can inspect variable values, check object properties, evaluate expressions, and see the call stack (the sequence of method calls that led to this point). From a breakpoint, you can step to the next line, step into method calls to see their internal workings, or step over calls to skip their details.

Breakpoints

Click in the left margin to set a breakpoint. Program pauses at this line when debugging. Press F9 to toggle.

Step Into (F11)

Execute one line and follow into method calls to see their implementation. Use to debug inside methods.

Step Over (F10)

Execute one line but don't follow into method calls. Use when you trust the method works correctly.

Watch Window

Monitor specific variables or expressions as you step through code. See how values change over time.

While debugging helps you find errors during development, error handling ensures your program behaves gracefully when something goes wrong at runtime. Not all errors can be prevented—users might enter invalid data, files might not exist, network connections might fail. The try-catch-finally structure lets you anticipate potential errors, handle them appropriately, and ensure cleanup code always runs. Think of it like a safety net: the try block is the tightrope walk, the catch block is the net that catches you if you fall, and the finally block is the crew that cleans up regardless of what happened.

```
// Error Handling with Try-Catch-Finally
using System;
using System.IO;

class ErrorHandlingDemo
{
    static void Main()
    {
        // Example 1: Handling division by zero
        try
        {
            Console.Write("Enter a number: ");
            int number = int.Parse(Console.ReadLine());
            int result = 100 / number;
            Console.WriteLine($"100 / {number} = {result}");
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("Error: Cannot divide by zero!");
            Console.WriteLine($"Details: {ex.Message}");
        }
        catch (FormatException ex)
        {
            Console.WriteLine("Error: Please enter a valid number!");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Unexpected error: {ex.Message}");
        }
        finally
        {
            Console.WriteLine("Operation completed.");
        }

        // Example 2: File operations with error handling
        FileStream fs = null;
        try
        {
            fs = new FileStream("data.txt", FileMode.Open);
            // Read from file...
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("File not found. Creating new file.");
            fs = new FileStream("data.txt", FileMode.Create);
        }
        finally
        {
            // Always close the file, whether error occurred or not
            if (fs != null)
            {
                fs.Close();
                Console.WriteLine("File closed properly.");
            }
        }
    }
}
```

Common Exception Types

- `NullReferenceException` - Using a null object
- `IndexOutOfRangeException` - Invalid array index
- `FormatException` - Invalid data format
- `FileNotFoundException` - Missing file
- `DivideByZeroException` - Division by zero

Best Practices

- Catch specific exceptions before general ones
- Use finally for cleanup code (closing files, connections)
- Don't swallow exceptions silently (empty catch)
- Log exceptions for troubleshooting
- Throw custom exceptions for business logic errors

You can also create custom exceptions by inheriting from the Exception class. This is useful when you want to signal specific error conditions in your application that aren't covered by the standard exception types. For example, you might create an `InsufficientFundsException` for a banking application or a `InvalidStudentIdException` for a student management system.

Debugging Tip: Use the Immediate Window (Debug → Windows → Immediate) during a debugging session to execute code and evaluate expressions on the fly. You can call methods, change variable values, and test hypotheses without modifying your source code.

String Manipulation in C#

Strings are one of the most commonly used data types in programming—nearly every application works with text data in some form. In C#, the String class is immutable, meaning once created, a string cannot be changed. Any operation that appears to modify a string actually creates a new string object. This immutability ensures thread safety and prevents accidental modifications, but it also means that repeatedly concatenating strings in a loop is inefficient because each concatenation creates a new string object in memory.

The String class provides dozens of methods for manipulating text: searching, replacing, splitting, combining, changing case, trimming whitespace, and much more. Understanding these methods and when to use them is crucial for processing user input, formatting output, parsing data files, and countless other text-related tasks. Let's explore the most commonly used string operations with practical examples you'll encounter in real-world development.

Searching & Checking

Contains(), StartsWith(), EndsWith(), IndexOf() - Find text within strings or check for patterns. Essential for validation and parsing.

Extracting & Cutting

Substring(), Split(), Trim() - Extract portions of strings, divide into parts, or remove extra whitespace. Key for processing formatted data.

Case Conversion

ToUpper(), ToLower(), ToTitleCase() - Change letter casing for normalization, display, or comparison purposes.

Combining & Replacing

Concat(), Join(), Replace() - Combine multiple strings or replace text. Use StringBuilder for heavy concatenation in loops.

```
// String Manipulation Examples
using System;
using System.Text;

class StringDemo
{
    static void Main()
    {
        string name = "  John Doe  ";
        string email = "john.doe@example.com";

        // Basic operations
        Console.WriteLine(name.Trim());            // Remove whitespace
        Console.WriteLine(name.ToUpper());         // Convert to uppercase
        Console.WriteLine(email.ToLower());        // Convert to lowercase

        // Searching and checking
        bool hasJohn = name.Contains("John");      // Check if contains
        bool startsWithJ = name.Trim().StartsWith("J");  // Check start
        int atPosition = email.IndexOf("@");       // Find character position

        // Extracting
        string firstName = name.Trim().Substring(0, 4);  // Get first 4 chars
        string domain = email.Substring(atPosition + 1); // After @ symbol

        // Splitting
        string[] parts = name.Trim().Split(' ');
        Console.WriteLine($"First: {parts[0]}, Last: {parts[1]}");

        // Replacing
        string masked = email.Replace("john", "****");
        Console.WriteLine(masked);

        // String formatting
        int age = 25;
        double salary = 50000.50;
        string info = string.Format("Name: {0}, Age: {1}, Salary: {2:C}",
                                    name.Trim(), age, salary);
        Console.WriteLine(info);

        // String interpolation (modern approach)
        string message = $"{name.Trim()} is {age} years old";

        // StringBuilder for efficient concatenation
        StringBuilder builder = new StringBuilder();
        for (int i = 1; i <= 100; i++)
        {
            builder.Append($"Number {i}\n");
        }
        // Much more efficient than string concatenation in loop!
    }
}
```

String vs StringBuilder

String: Immutable, creates new object on each modification. Fine for occasional operations.

StringBuilder: Mutable, modifies internal buffer. Use for loops with many concatenations (10+ operations).

```
// Inefficient
string result = "";
for (int i = 0; i < 1000; i++)
    result += i; // Creates 1000 strings!

// Efficient
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++)
    sb.Append(i); // Modifies one buffer
```

String Comparison

Use proper methods for comparing strings:

- `==` operator - Exact match
- `Equals()` - Case-sensitive equality
- `Equals(StringComparison.OrdinalIgnoreCase)` - Case-insensitive
- `CompareTo()` - Alphabetical ordering

```
string a = "hello";
string b = "HELLO";

// Different results
a == b                    // false
a.Equals(b, StringComparison
  .OrdinalIgnoreCase)     // true
```

String interpolation (using `$` prefix) is the modern, readable way to embed expressions in strings. Instead of writing `string.Format("Hello {0}", name)`, you can write `$"Hello {name}"`. This makes code much clearer, especially with multiple variables or expressions. You can even include method calls and calculations directly: `$"Total: {price * quantity:C}"` (the `:C` formats as currency).

Performance Tip: If you need to concatenate strings in a loop more than 5-10 times, use StringBuilder. Each string concatenation creates a new string object and copies all previous characters, making loops with hundreds or thousands of iterations extremely slow with regular strings.

File Handling and Input/Output Operations

File handling is essential for creating applications that persist data beyond program execution. Whether you're saving user preferences, logging errors, importing data, or generating reports, you need to read from and write to files. The .NET Framework provides comprehensive support for file operations through the System.IO namespace, offering both simple methods for common tasks and powerful stream-based APIs for advanced scenarios.

There are two main approaches to file I/O in C#: convenience methods and stream-based operations. Convenience methods like File.WriteAllText() and File.ReadAllText() are perfect for simple scenarios—they handle everything in one line of code. Stream-based operations using StreamReader and StreamWriter give you more control, allowing you to read or write files line by line or in chunks, which is essential for large files that won't fit entirely in memory.

01

Check File Exists

Use File.Exists() to verify file presence before operations to avoid exceptions

02

Choose Method

Simple files: use File class methods. Large files or line-by-line: use Streams

03

Handle Errors

Wrap file operations in try-catch to handle missing files, permission issues

04

Close Resources

Always close streams with using statement to prevent resource leaks

```
// File Handling Examples
using System;
using System.IO;

class FileDemo
{
    static void Main()
    {
        // Example 1: Simple read/write with File class
        string filePath = "student_data.txt";

        // Write to file (creates or overwrites)
        File.WriteAllText(filePath, "John Doe, 85\nJane Smith, 92\n");

        // Append to file
        File.AppendAllText(filePath, "Bob Johnson, 78\n");

        // Read entire file
        string content = File.ReadAllText(filePath);
        Console.WriteLine("File contents:\n" + content);

        // Read all lines into array
        string[] lines = File.ReadAllLines(filePath);
        foreach (string line in lines)
        {
            string[] parts = line.Split(',');
            Console.WriteLine($"Student: {parts[0]}, Grade: {parts[1]}");
        }

        // Example 2: StreamWriter for efficient writing
        using (StreamWriter writer = new StreamWriter("output.txt"))
        {
            writer.WriteLine("Header Line");
            for (int i = 1; i <= 100; i++)
            {
                writer.WriteLine($"Line {i}: Some data here");
            }
        } // Automatically closes the stream

        // Example 3: StreamReader for line-by-line reading
        if (File.Exists("output.txt"))
        {
            using (StreamReader reader = new StreamReader("output.txt"))
            {
                string line;
                int lineNumber = 0;
                while ((line = reader.ReadLine()) != null)
                {
                    lineNumber++;
                    Console.WriteLine($"{lineNumber}: {line}");

                    // Can process huge files without loading all into memory
                    if (lineNumber >= 5) break; // Just show first 5 lines
                }
            }
        }

        // Example 4: Error handling with files
        try
        {
            string data = File.ReadAllText("nonexistent.txt");
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("File not found!");
        }
        catch (UnauthorizedAccessException)
        {
            Console.WriteLine("No permission to access file!");
        }
        catch (IOException ex)
        {
            Console.WriteLine($"I/O error: {ex.Message}");
        }
    }
}
```

File Class Methods

Simple, one-line operations:

- `File.WriteAllText()` - Write string to file
- `File.ReadAllText()` - Read entire file as string
- `File.AppendAllText()` - Add to end of file
- `File.ReadAllLines()` - Read file as string array
- `File.Exists()` - Check if file exists
- `File.Delete()` - Remove a file
- `File.Copy()` - Copy file to new location

When to Use Streams

- Large files (100+ MB) - avoid loading all into memory
- Line-by-line processing (log files, CSV data)
- Real-time data writing (logging during execution)
- Binary files (images, audio, custom formats)
- Network streams or compression

Using Statement: Automatically disposes resources even if exception occurs. Always use with streams!

The `using` statement is crucial when working with streams and file handles. Files are system resources that need to be properly closed after use—forgetting to close them can lead to locked files, resource leaks, and eventually program crashes. The using statement ensures that the file handle is closed and resources are released, even if an exception occurs during processing. It's like borrowing a book from the library—the using statement guarantees you return it, even if you get interrupted while reading.

Path Handling: Use `Path.Combine()` to build file paths instead of string concatenation. This handles different path separators (/ vs \) automatically across operating systems: `Path.Combine("C:", "Data", "file.txt")` works correctly on Windows, Linux, and Mac.

Collections: Managing Groups of Data

Collections are data structures that hold multiple related items, providing a more flexible and powerful alternative to arrays. While arrays have a fixed size that must be determined at creation, collections like List, Dictionary, Queue, and Stack can grow or shrink dynamically as you add or remove items. Think of an array as a parking lot with numbered spaces—you must know how many spaces you need upfront. A collection is more like a flexible parking system that expands as more cars arrive.

The .NET Framework provides both generic and non-generic collections. Generic collections (in System.Collections.Generic) are type-safe—they only hold one specific type of object, catching type errors at compile time. Non-generic collections (in System.Collections) can hold any type of object, but you lose type safety and performance. Always prefer generic collections in modern C# development unless you have a specific reason to use non-generic ones.

List<T>

Dynamic array that can grow/shrink. Best for ordered collections where you need index access, sorting, and flexible size. Most commonly used collection.

`List<string> names = new List<string>();`

Dictionary<TKey, TValue>

Stores key-value pairs for fast lookup by key. Perfect for mappings like student ID to student object, or word to definition.

`Dictionary<string, int> ages = new Dictionary<string, int>();`

Queue<T>

First-In-First-Out (FIFO) collection. Items added to end, removed from front. Use for task scheduling, message processing.

`Queue<string> tasks = new Queue<string>();`

Stack<T>

Last-In-First-Out (LIFO) collection. Items added and removed from top. Use for undo functionality, expression evaluation.

`Stack<int> history = new Stack<int>();`

```
// Collection Examples
using System;
using System.Collections.Generic;

class CollectionDemo
{
    static void Main()
    {
        // List - Most common collection
        List students = new List();
        students.Add("Alice");
        students.Add("Bob");
        students.Add("Charlie");
        students.Insert(1, "David");  // Insert at specific position
        students.Remove("Bob");       // Remove by value
        students.RemoveAt(0);         // Remove by index

        Console.WriteLine($"Count: {students.Count}");
        foreach (string student in students)
        {
            Console.WriteLine(student);
        }

        // Dictionary - Key-value pairs
        Dictionary grades = new Dictionary();
        grades["Alice"] = 85;
        grades["Bob"] = 92;
        grades["Charlie"] = 78;

        // Access by key
        Console.WriteLine($"Alice's grade: {grades["Alice"]}");

        // Check if key exists before accessing
        if (grades.ContainsKey("David"))
        {
            Console.WriteLine(grades["David"]);
        }
        else
        {
            Console.WriteLine("David not found");
        }

        // Iterate through dictionary
        foreach (KeyValuePair entry in grades)
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }

        // Queue - FIFO (First In, First Out)
        Queue printQueue = new Queue();
        printQueue.Enqueue("Document1.pdf");
        printQueue.Enqueue("Document2.pdf");
        printQueue.Enqueue("Document3.pdf");

        while (printQueue.Count > 0)
        {
            string doc = printQueue.Dequeue();  // Remove from front
            Console.WriteLine($"Printing: {doc}");
        }

        // Stack - LIFO (Last In, First Out)
        Stack browserHistory = new Stack();
        browserHistory.Push("google.com");
        browserHistory.Push("github.com");
        browserHistory.Push("stackoverflow.com");

        // Go back through history
        Console.WriteLine($"Current: {browserHistory.Peek()}");  // Look without removing
        browserHistory.Pop();  // Remove from top
        Console.WriteLine($"After back: {browserHistory.Peek()}");
    }
}
```

Choosing the Right Collection

- **Need order + index access?** → List<T>
- **Need fast lookup by key?** → Dictionary<TKey, TValue>
- **Need unique items only?** → HashSet<T>
- **Need FIFO processing?** → Queue<T>
- **Need LIFO processing?** → Stack<T>
- **Need sorted items?** → SortedList or SortedSet

Common List Operations

```
List<int> numbers = new List<int>();

// Adding
numbers.Add(10);
numbers.AddRange(new int[] {20, 30});

// Removing
numbers.Remove(10);     // By value
numbers.RemoveAt(0);    // By index
numbers.Clear();        // All

// Searching
bool has20 = numbers.Contains(20);
int index = numbers.IndexOf(30);

// Sorting
numbers.Sort();
numbers.Reverse();
```

Collections implement IEnumerable, which means they can be used with foreach loops and LINQ queries. This makes it easy to iterate through items, filter data, transform collections, and perform complex queries using consistent syntax across all collection types. Modern C# development heavily relies on collections and LINQ to process data efficiently and expressively.

Performance Consideration: Dictionary lookups are O(1) constant time, while List searching is O(n) linear time. If you frequently search for items by a unique identifier, use Dictionary instead of searching through a List repeatedly—the performance difference becomes significant with thousands of items.

Practical Application: Student Management System

Now that we've learned about C#.NET language features, file handling, collections, and error handling, let's combine these concepts into a practical project. We'll build a simple student management system that demonstrates real-world application of everything we've covered. This console application will store student records in memory using collections, save data to files for persistence, handle user input with error checking, and provide a menu-driven interface.

This practical exercise reinforces how different C# concepts work together in a cohesive application. You'll see how List<T> provides dynamic storage, file I/O enables data persistence across program runs, try-catch blocks create a robust user experience, and string manipulation processes user input safely. Real applications layer these fundamental concepts to create increasingly sophisticated functionality.

```
// Student Management System - Complete Example
using System;
using System.Collections.Generic;
using System.IO;

namespace StudentManagement
{
  // Student class with properties
  class Student
  {
    public string StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }

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

    public static Student FromString(string data)
    {
      string[] parts = data.Split(',');
      return new Student
      {
        StudentId = parts[0],
        Name = parts[1],
        Age = int.Parse(parts[2]),
        GPA = double.Parse(parts[3])
      };
    }
  }

  class Program
  {
    static List students = new List();
    static string dataFile = "students.txt";

    static void Main(string[] args)
    {
      LoadStudents();

      while (true)
      {
        Console.WriteLine("\n=== Student Management System ===");
        Console.WriteLine("1. Add Student");
        Console.WriteLine("2. View All Students");
        Console.WriteLine("3. Search Student");
        Console.WriteLine("4. Delete Student");
        Console.WriteLine("5. Save and Exit");
        Console.Write("Choose option: ");

        string choice = Console.ReadLine();

        switch (choice)
        {
          case "1": AddStudent(); break;
          case "2": ViewAllStudents(); break;
          case "3": SearchStudent(); break;
          case "4": DeleteStudent(); break;
          case "5": SaveStudents(); return;
          default: Console.WriteLine("Invalid option!"); break;
        }
      }
    }

    static void AddStudent()
    {
      try
      {
        Console.Write("Enter Student ID: ");
        string id = Console.ReadLine();

        Console.Write("Enter Name: ");
        string name = Console.ReadLine();

        Console.Write("Enter Age: ");
        int age = int.Parse(Console.ReadLine());

        Console.Write("Enter GPA: ");
        double gpa = double.Parse(Console.ReadLine());

        Student student = new Student
        {
          StudentId = id,
          Name = name,
          Age = age,
          GPA = gpa
        };

        students.Add(student);
        Console.WriteLine("Student added successfully!");
      }
      catch (FormatException)
      {
        Console.WriteLine("Invalid input format!");
      }
    }

    static void ViewAllStudents()
    {
      if (students.Count == 0)
      {
        Console.WriteLine("No students found.");
        return;
      }

      Console.WriteLine("\n{0,-10} {1,-20} {2,-5} {3,-5}",
                "ID", "Name", "Age", "GPA");
      Console.WriteLine(new string('-', 50));

      foreach (Student s in students)
      {
        Console.WriteLine("{0,-10} {1,-20} {2,-5} {3,-5:F2}",
                  s.StudentId, s.Name, s.Age, s.GPA);
      }
    }

    static void SearchStudent()
    {
      Console.Write("Enter Student ID to search: ");
      string id = Console.ReadLine();

      Student found = students.Find(s => s.StudentId == id);

      if (found != null)
      {
        Console.WriteLine($"Found: {found.Name}, Age: {found.Age}, GPA: {found.GPA}");
      }
      else
      {
        Console.WriteLine("Student not found.");
      }
    }

    static void DeleteStudent()
    {
      Console.Write("Enter Student ID to delete: ");
      string id = Console.ReadLine();

      int removed = students.RemoveAll(s => s.StudentId == id);

      if (removed > 0)
        Console.WriteLine("Student deleted successfully!");
      else
        Console.WriteLine("Student not found.");
    }

    static void LoadStudents()
    {
      if (File.Exists(dataFile))
      {
        try
        {
          string[] lines = File.ReadAllLines(dataFile);
          foreach (string line in lines)
          {
            if (!string.IsNullOrWhiteSpace(line))
            {
              students.Add(Student.FromString(line));
            }
          }
          Console.WriteLine($"Loaded {students.Count} students from file.");
        }
        catch (Exception ex)
        {
          Console.WriteLine($"Error loading data: {ex.Message}");
        }
      }
    }

    static void SaveStudents()
    {
      try
      {
        using (StreamWriter writer = new StreamWriter(dataFile))
        {
          foreach (Student s in students)
          {
            writer.WriteLine(s.ToString());
          }
        }
        Console.WriteLine("Data saved successfully!");
      }
      catch (Exception ex)
      {
        Console.WriteLine($"Error saving data: {ex.Message}");
      }
    }
  }
}
```

1. **Data Model**
   Student class defines structure with properties and methods for serialization
2. **In-Memory Storage**
   List<Student> holds active data during program execution
3. **File Persistence**
   Load data on startup, save on exit using CSV format
4. **User Interface**
   Menu-driven console interface with input validation

This example demonstrates professional programming practices: separating data (Student class) from logic (Program class), using collections for flexible storage, implementing persistence with file I/O, handling errors gracefully, and providing clear user feedback. As you build more complex applications, you'll apply these same patterns while adding layers like databases, web interfaces, or graphical UIs.

Key Takeaways and Next Steps

Congratulations on completing this comprehensive exploration of C#.NET language features and the .NET framework! Let's recap the key concepts you've learned and how they fit together to create robust applications. You now understand how to create and organize .NET projects using namespaces and assemblies, work with C#'s type system including value and reference types, manipulate strings efficiently, handle files and data persistence, manage collections of data dynamically, and implement proper error handling and debugging practices.

Project Organization

Use namespaces to organize code logically. Understand assemblies as deployment units. Create modular, maintainable project structures.

Type System Mastery

Choose appropriate data types for variables. Understand value vs reference semantics. Use type conversion safely with proper error handling.

Data Management

Leverage collections instead of arrays for flexibility. Use Dictionary for fast lookups. Apply appropriate collection types for specific scenarios.

Professional Practices

Implement robust error handling with try-catch. Use debugging tools effectively. Write clean, readable code with proper conventions.

Common Pitfalls to Avoid

- Forgetting to close file streams (always use `using`)
- Using string concatenation in loops (use StringBuilder)
- Catching generic exceptions and swallowing errors
- Not checking for null references before use
- Using non-generic collections in new code
- Hardcoding file paths instead of using Path.Combine()

Best Practices Checklist

- ✓ Follow naming conventions (PascalCase for classes, camelCase for variables)
- ✓ Always use generic collections (List<T>, not ArrayList)
- ✓ Handle exceptions at appropriate levels
- ✓ Use using statements with IDisposable resources
- ✓ Validate user input before processing
- ✓ Comment complex logic, not obvious code

The foundation you've built with C#.NET language features prepares you for the next phase of your learning journey: ADO.NET and database integration. You'll soon learn how to connect your C# applications to databases, execute SQL queries, manage data connections efficiently, and build data-driven applications. All the concepts you've learned—collections for storing query results, error handling for database exceptions, file I/O for import/export operations—will directly apply as you work with databases.

Unit 2: C#.NET Basics

Language features, collections, file I/O - Current unit completed

1

Unit 3: ADO.NET

Database connectivity, SQL integration, data management - Coming next

2

Future Units

Web development, APIs, advanced frameworks - Building on fundamentals

3

Practice Recommendation: To solidify these concepts, build small projects that combine multiple topics. Create a library management system (classes + collections + file I/O), a grade calculator (strings + math + error handling), or a simple contact manager. Real practice makes these concepts second nature.

Keep experimenting, keep coding, and don't hesitate to revisit these notes as you build more complex applications. The journey from understanding syntax to writing professional software is iterative—each project you build reinforces fundamentals while introducing new challenges. Your next step into database programming will open exciting possibilities for creating data-driven applications that store and retrieve information persistently, handle concurrent users, and scale to real-world demands.