

- # .NET
 - > ### **Overview of .NET Framework** ...
 - > ### **C# Program Compilation and Execution** ...
 - > ### **CLR (Common Language Runtime)** ...
 - > ### **CLS (Common Language Specification) and CTS (Common Type System)** ...
 - > ### **.Net Assemblies (.exe or .dll)** ...
 - > ### **IL (Intermediate Language)** ...
 - > ### **Visual Studio Overview** ...
 - > ### **Introduction to OOP & C# Classes** ...

Enterprise Technologies

1. Java - 3 + 3 + 1.5
2. MERN - 3
3. .NET - 2

Career ->
.NET

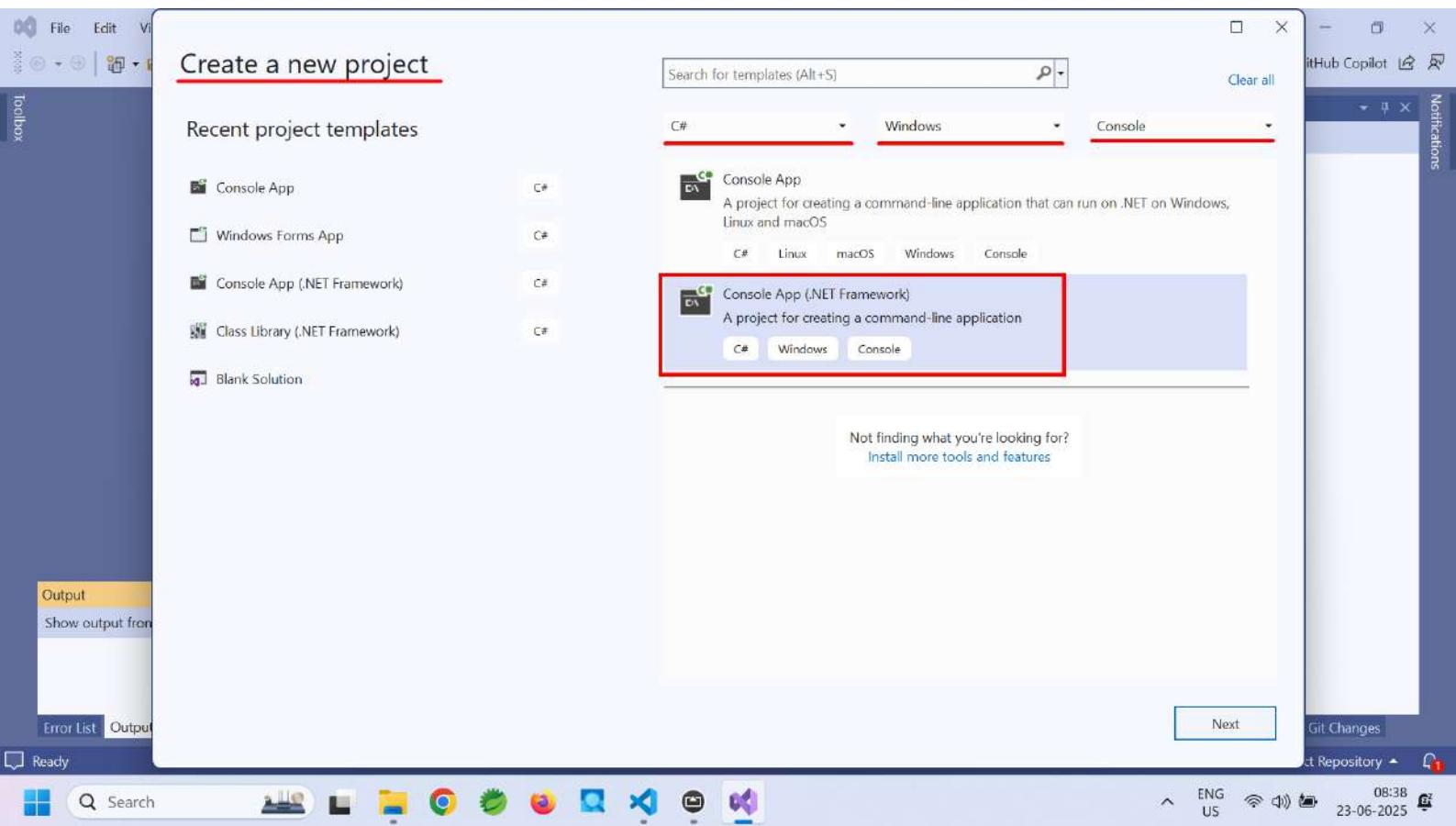
```
class Program {  
    public static void Main(String[] args) {  
        Console.WriteLine("Hello");  
    }  
}
```



C# .NET @ Sunbeam Infotech

Trainer: Nilesh Ghule





Configure your new project

Console App (.NET Framework) C# Windows Console

Project name: HelloWorld

Location: D:\Nilesh\Feb25\DAC\dotnet\day01\

Solution name: HelloWorld

Place solution and project in the same directory

Framework: .NET Framework 4.8

Project will be created in "D:\Nilesh\Feb25\DAC\dotnet\day01\HelloWorld\"

Solution Info -> .sln file

Project Info -> .csproj file

For very simple project
1 soln -> only 1 project Solution

Project

Solution = Group of related projects e.g. Business logic, Win App, Web App.

Back Create

The screenshot shows the 'Create New Project' dialog in Visual Studio. It includes fields for Project name (HelloWorld), Location (D:\Nilesh\Feb25\DAC\dotnet\day01\), Solution name (HelloWorld), and Framework (.NET Framework 4.8). A checkbox for placing the solution and project in the same directory is checked. A note at the bottom states the project will be created in the specified directory. Handwritten annotations include a pink arrow pointing to the 'Framework' dropdown with the text 'Classic .NET framework'. Red text on the right side explains 'Solution' as a group of related projects, listing examples like Business logic, Win App, and Web App. Another red text block on the right side states 'For very simple project 1 soln -> only 1 project Solution'. A diagram shows a single large rectangle labeled 'Project' with a red border, representing a single project within a solution.

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search HelloWorld

Program.cs

```
6
7     namespace HelloWorld
8     {
9         internal class Program
10        {
11             static void Main(string[] args)
12             {
13                 Console.WriteLine("Hello, DAC!");
14             }
15         }
16     }
17 }
18 
```

Entry-point fn

built-in static method

(from System namespace)

Solution

Project

config (xml)

Source code (xml)

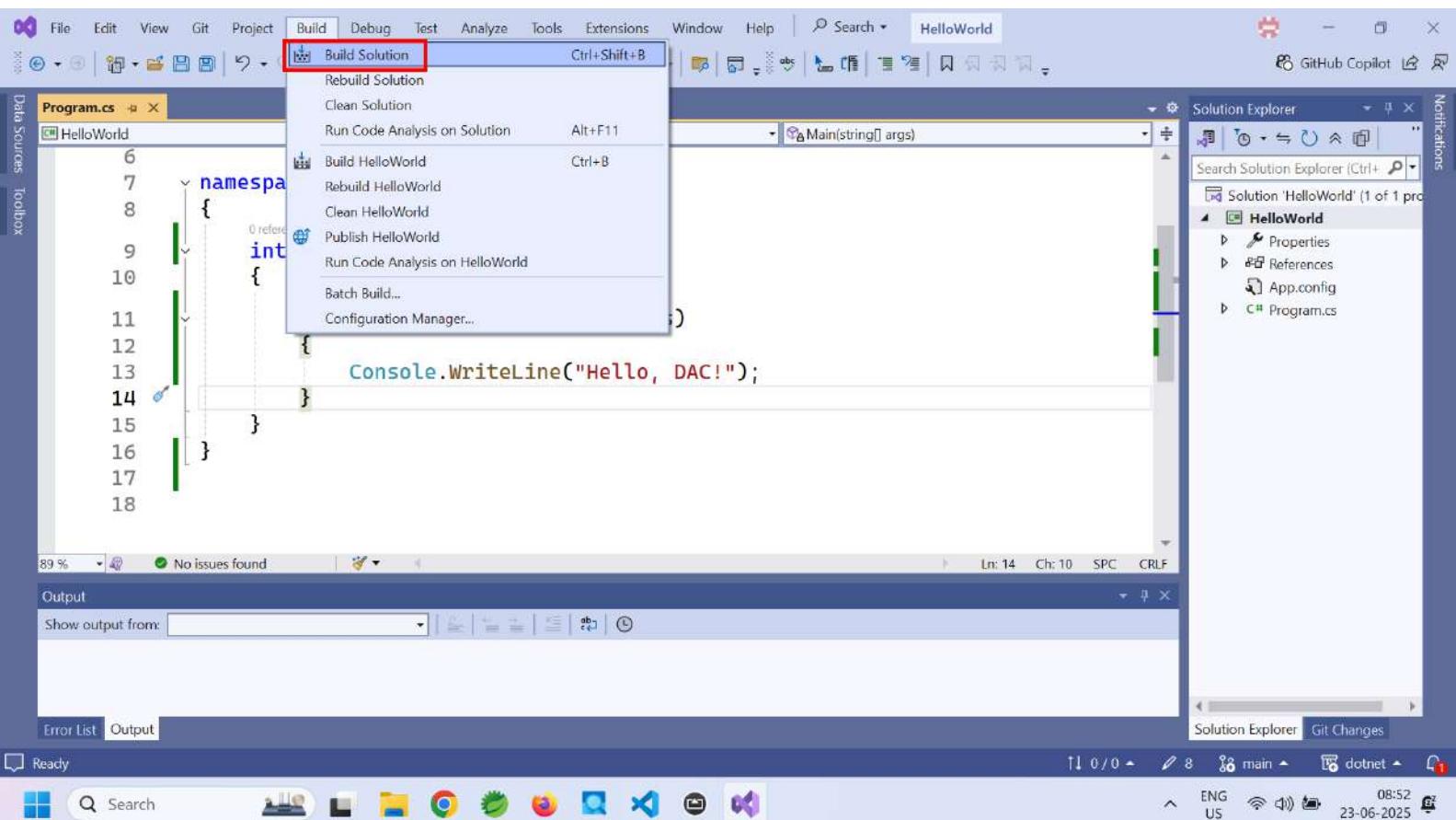
Output

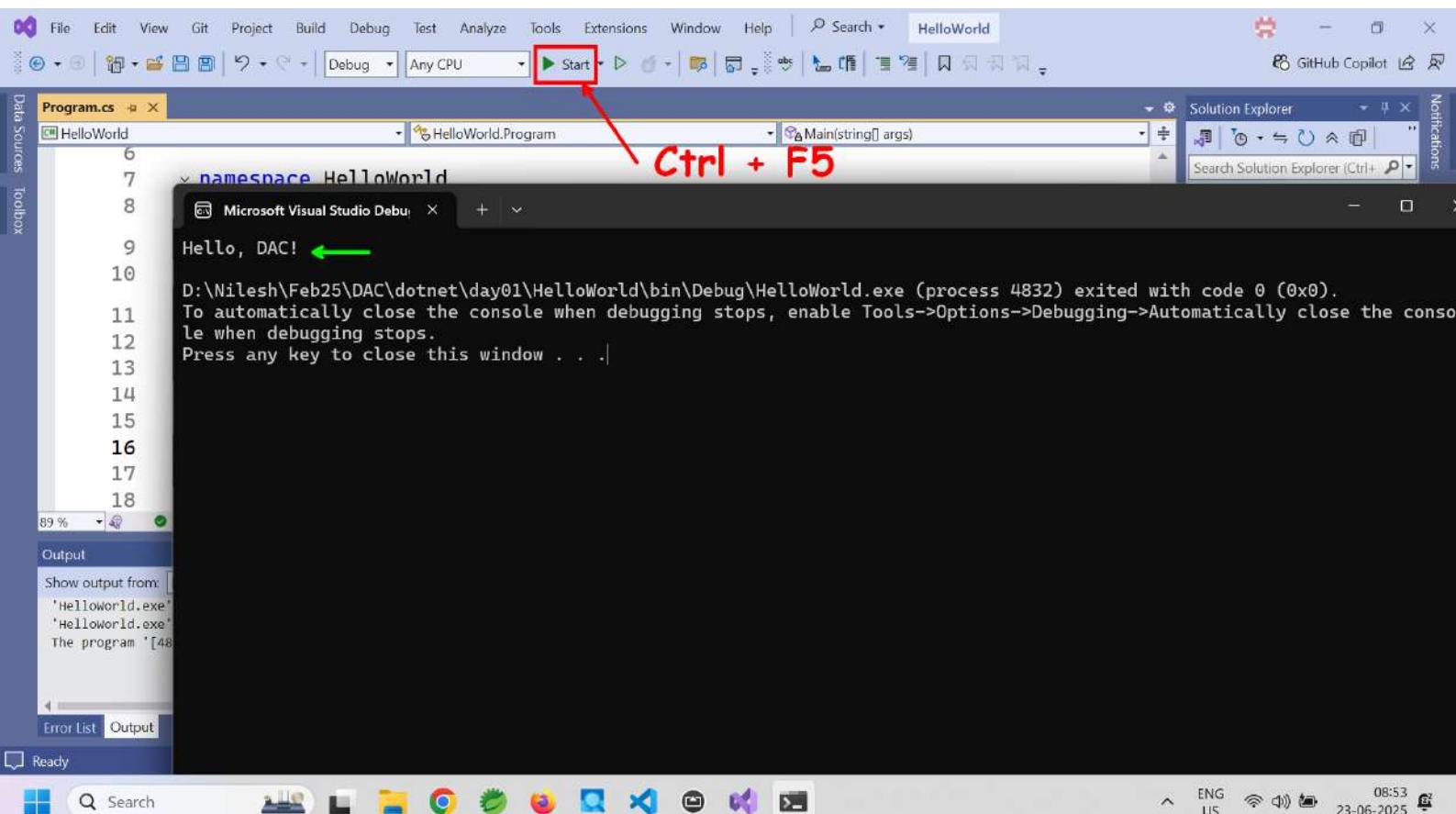
Error List Output

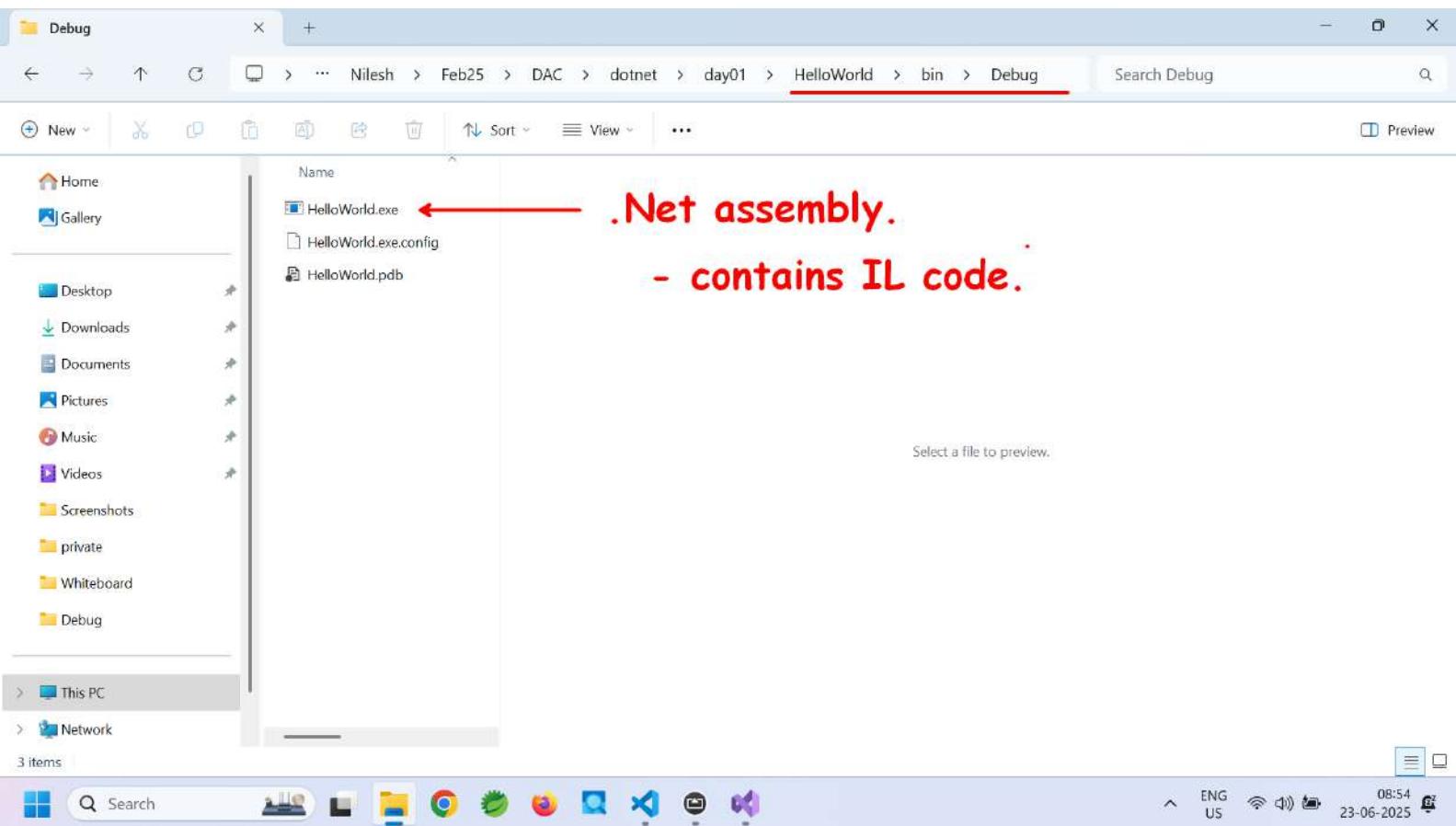
Item(s) Saved

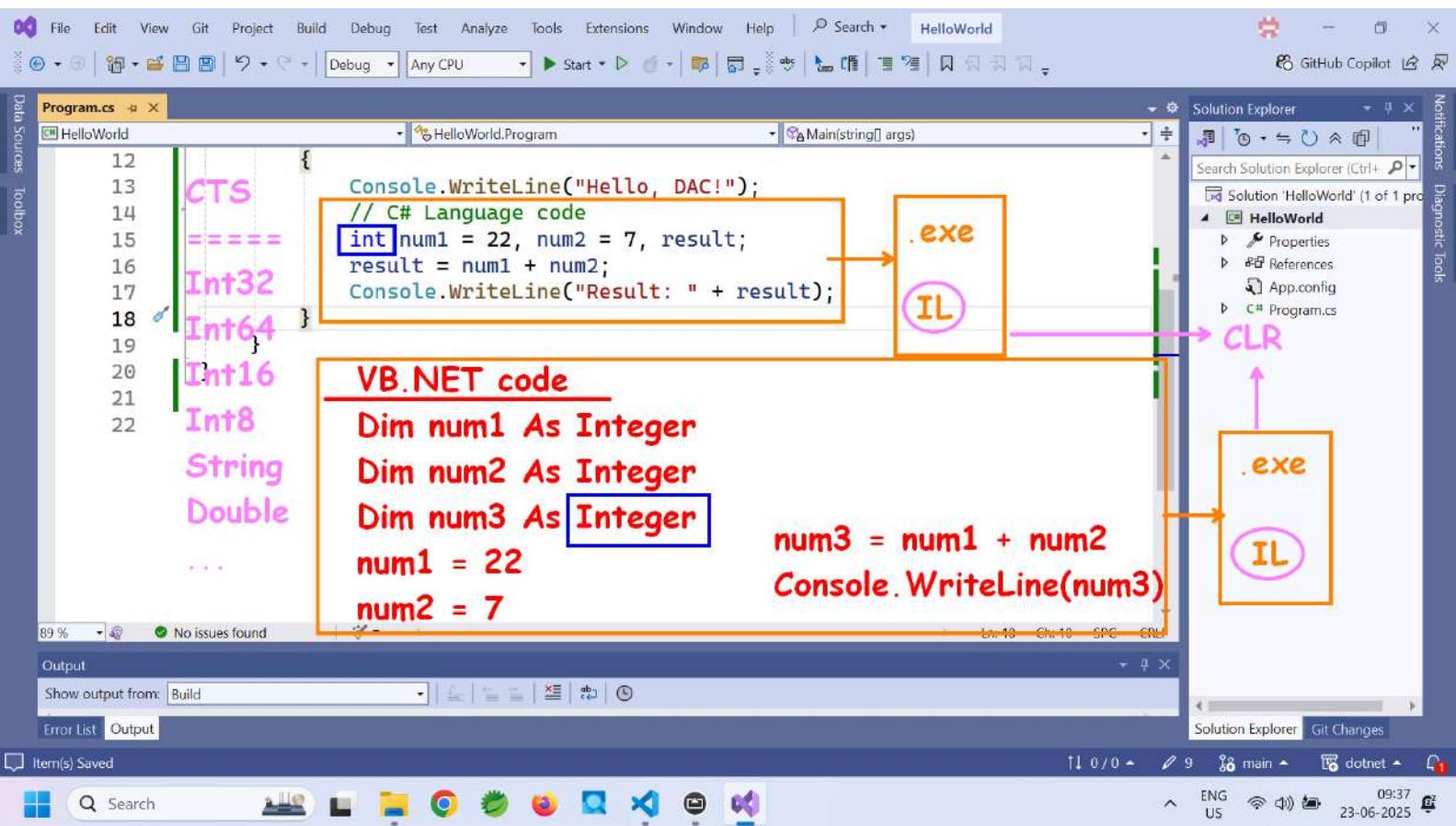
Search

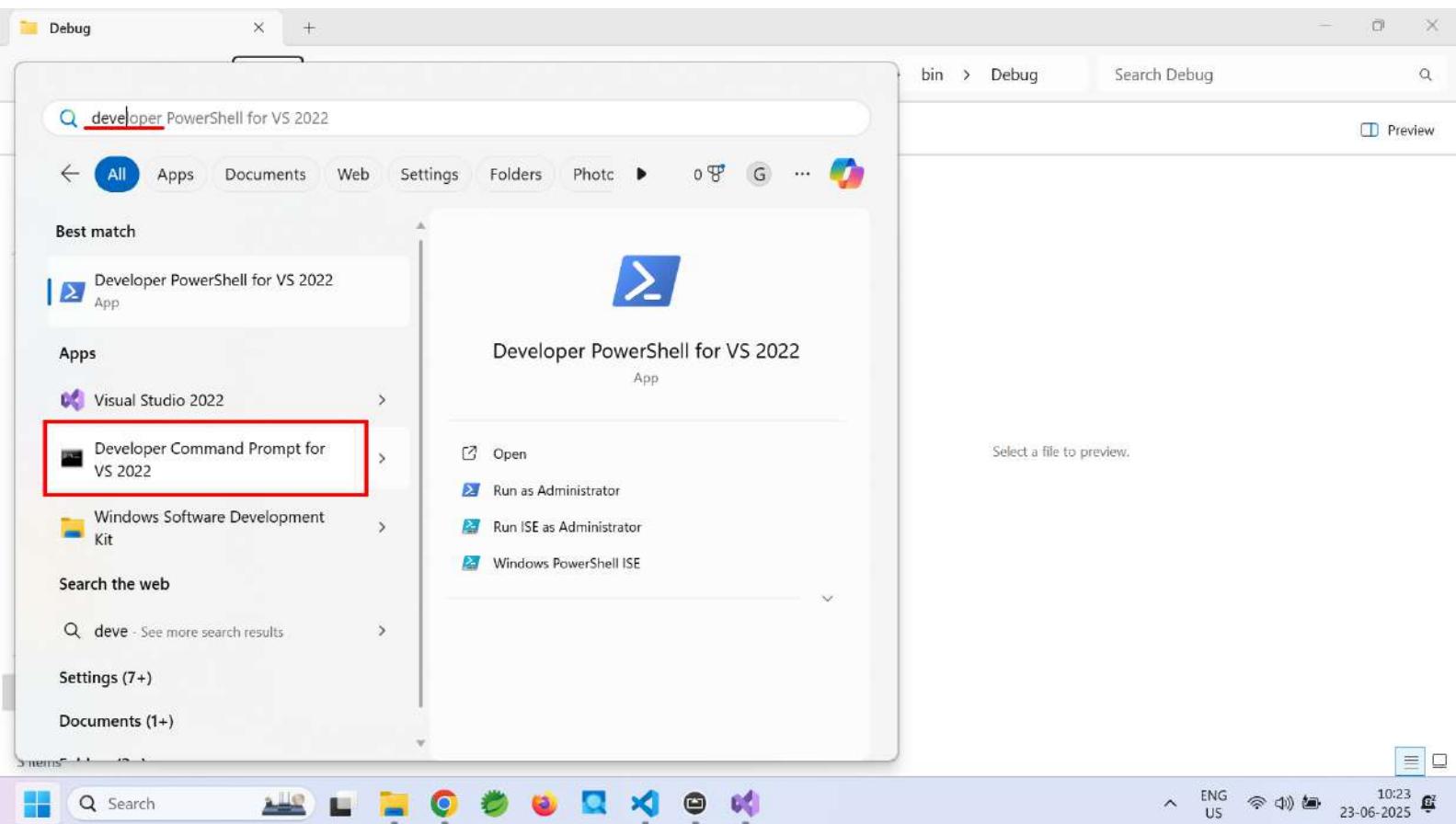
08:48 23-06-2025











Debug

Name: HelloWorld.exe

Date: 23-06

Developer Command Prompt

```
*****  
** Visual Studio 2022 Developer Command Prompt v17.14.3  
** Copyright (c) 2022 Microsoft Corporation  
*****  
C:\Program Files\Microsoft Visual Studio\2022\Community>cd D:\Nilesh\Feb25\DAC\dotnet  
C:\Program Files\Microsoft Visual Studio\2022\Community>D:  
D:\Nilesh\Feb25\DAC\dotnet\day01\HelloWorld\bin\Debug>ildasm.exe HelloWorld.exe  
D:\Nilesh\Feb25\DAC\dotnet\day01\HelloWorld\bin\Debug>
```

path of assembly.

File View Help

HelloWorld.exe - IL DASM

Assembly metadata

class members - IL code.

class (metadata + IL)

.assembly HelloWorld
{
.ver 1:0:0:0}

Debug

Name: HelloWorld.exe Date: 23-06

Developer Command Prompt

```
*****  
** Visual Studio 2022 Developer Command Prompt v17.14.3  
** Copyright (c) 2022 Microsoft Corporation  
*****  
C:\Program Files\Microsoft Visual Studio\2022\Community>cd D:  
C:\Program Files\Microsoft Visual Studio\2022\Community>D:  
D:\Nilesh\Feb25\DAC\dotnet\day01\HelloWorld\bin\Debug>ildasm  
D:\Nilesh\Feb25\DAC\dotnet\day01\HelloWorld\bin\Debug>
```

File View Help

FileWorld.exe - IL DASM

FileWorld

FileWorld.Program

.ctor : void()

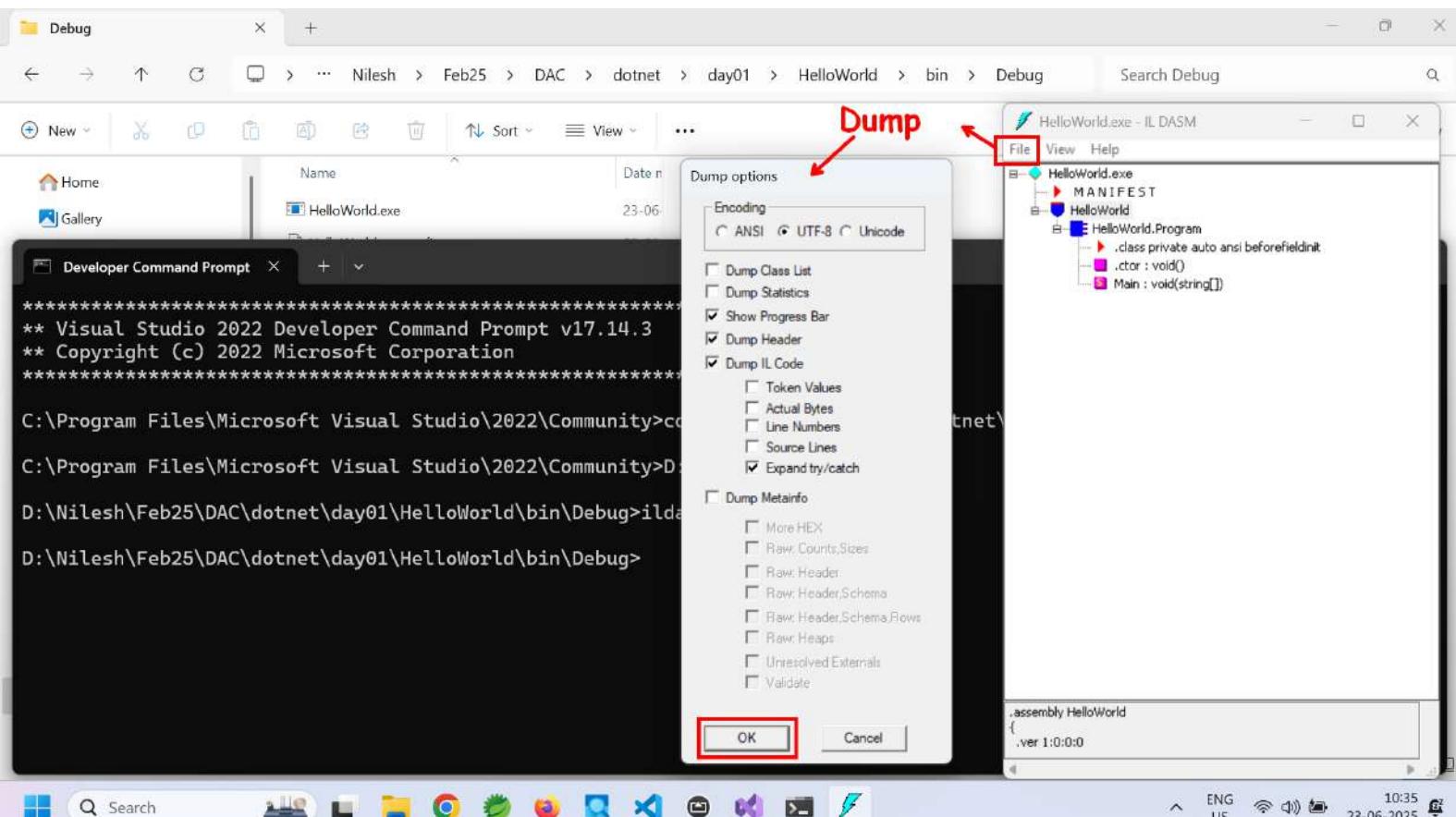
Main : void(string[])

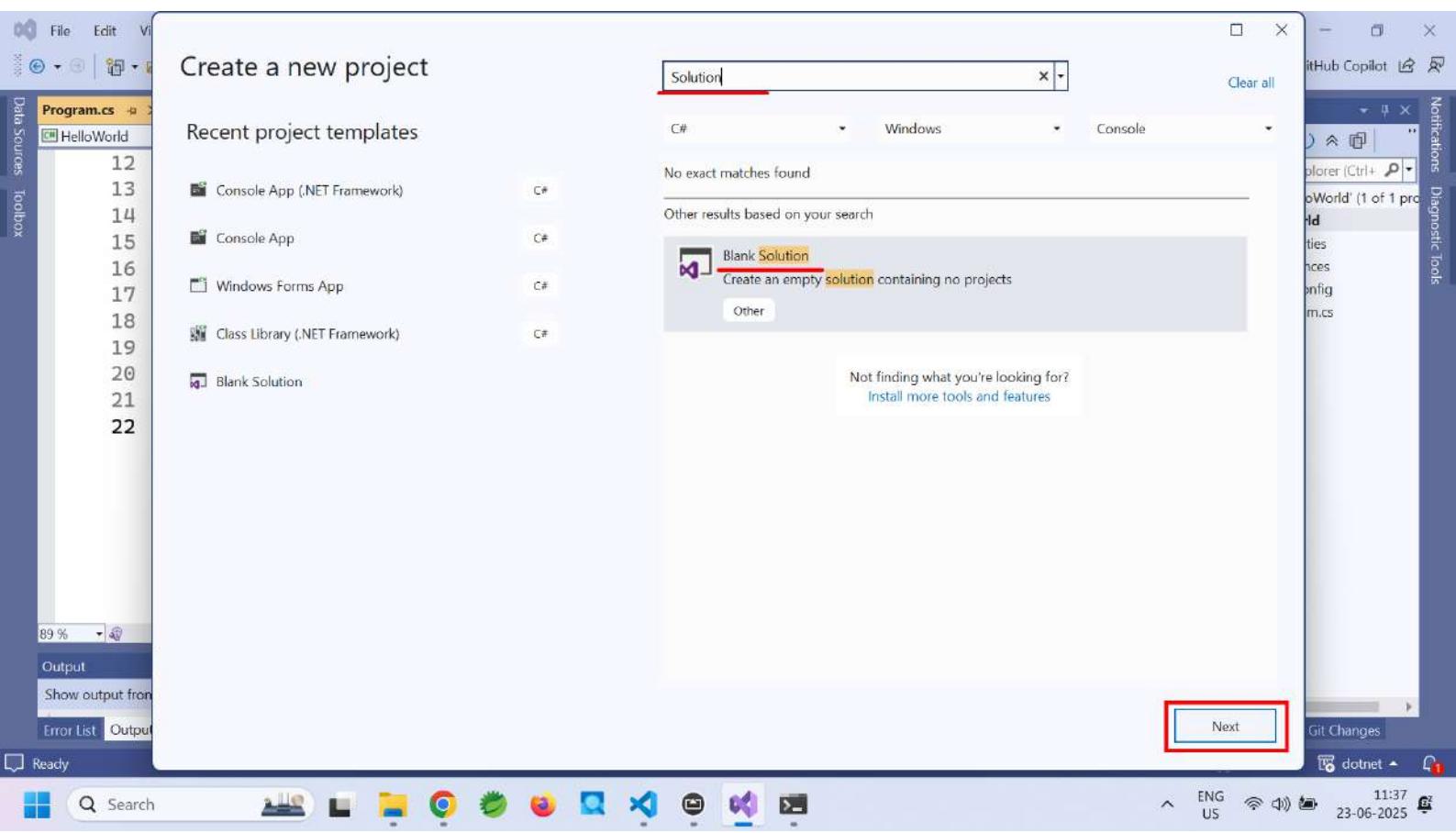
Find Find Next

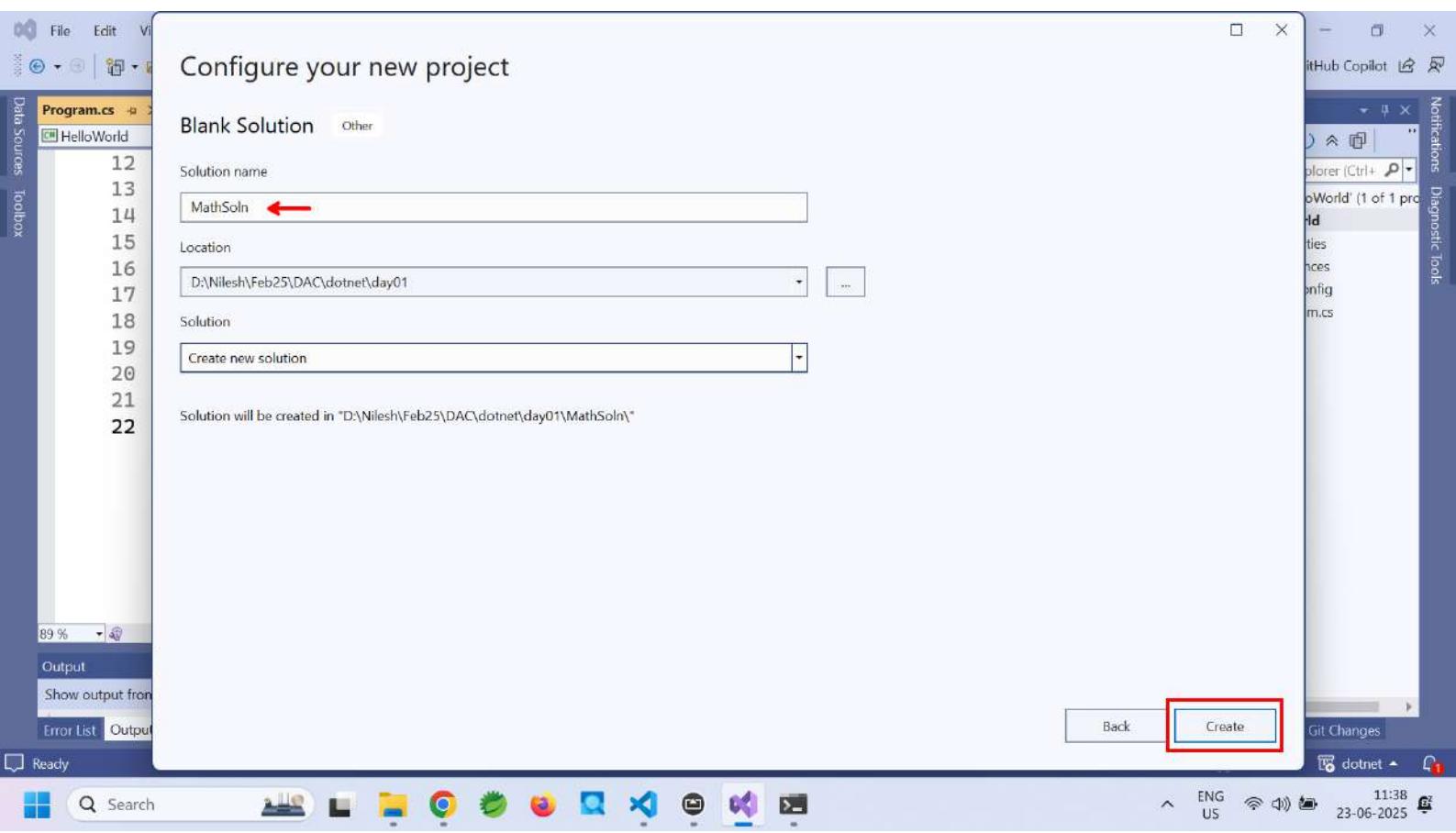
// Code size | 45 (0x2d)
.maxstack 2
.locals init {[0] int32 num1,
[1] int32 num2,
[2] int32 result)
IL_0000: nop
IL_0001: ldstr "Hello, DAC!"
IL_0006: call void [mscorlib]System.Console::WriteLine(string)
IL_000b: nop
IL_000c: ldc.i4.s 22 ← num1 = 22
IL_000e: stloc.0 ← num2 = 7
IL_000f: ldc.i4.7
IL_0010: stloc.1
IL_0011: ldloc.0
IL_0012: idloc.1
IL_0013: add ← num1 + num2
IL_0014: stloc.2
IL_0015: ldstr "Result:"
IL_001a: ldloca.s result
IL_001c: call instance string [mscorlib]System.Int32::ToString()
IL_0021: call string [mscorlib]System.String::Concat(string,
string)
IL_0024: call void [mscorlib]System.String::

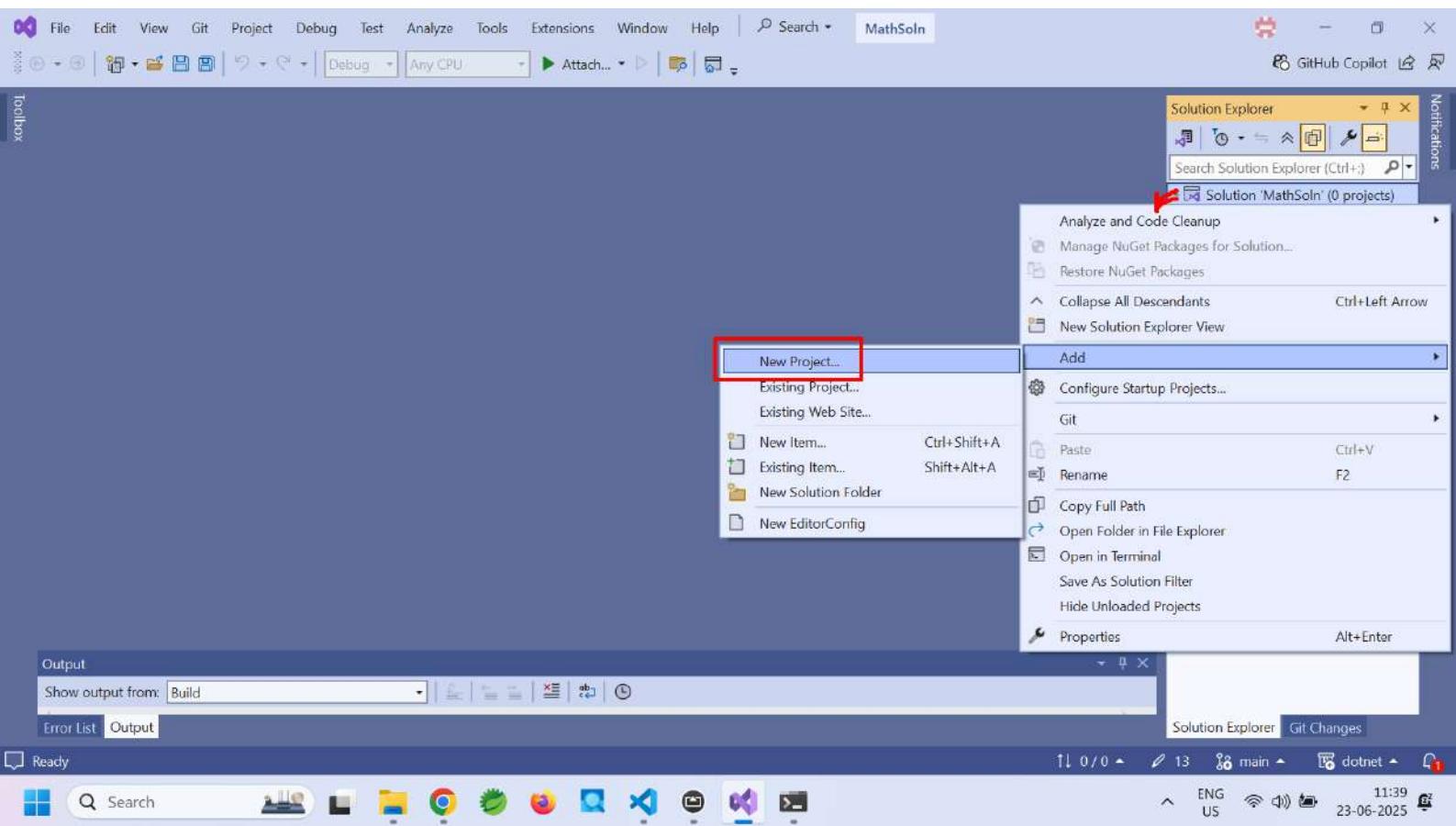
local vars on stack
C# int → IL Int32

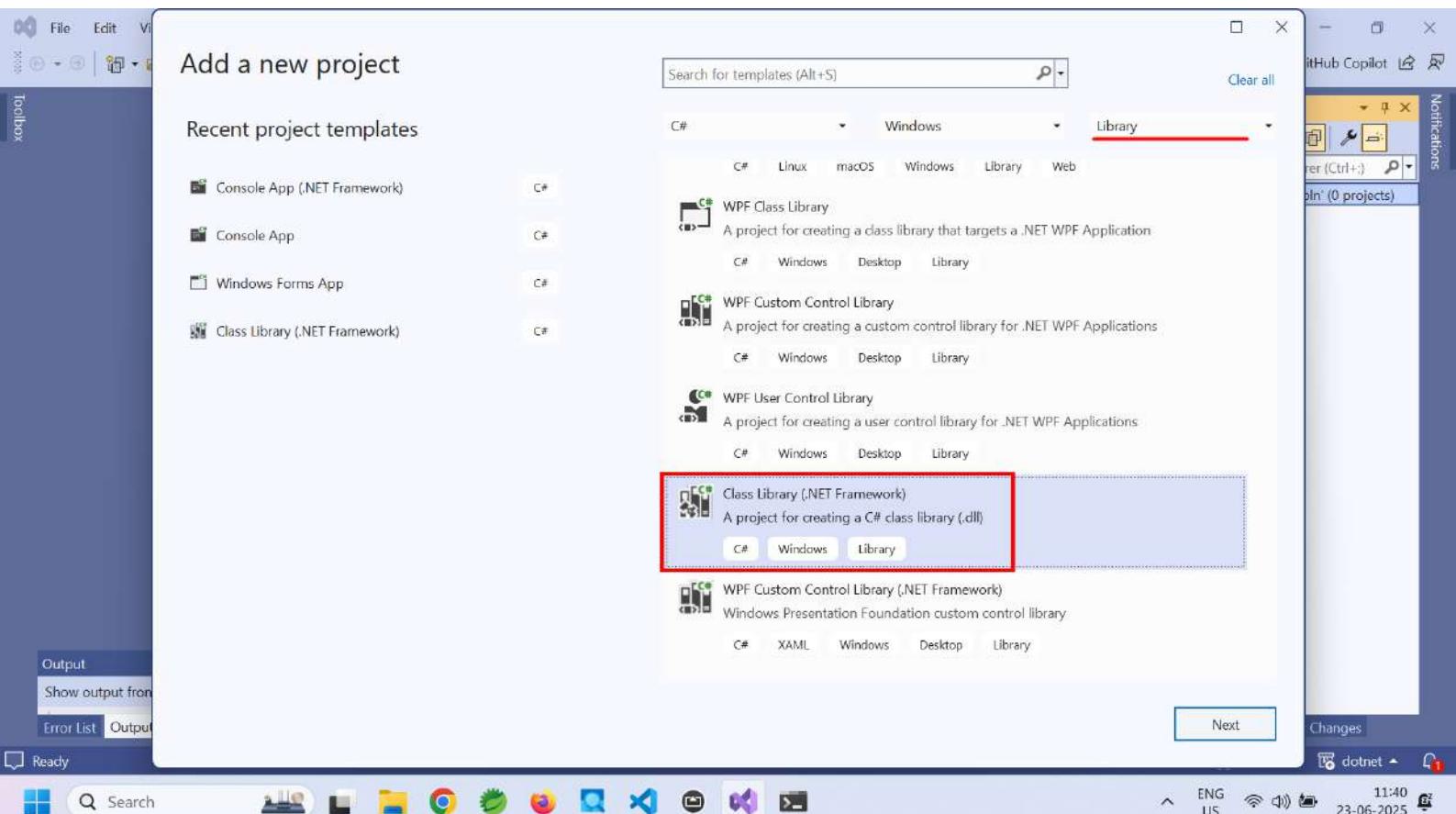
10:31 23-06-2025

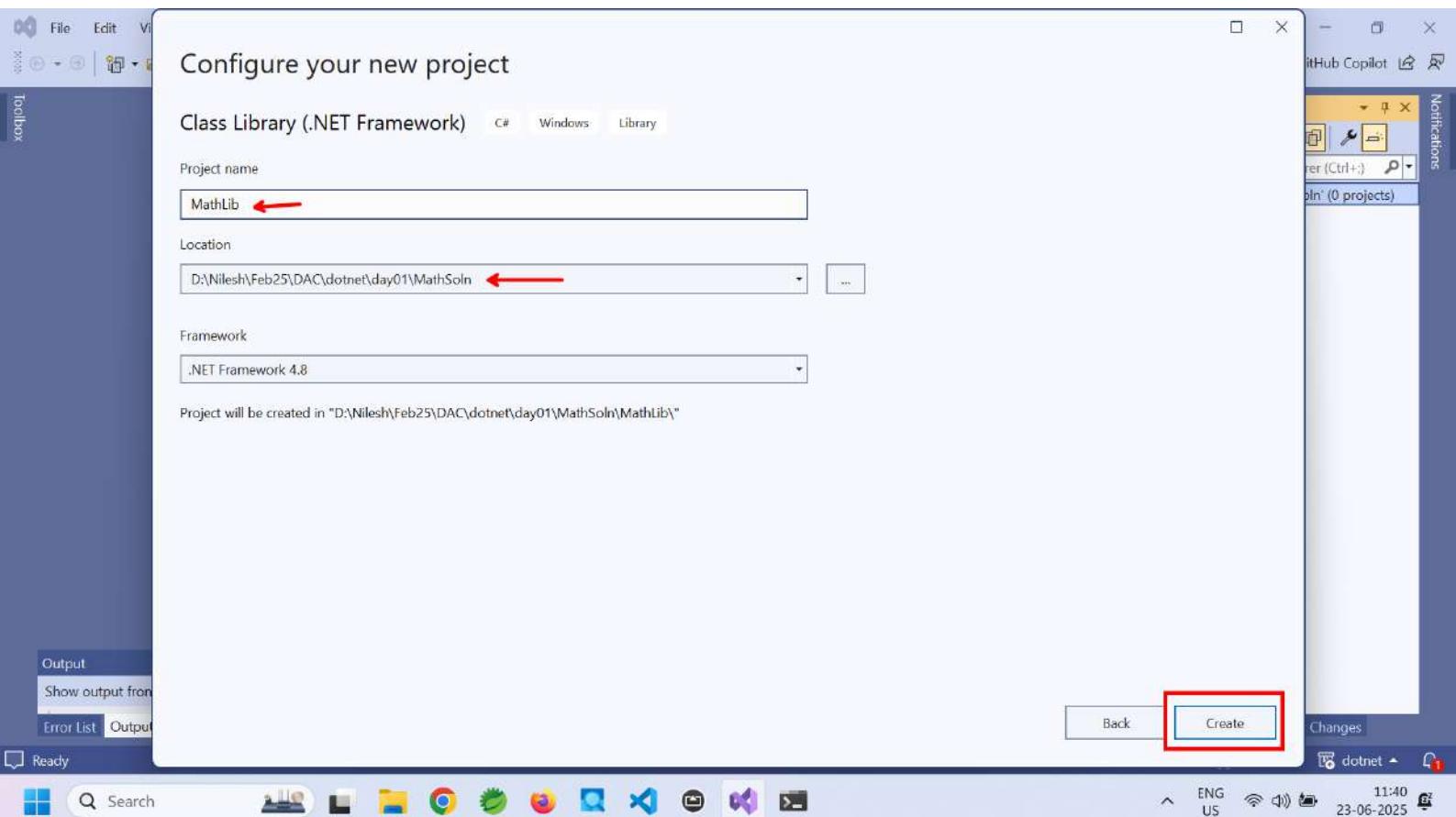










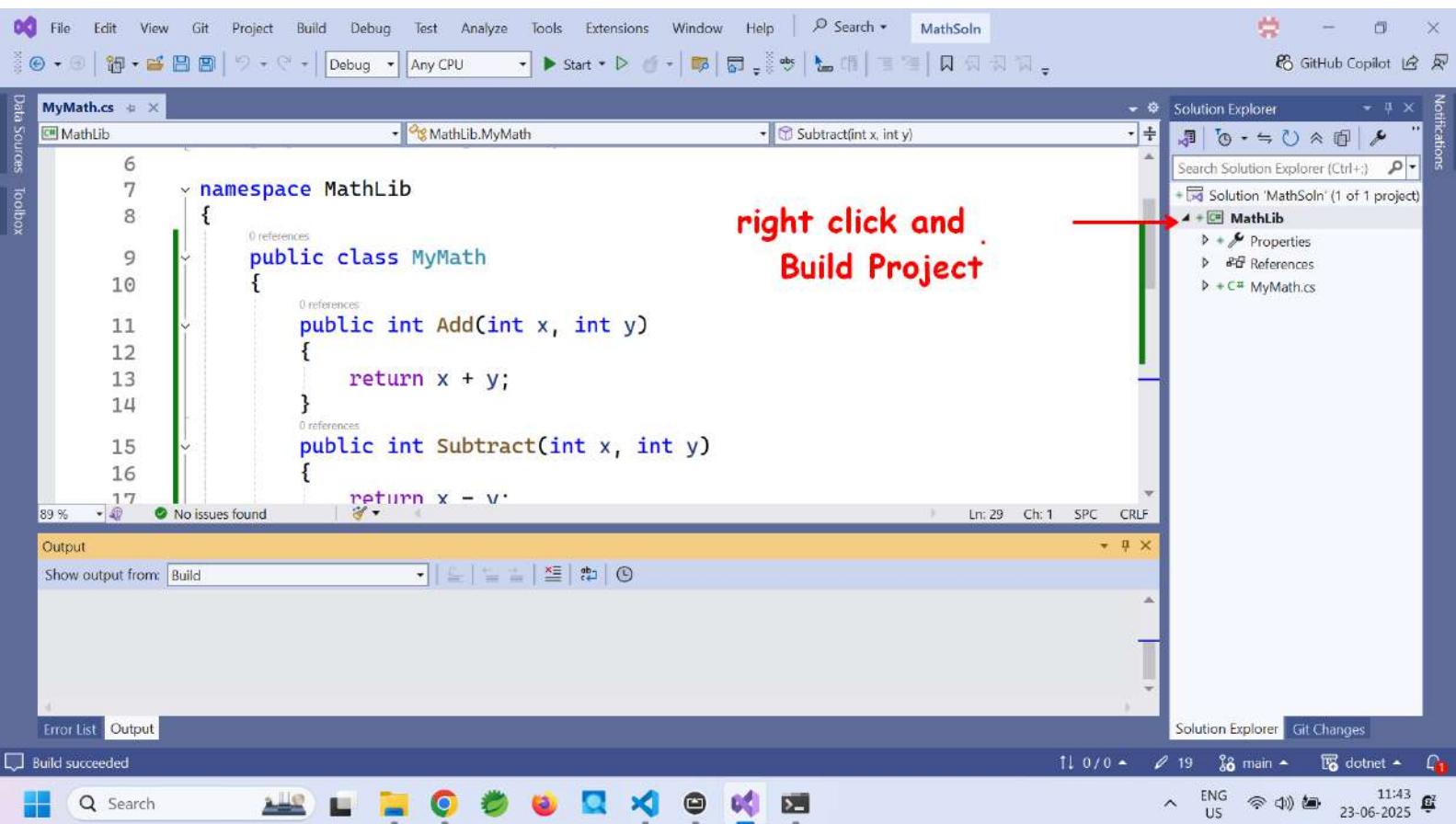


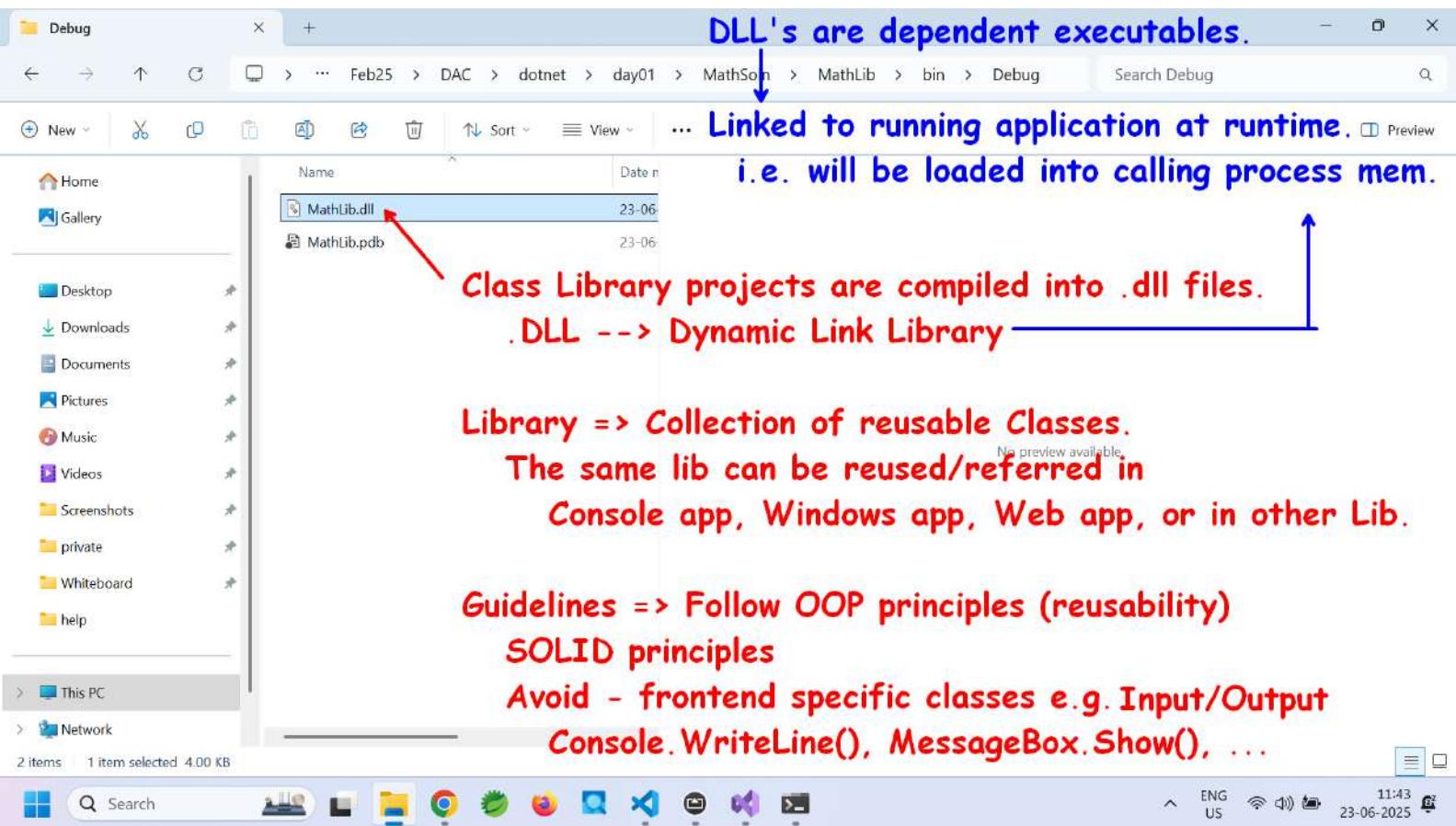
A screenshot of the Microsoft Visual Studio IDE. The main window shows a code editor with the file 'MyMath.cs' open. The code defines a public class 'MyMath' with three methods: Add, Subtract, and Multiply. A red arrow points from the word 'public' in the class declaration to a red annotation text.

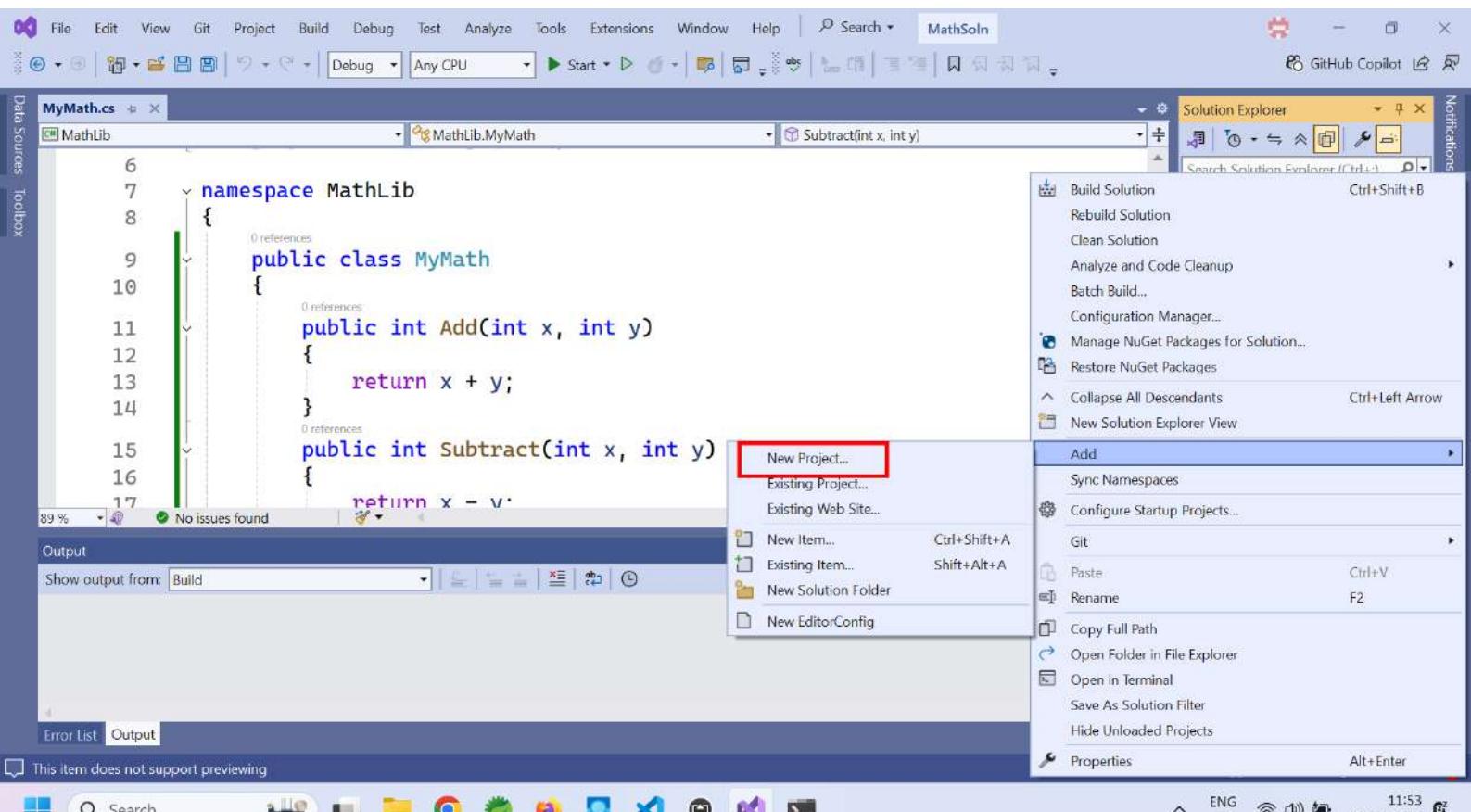
public classes are accessible outside the current project i.e. assembly.

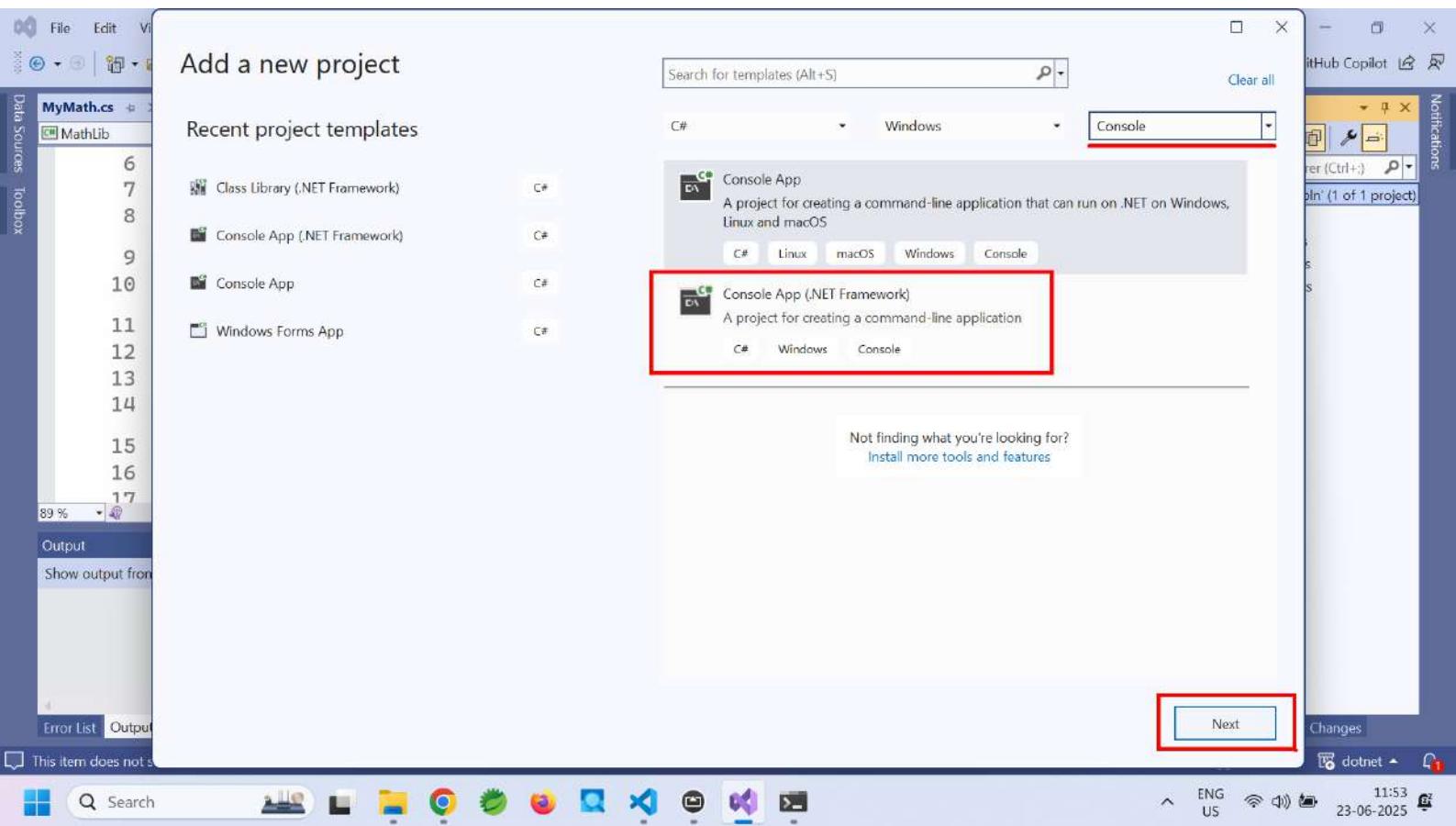
```
6  namespace MathLib
7  {
8      public class MyMath
9      {
10         public int Add(int x, int y)
11         {
12             return x + y;
13         }
14         public int Subtract(int x, int y)
15         {
16             return x - y;
17         }
18         public int Multiply(int x, int y)
19         {
20             return x * y;
21         }
22     }
23 }
```

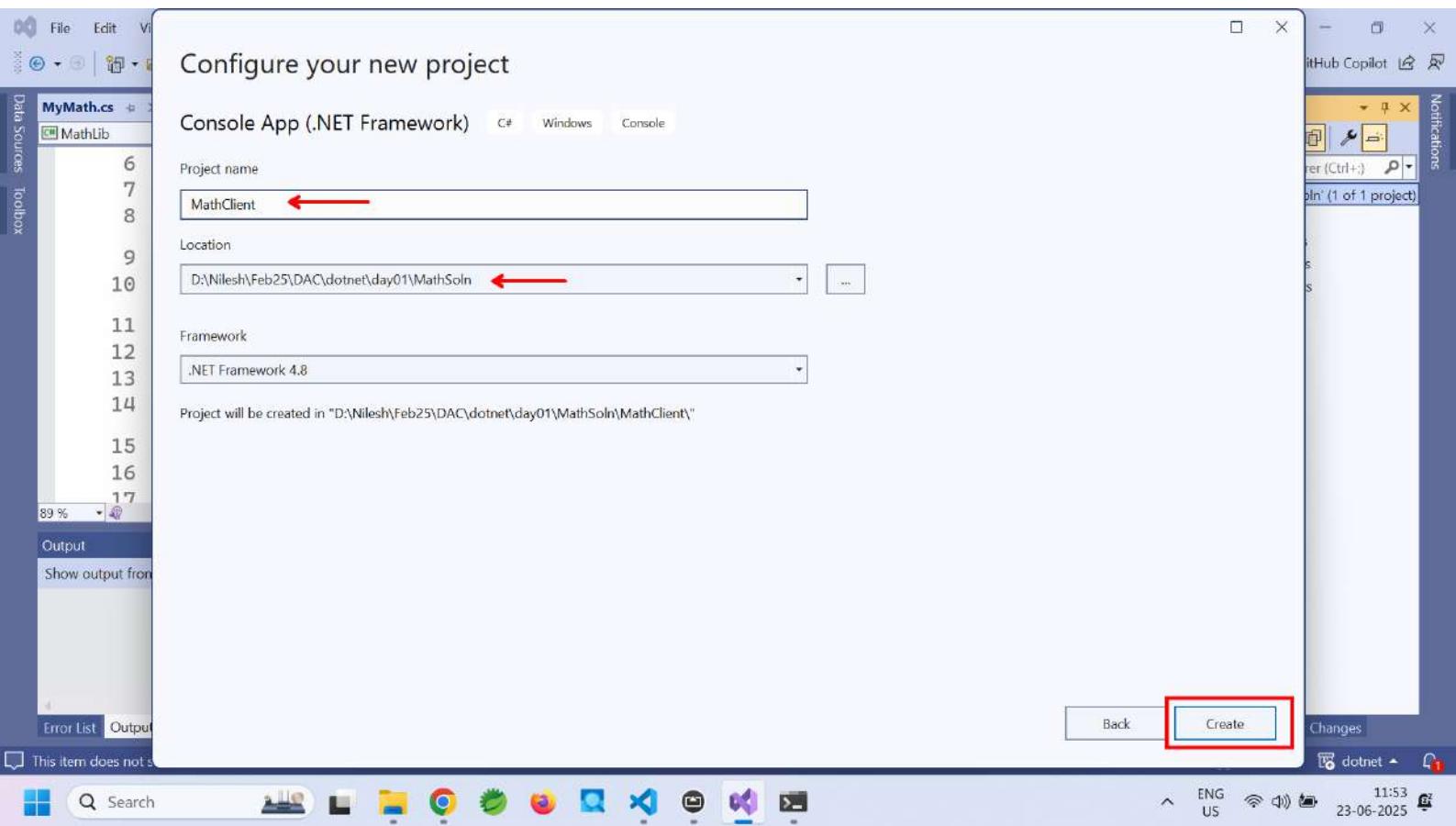
The Solution Explorer sidebar shows the project structure with 'MathLib' selected, containing 'Properties', 'References', and 'MyMath.cs'. The status bar at the bottom right shows the date and time as 23-06-2025 11:42.

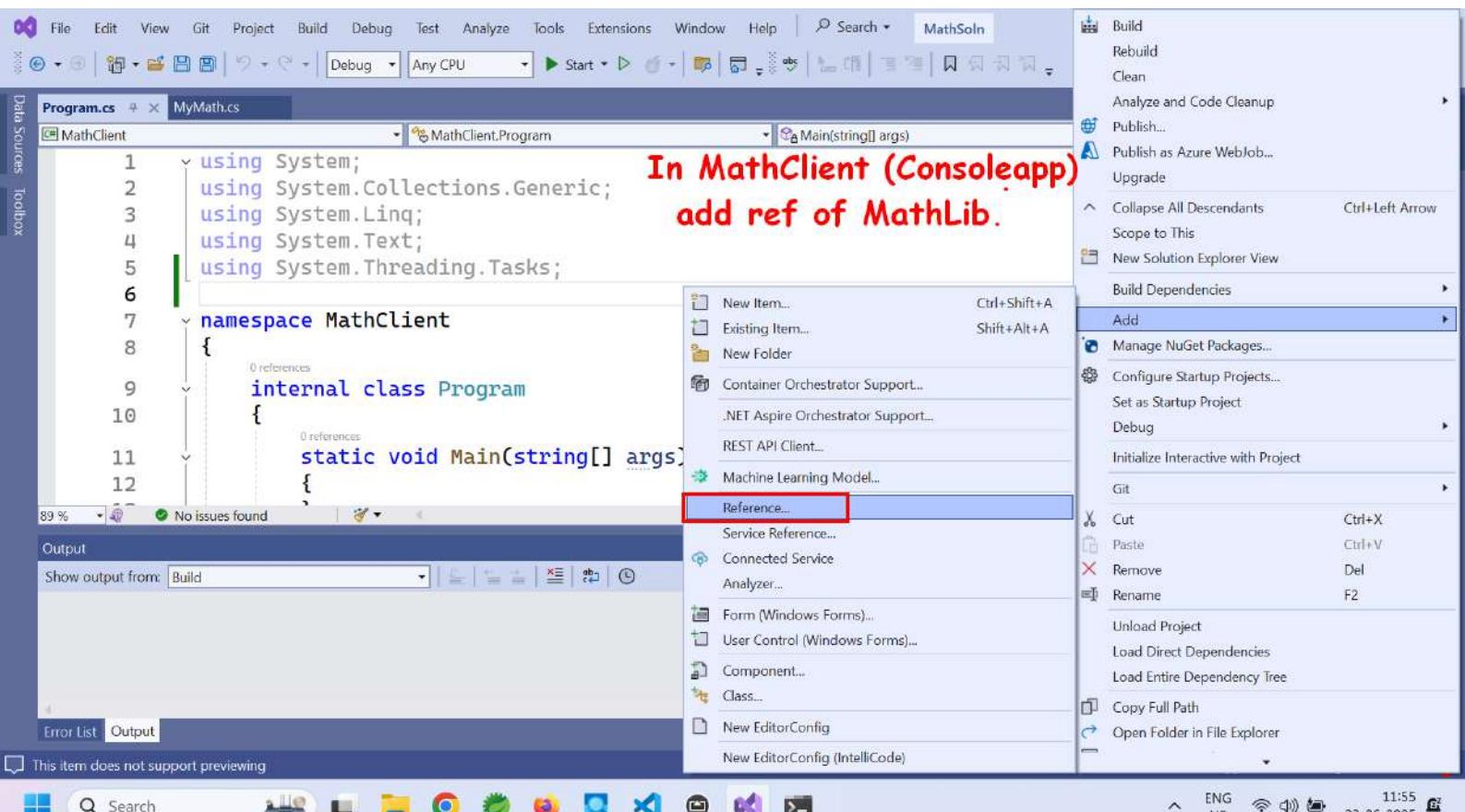






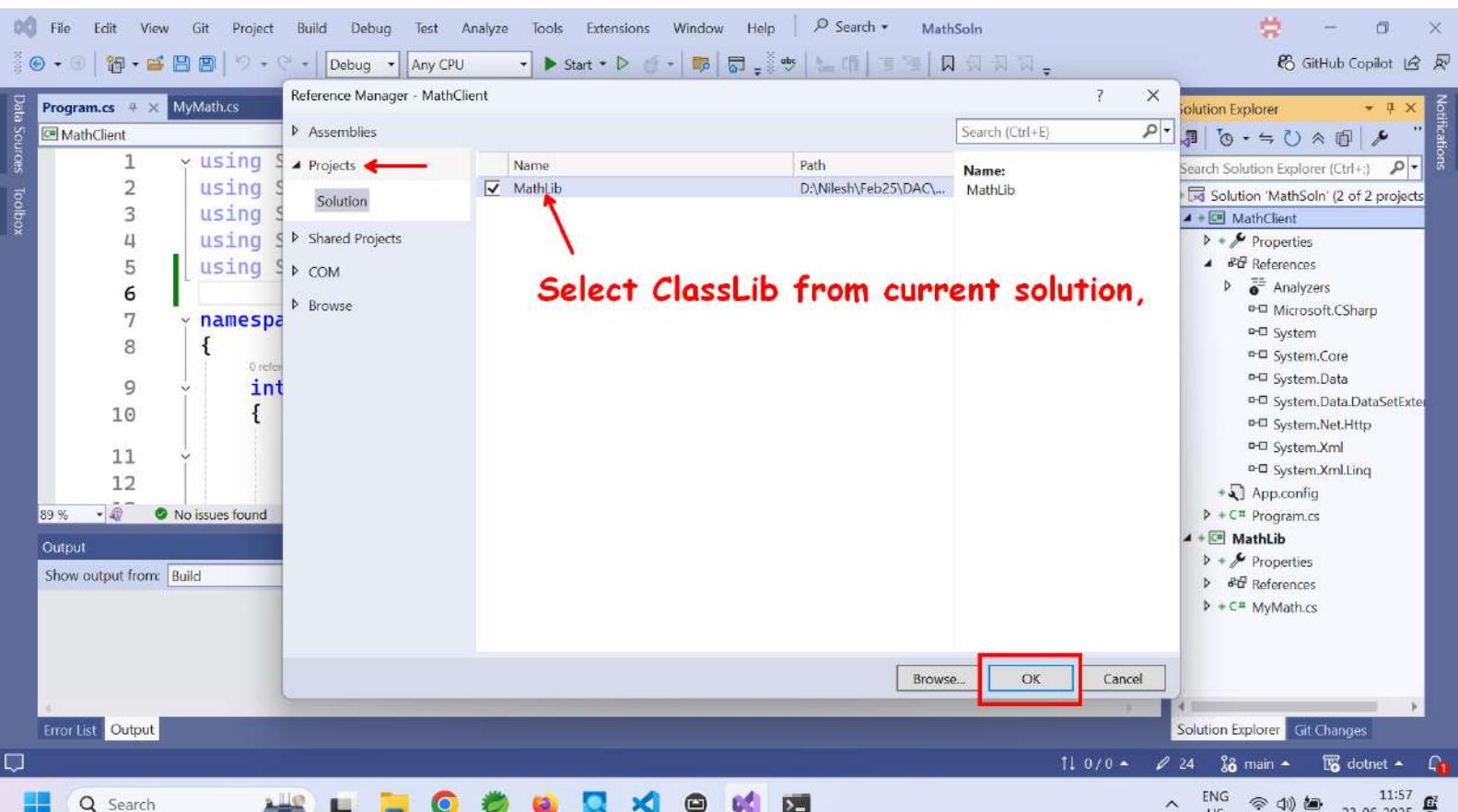


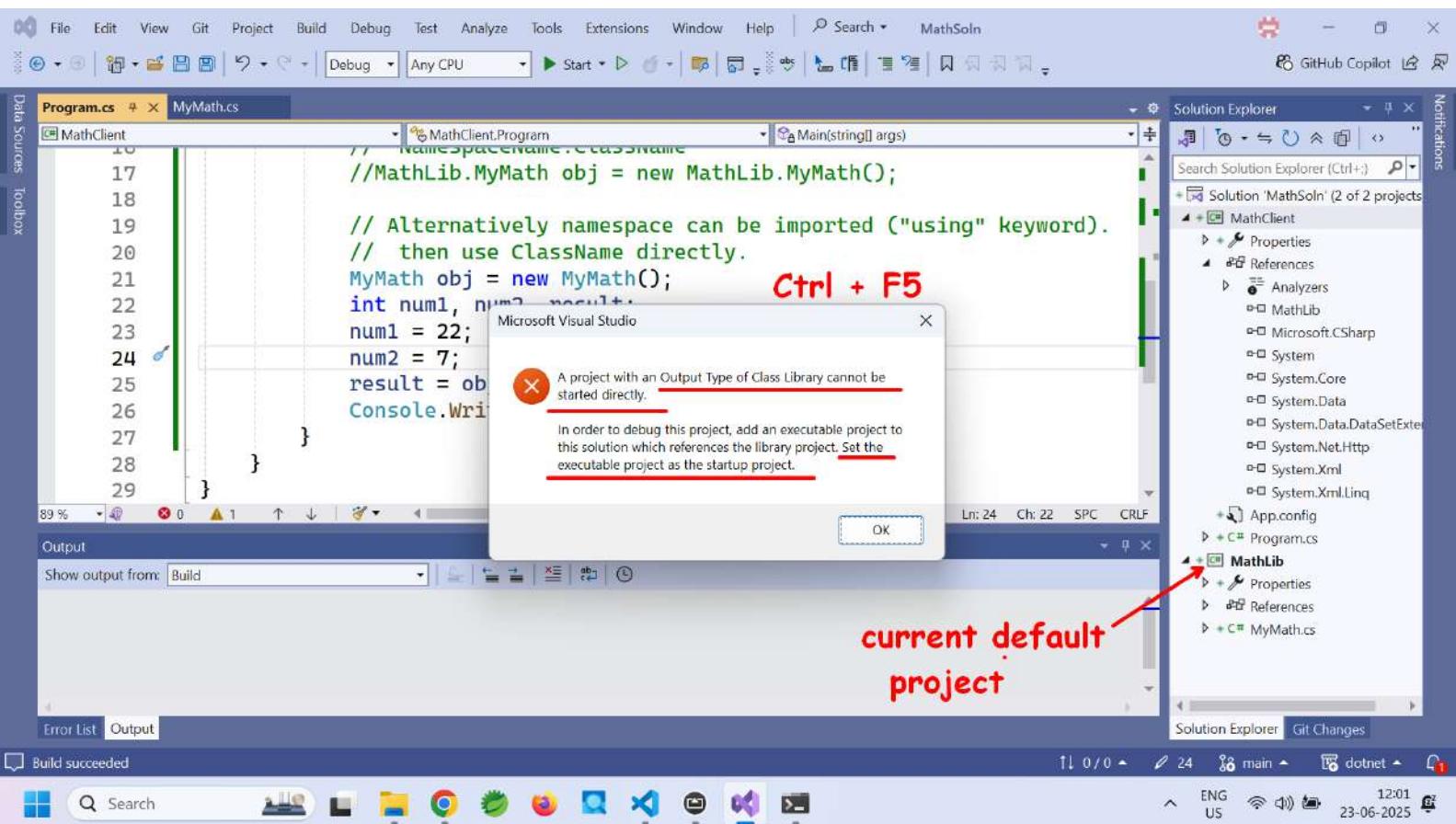




This item does not support previewing

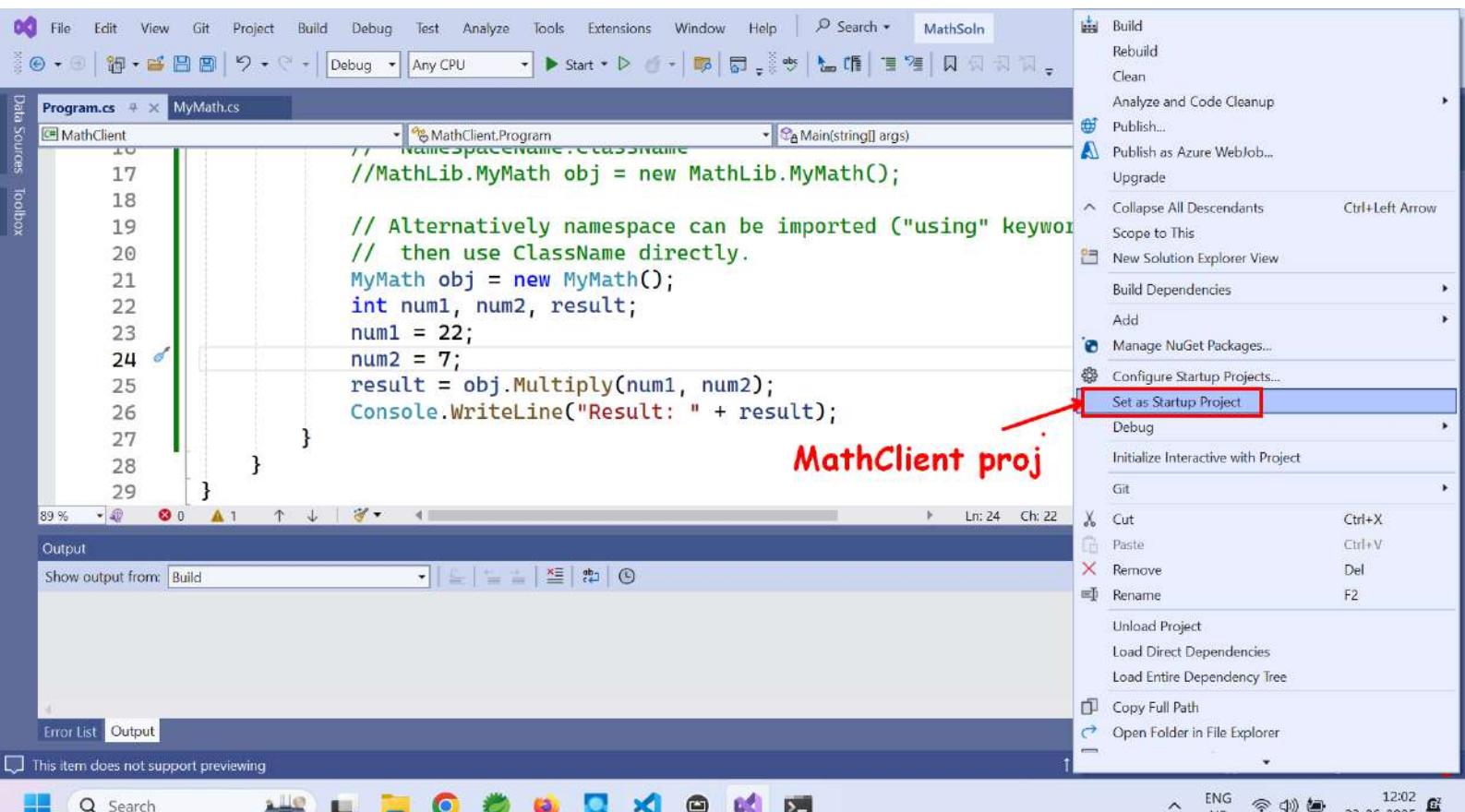
11:55
23-06-2025





12:01

23-06-2025



"internal" types are accessible in current project i.e. current assembly only -- in this example MathClient.exe

```
6  using System.Threading.Tasks;
7  using MathLib;
8
9  namespace MathClient
10 {
11     internal class Program
12     {
13         static void Main(string[] args)
14         {
15             // Object can be created using fully qualified class name
16             // NamespaceName.ClassName
17             //MathLib.MyMath obj = new MathLib.MyMath();
        }
    }
}
```

Output

Show output from: Build

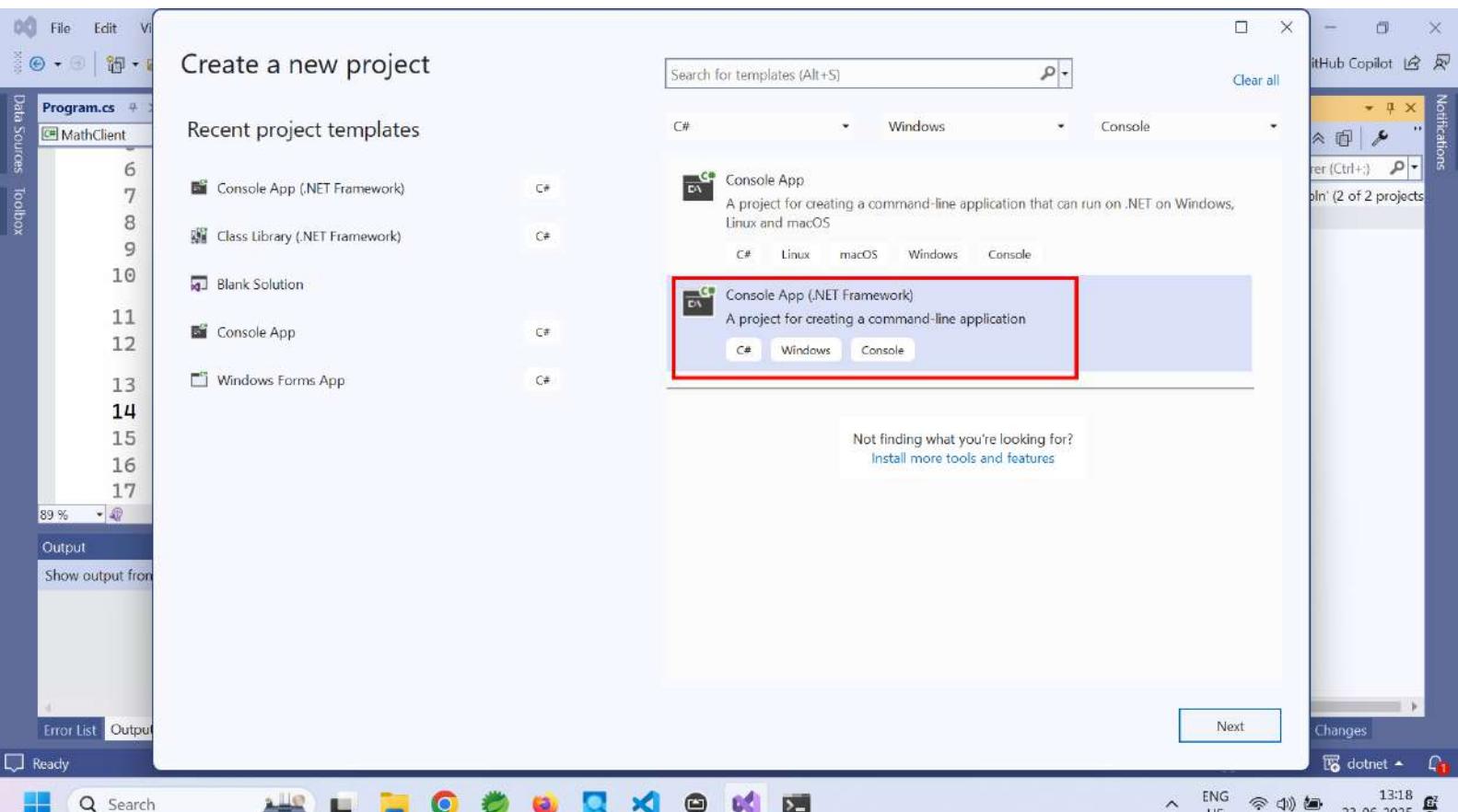
Error List Output

Solution Explorer Git Changes

Ready

Search

12:04 23-06-2025



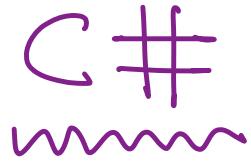
Changes
dotnet 1

13:18
23-06-2025

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays a C# code editor for a file named `Program.cs` within a project titled `ClassDemo`. The code defines a class `Person` with three private fields: `age`, `name`, and `addr`. It contains two constructors: a default constructor that initializes all fields to empty strings, and a parameterized constructor that initializes them to specific values. A red arrow points from the text "constructor chaining" to the first line of the parameterized constructor. Another red arrow points from the word "string" in the parameter list to the corresponding `string` type in the assignment statement `this.name = name;`. The Solution Explorer on the right shows the project structure with files like `Properties`, `App.config`, and `Program.cs`.

```
9  class Person
10 {
11     // fields
12     private int age;
13     private string name;
14     private string addr;
15     // constructors
16     public Person() : this(0, "", "") {
17         // if not initialized explicitly,
18         // the fields will have default values of respective types.
19         // e.g. age - int (0), name - string (null), addr - string (nul
20     }
21     public Person(int age, string name, string addr) {
22         this.age = age;
23         this.name = name;
24         this.addr = addr;
25     }
26     // methods
27 }
28 
```

constructor chaining



C#.NET @ Sunbeam Infotech

Trainer: Nilesh Ghule

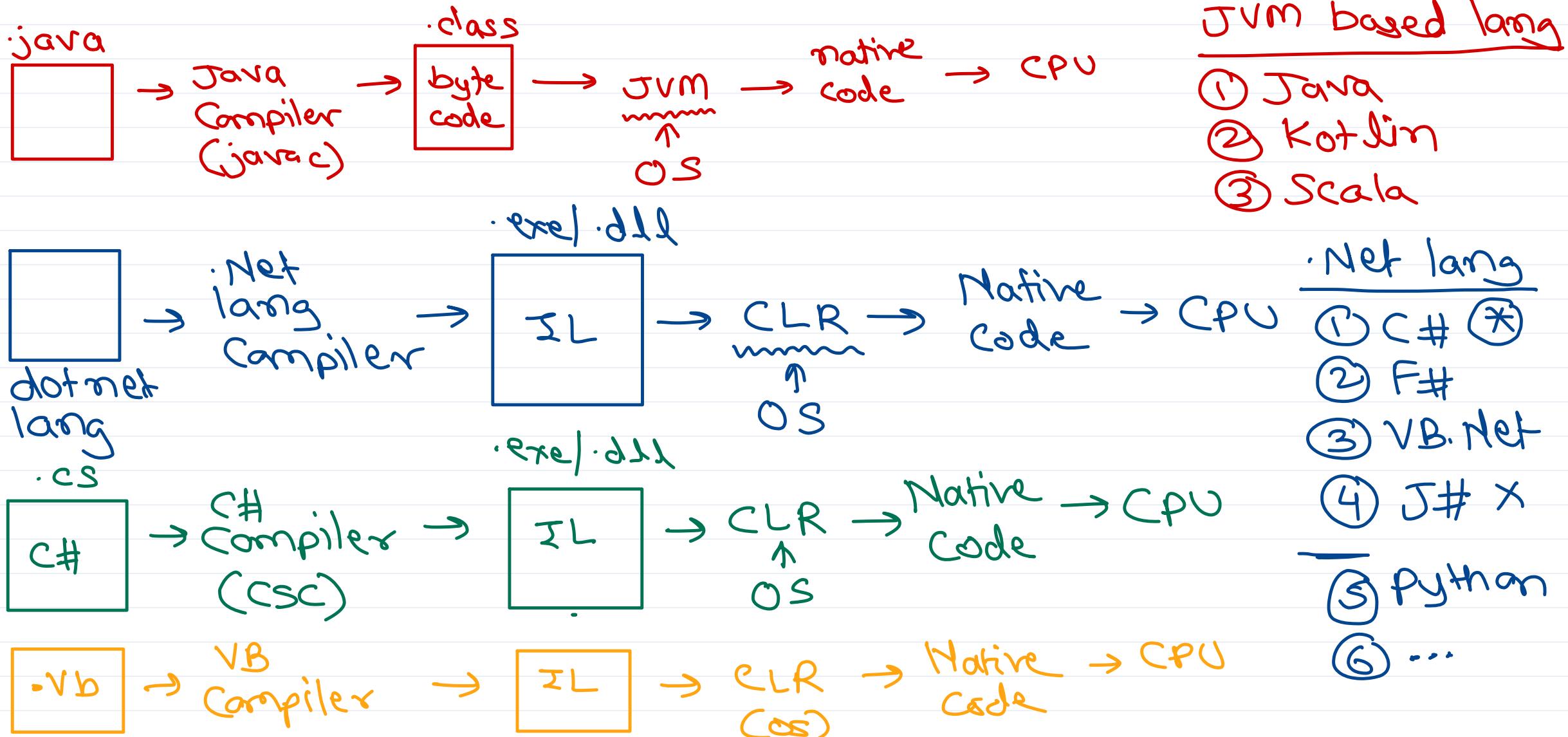


Agenda

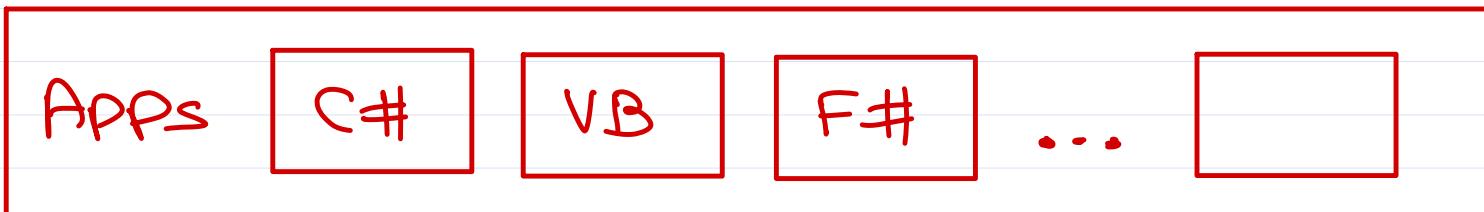
- ① Java code compilation vs .Net lang code compilation
- ② C# code vs VB .Net code compilation
- ③ .Net framework & versions.
- ④ CTS, CLS, CLR (JIT, GC, ...)
- ⑤ Installations - on client machine vs on dev machine
- ⑥ Hello world appln (using Visual Studio) Console app
Class Lib
- ⑦ .Net assembly (structure, exe execution) ildasm,
sharpLab.io,...
- ⑧ Class libs & Namespaces
- ⑨ C# classes (fields, methods, properties, constructor, destructor,...)



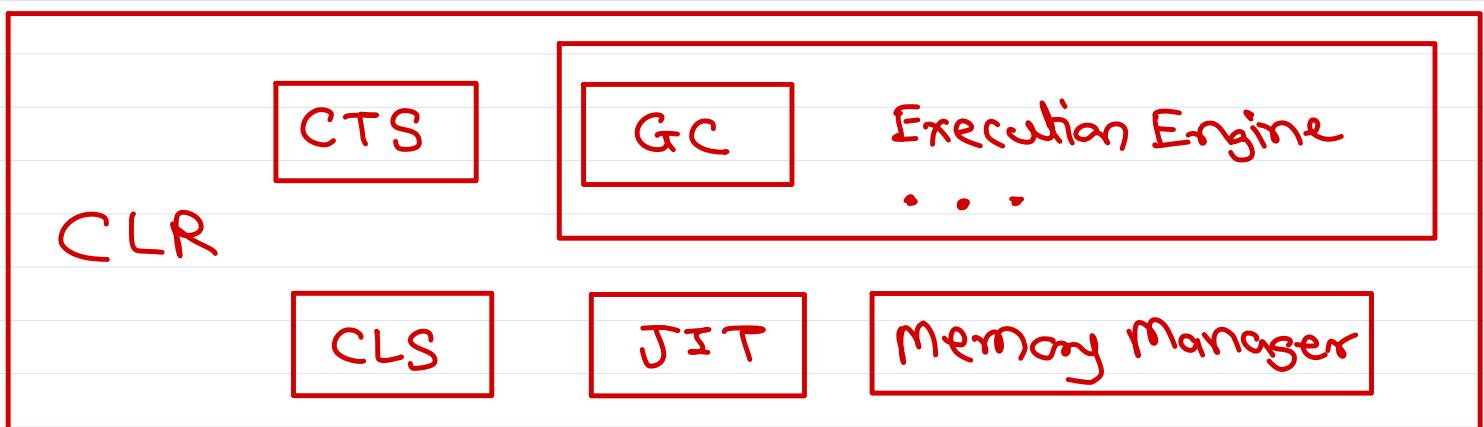
.Net - Compilation



.Net Framework



BCL (types, classes) - mscorelib.dll



client machine setup

- Net framework
 - ↳ CLR + BCL + FCL

Dev machine setup

.Net framework SDK

- Net framework
 - ↳ CLR + BCL + FCL

+ Dev tools

- ↳ Lang Compilers + Debugger
- + tools (ildasm, cts, ...)

+ IDE (RAD tools) + ...

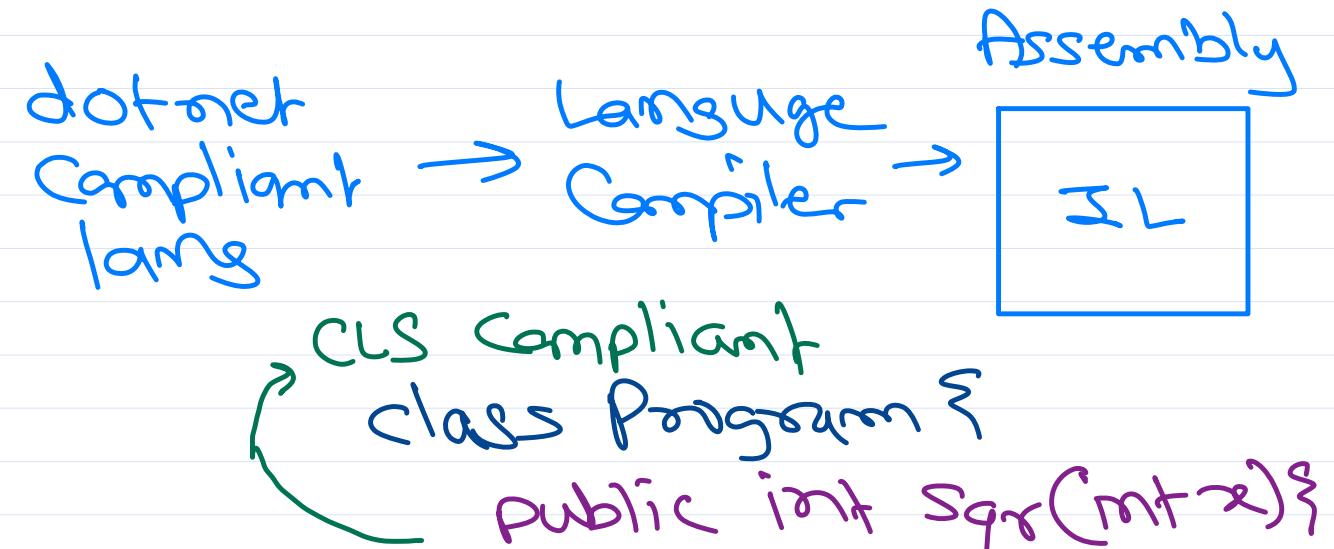
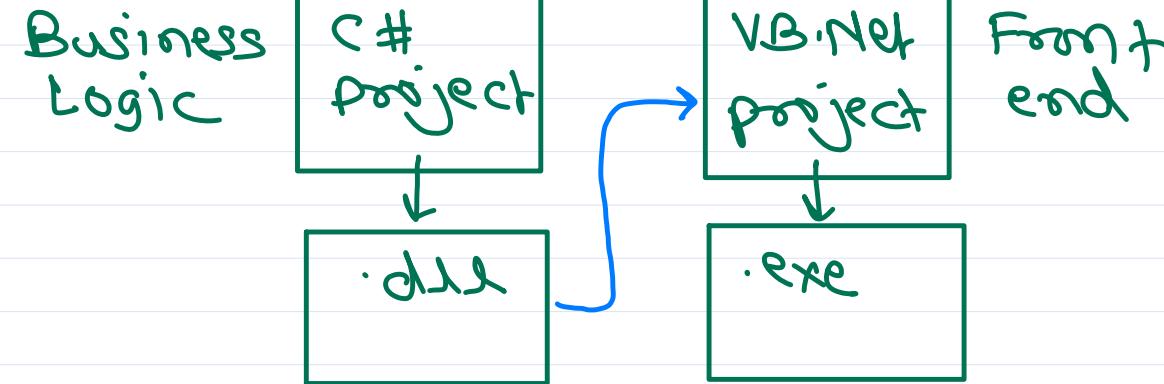
Visual Studio 2022

Community
Professional
Enterprise



CLR – CTS & CLS

* Language interoperability



* Languages must follow certain rules.

→ .Net Compliant lang → lang Compiler

→ lang rules → CLS

→ objects allocated using new

→ only signed types

→ stack based execution

CLS Non-compliant



CLR – CTS & CLS

Common Type System → Types in IL → understood by CLR

CTS types

→ Value Types → represent value

① Int16 ⑥ Decimal

② Int32 ⑦ Boolean

③ Int64 ⑧ Char

④ Single

⑤ Double

→ Reference Types → represent

① Class (e.g. String, ...) ref / addr.

② Interface

③ Array

④ Delegate

⑤ Object

Int32 a=123; a 123

* Typically on stack (locals)

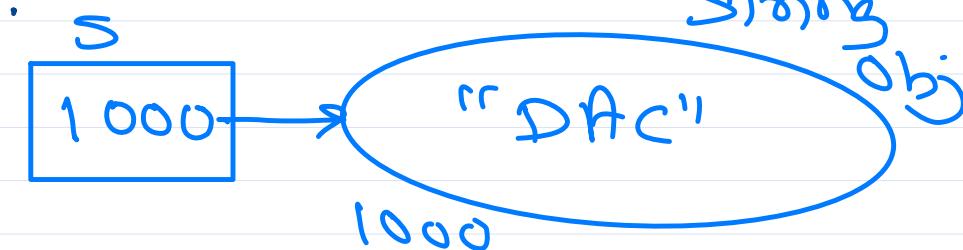
* If part/field in some obj
→ on Heap.

Structure

Enum

String s = "DAC";

String



CLR – JIT

* JIT

- ✓ IL code → JIT → Native code
- ✓ Does conversion Method by Method.
- ✓ Frequently called methods - native code - Cached.
- ✓ JIT types

① Standard JIT (Part of CLR) - all devices.

② Economy JIT - on small devices (phones, embedded);
- .Net compact framework - outdated.

③ Pre-JIT (AOT - Ahead of Time):

- all IL code → Pre-JIT → Native code ↴
→ ngen.exe

Before
Execution



* GC → auto deallocate
unreferenced objects

- Mark - Sweep - Compact algorithm.

* Execution Engine

↳ GC
↳ Security

* Memory Manager

- ↳ allocates memory for
 - class loading
 - stack
 - cache
 - ...

.Net Assemblies

File Extensions

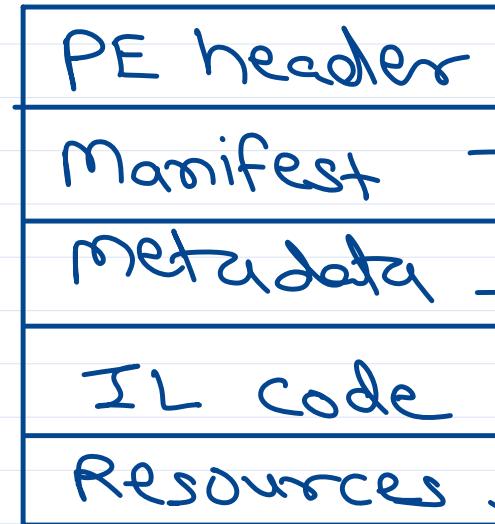
① .exe

- executable file
- OS creates process when end user starts execution.

② .dll

- lib of classes (reuse)
- dependent executable loaded in calling process memory, when its classes accessed in that process.

Assembly Structure



→ CLR info, referenced assemblies info, versions, ...

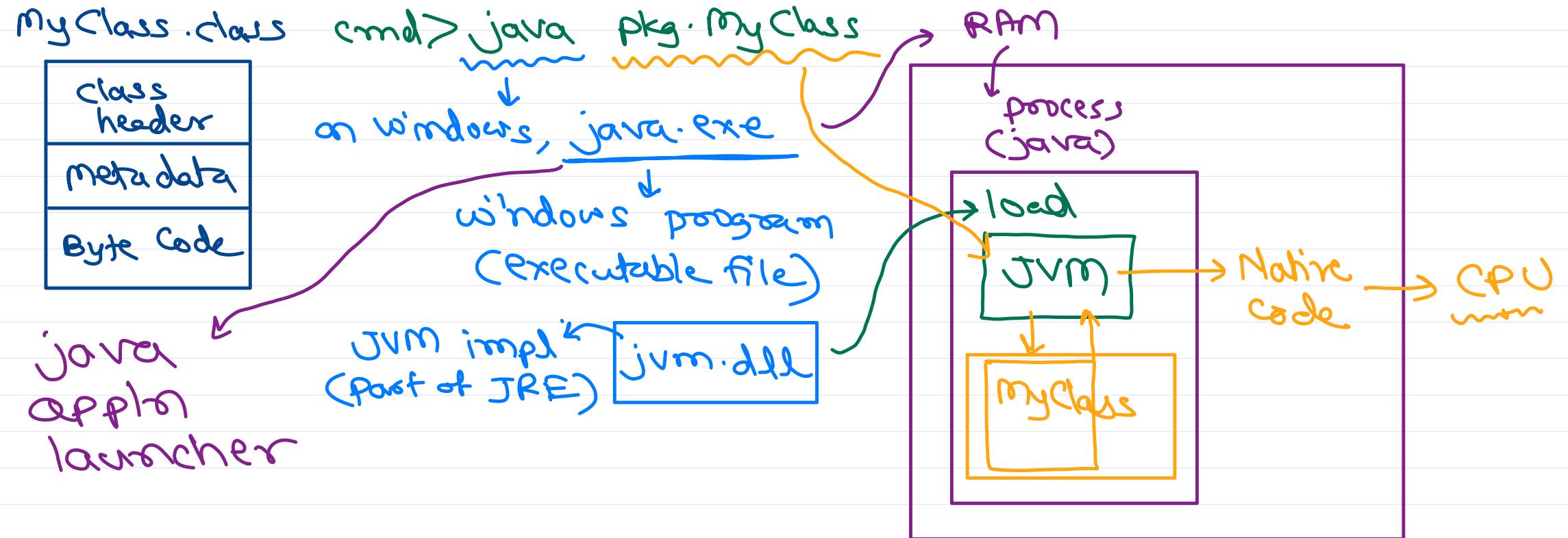
a.k.a. assembly metadata

type metadata - type name, fields, methods, ctors, ...

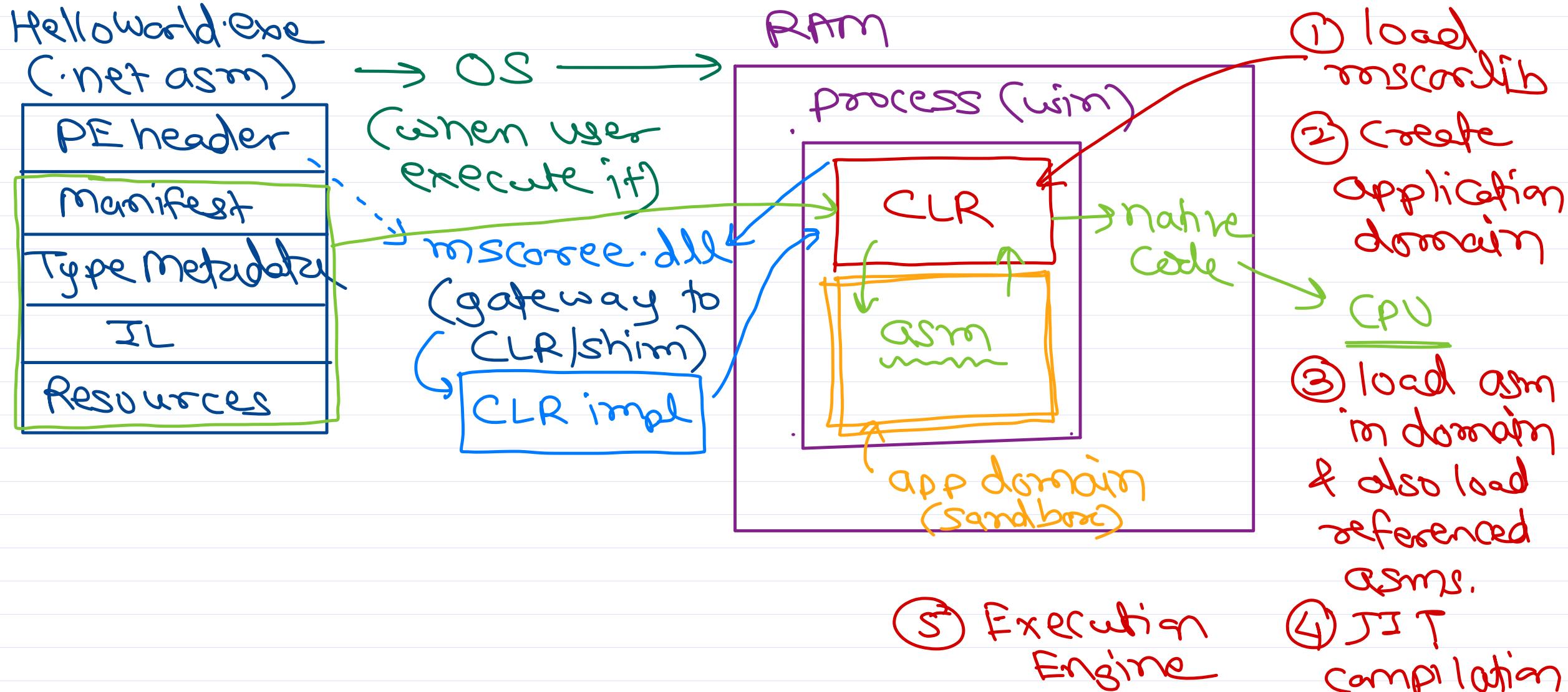
images, config files, ...



Java execution



.Net program execution





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

.NET

Overview of .NET Framework

1. Historical Context & Introduction

- **Released in 2002** as Microsoft's first unified platform for Windows application development.
- Designed to **replace COM (Component Object Model)** and simplify Windows programming.
- Initially targeted **Windows-only** desktop (WinForms), web (ASP.NET), and service applications.
- **Key Milestones:**
 - **.NET 1.0 (2002)**: Introduced CLR, CTS, and base class libraries (BCL).
 - **.NET 2.0 (2005)**: Generics, ASP.NET 2.0, and ADO.NET enhancements.
 - **.NET 3.5 (2007)**: LINQ, WPF, WCF, WF, and `var` keyword (C# 3.0).
 - **.NET 4.0 (2010)**: Dynamic Language Runtime (DLR), `dynamic`, Parallel LINQ (PLINQ).
 - **.NET 4.8 (2019)**: Last major version (now in maintenance mode).

2. Key Components

1. Common Language Runtime (CLR)

- Manages memory (garbage collection), thread execution, and exception handling.
- Compiles Intermediate Language (IL) to native code via **JIT (Just-In-Time)** compilation.

2. Base Class Library (BCL)

- Provides foundational APIs for I/O, collections, threading, and reflection (`System.*` namespaces).

3. Framework Class Library (FCL)

- Extends BCL with **Windows-specific** libraries (e.g., WinForms, WPF, ASP.NET).

4. Common Type System (CTS)

- Defines rules for type safety and cross-language interoperability (e.g., C#, VB.NET).

5. Assembly Model

- **DLL/EXE** files containing IL code and metadata. Supports versioning (GAC) and side-by-side execution.

3. Supported Languages

- **Officially support:** 35+ Languages
- **Community driven:** 80+ Languages
- **Most popular:** C#.
- **Interoperability:** COM via **RCW** (Runtime Callable Wrappers) and **P/Invoke** for native code.

4. Application Models

- **Desktop:** WinForms (1.0), WPF (3.0).
- **Web:** ASP.NET WebForms (1.0), MVC (4.0), Web API (4.5).
- **Services:** WCF (3.0), Remoting (1.0).
- **Data:** ADO.NET (1.0), Entity Framework (3.5).

5. Deployment Model

- **Windows-Only:** Tightly integrated with OS (registry, GAC).
- **XCOPY Deployment:** Limited support (DLL hell mitigated via strong naming).
- **Setup Projects:** MSI installers or ClickOnce for updates.

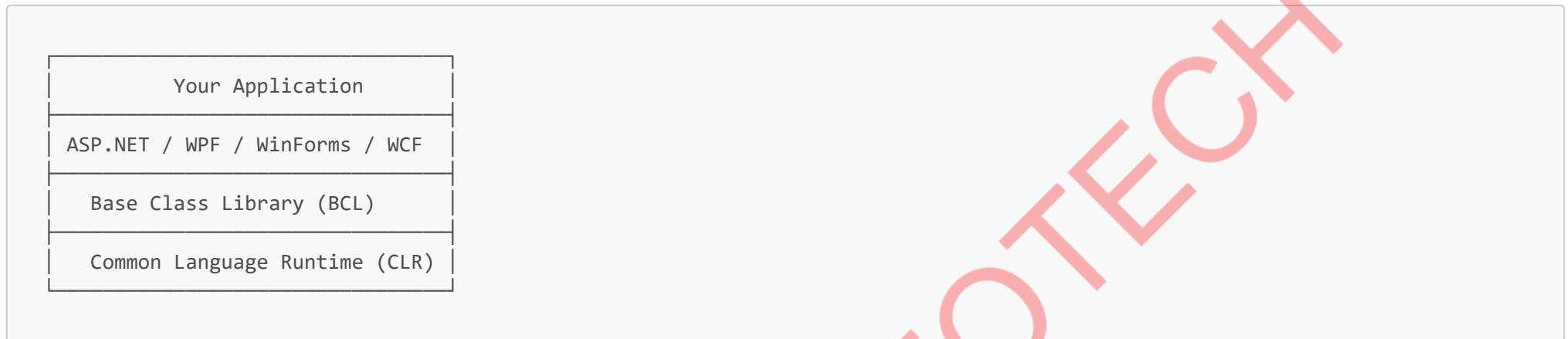
6. Advantages

- **Mature Ecosystem:** Extensive libraries for enterprise applications.
- **Windows Integration:** Deep OS access (e.g., COM+, DirectX).
- **Stability:** Long-term support (LTS) for critical versions.

7. Limitations

- **Windows Dependency:** No cross-platform support (unlike .NET Core).
- **Heavyweight:** Large runtime (~200 MB) and slower updates.
- **Performance:** Lacks modern optimizations (AOT, minimal APIs).

8. Example: Framework Stack



9. Version Compatibility

- **Backward-Compatible:** Apps targeting older versions run on newer CLRs (with quirks).
- **Side-by-Side Execution:** Multiple framework versions can coexist (e.g., 4.0 and 4.8).

10. MSDN References

- [.NET Framework Guide](#)
- [CLR Overview](#)
- [BCL Documentation](#)

C# Program Compilation and Execution

1. Overview of the Compilation Process

The C# compilation and execution pipeline follows a **multi-stage process**, converting human-readable code into machine-executable instructions. This involves:

1. **Source Code Compilation → Intermediate Language (IL)**
2. **Just-In-Time (JIT) Compilation → Native Machine Code**
3. **Execution by the CLR (Common Language Runtime)**

2. Step 1: Writing C# Source Code

- C# programs are written in **.cs** files (e.g., **Program.cs**).
- Example:

```
using System;
class Program {
    static void Main() {
        Console.WriteLine("Hello, World!");
    }
}
```

3. Step 2: Compilation to Intermediate Language (IL)

- The **C# Compiler (csc.exe)** converts **.cs** files into **IL code** (stored in **.exe** or **.dll** files).
- **IL (MSIL / CIL)** is a CPU-agnostic, stack-based instruction set.
- Tools:
 - **csc (Command-line compiler)**
 - **MSBuild / Visual Studio** (managed build systems).
- Example (IL snippet from **ildasm**):

```
.method private hidebysig static void Main() cil managed {
    ldstr "Hello, World!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

4. Step 3: Assembly (.exe) Execution

- A **.NET Assembly** contains:
 - **IL Code** (executable logic).

- **Metadata** (types, methods, dependencies).
 - **Manifest** (version, culture, strong name).
 - **Resources** (images, configs).
- The **CLR** loads assemblies into **AppDomains** (logical containers for isolation).
 - At runtime, the **JIT Compiler** converts IL into **native machine code** (optimized for the CPU).
 - CLR's **Execution Engine** runs the native code, managed by:
 - **Garbage Collector (GC)** → Memory management.
 - **Exception Handler** → Structured error handling.
 - **Security Engine** → Code access security (CAS).

5. Compilation via Command Line

```
csc /target:exe /out:HelloWorld.exe Program.cs
```

6. Tools for Inspection

- **ildasm.exe** → Disassembles IL from assemblies.
- **pverify** → Validates type safety.
- **ILSpy** → Open-source .NET assembly browser and decompiler.

12. MSDN References

- [C# Compiler Options](#)
- [Managed Execution Process](#)
- [JIT Compilation](#)

CLR (Common Language Runtime)

1. Overview of the CLR

1.1 Definition & Role

- The **CLR** is the **execution engine** of the .NET Framework, responsible for running managed code.
- It provides services like **memory management, security, exception handling, and threading**.
- Acts as a **virtual machine** that abstracts hardware differences.

1.2 Key Responsibilities

- **Just-In-Time (JIT) Compilation** – Converts IL to native machine code.
- **Memory Management** – Allocates and deallocates memory (Garbage Collection).
- **Type Safety & Security** – Enforces access rules and prevents buffer overflows.
- **Exception Handling** – Provides structured error handling.
- **Thread Management** – Manages multithreading and synchronization.
- **AppDomain Management** – Isolates applications within a single process.

1.3 CLR vs. JVM (Comparison)

Feature	CLR (.NET)	JVM (Java)
Language Support	C#, F#, VB.NET	Java, Kotlin, Scala
Memory Model	Automatic GC + IDisposable	Automatic GC (no finalize control)
Platform	Originally Windows-only	Cross-platform

2. Just-In-Time (JIT) Compilation

2.1 How JIT Works

1. **IL Code Loading** → CLR loads Intermediate Language (IL) from assemblies.
2. **Method Stub Generation** → On first call, JIT generates a **stub** pointing to native code.

3. **Native Code Compilation** → JIT compiles IL into **optimized machine code**.
4. **Execution** → The compiled code runs directly on the CPU.

2.2 Types of JIT Compilation

- **Standard JIT** – Compiles methods on first use (default).
- **Economy JIT** – Used in low-memory scenarios (minimal optimizations) - obsolete.
- **Pre-JIT (`ngen.exe`)** – Pre-compiles assemblies to native code (reduces startup time).

2.3 Performance Considerations

- **First-run penalty** (JIT overhead on initial calls).
- **Tiered Compilation (.NET Core+)** – Starts with quick JIT, re-optimizes hot paths.

3. Memory Management & Garbage Collection (GC)

3.1 Memory Allocation

- **Stack** → Stores value types (primitives, structs) and method call frames.
- **Heap** → Stores reference types (objects, arrays). Managed by GC.

3.2 Garbage Collection (GC) Process

1. **Mark Phase** – Identifies reachable objects (starting from GC roots).
2. **Sweep Phase** – Reclaims memory from unreachable objects.
3. **Compact Phase** – Defragments memory (reduces fragmentation).

3.3 Generations in GC

Generation	Description	Example Objects
Gen 0	Short-lived	Local variables, temp objects

Generation	Description	Example Objects
Gen 1	Medium-lived	Cached data
Gen 2	Long-lived	Static fields, singletons

3.4 GC Modes

- **Workstation GC** – Optimized for UI apps (low latency).
- **Server GC** – Optimized for throughput (multiple CPU cores).

3.5 Manual Memory Management

- We will demonstrate this later.
- **IDisposable** → Used for deterministic cleanup (e.g., file handles).
- **using statement** → Ensures **Dispose()** is called.

```
using (var file = new FileStream("test.txt", FileMode.Open)) {
    // Automatically disposed
}
```

4. AppDomain (Application Domain) Management

4.1 What is an AppDomain?

- A **lightweight process-like isolation** mechanism within a single OS process.
- Allows **loading/unloading assemblies without killing the process**.

4.2 Key Use Cases

- **Plugin Architectures** – Load/unload DLLs dynamically.

- **Fault Isolation** – Crash in one AppDomain doesn't affect others.
- **Security Sandboxing** – Restrict permissions per domain.

4.3 Limitations

- **.NET Core+ Deprecation** – AppDomains are **not fully supported** in .NET Core (replaced by `AssemblyLoadContext`).
- **Performance Overhead** – Cross-domain calls require marshaling.

5. MSDN References

- [CLR Overview](#)
- [JIT Compilation](#)
- [Garbage Collection](#)
- [AppDomains](#)

CLS (Common Language Specification) and CTS (Common Type System)

(Core Pillars of .NET Language Interoperability)

1. Introduction to CLS and CTS

1.1 Role in .NET Ecosystem

- **CTS** defines **how types are declared, used, and managed** in .NET.
- **CLS** ensures **cross-language compatibility** by defining a subset of CTS rules.
- Together, they enable **C#, F#, VB.NET** to interoperate seamlessly.

1.2 Historical Context

- Introduced in **.NET Framework 1.0 (2002)** to solve language fragmentation (e.g., C++ vs VB6).
- Critical for **multi-language projects** (e.g., C# library consumed by VB.NET).

2. Common Type System (CTS)

2.1 Purpose

- Standardizes **type definitions** across .NET languages.
- Ensures **type safety** and **memory integrity** via runtime checks.

2.2 Key Components

(A) Type Categories

Category	Description	Examples
Value Types	Stored on stack, direct data.	int, struct, enum
Reference Types	Stored on heap, accessed via reference.	class, string, delegate

(B) Type Members

- **Fields, Properties, Methods, Events** (unified across languages).
- Example: A **class** in C# compiles to the same IL as a VB.NET **Class**.

(C) Type Hierarchy

- All types inherit from **System.Object** (directly or indirectly).
- Supports **interfaces, inheritance, and polymorphism**.

2.3 Example: CTS in Action

```
// C# Code (CTS-compliant)
public struct Point { public int X; public int Y; }
```

```
' VB.NET Code (same CTS type)
Public Structure Point
    Public X As Integer
    Public Y As Integer
End Structure
```

→ Both compile to identical IL metadata.

3. Common Language Specification (CLS)

3.1 Purpose

- Defines a **subset of CTS rules** that languages **must follow** to ensure interoperability.
- Avoids language-specific quirks (e.g., C#'s `uint` isn't CLS-compliant).

3.2 Key CLS Compliance Rules

- ✓ **No unsigned types** (e.g., `uint` → use `int`).
- ✓ **Method overloading** must differ by more than return type.
- ✓ **Identifiers** must be case-insensitive across languages.
- ✓ **No global methods** (all code must be in a type).

3.3 Enforcing CLS Compliance

- Use `[assembly: CLSCompliant(true)]` to enable checks:

```
[assembly: CLSCompliant(true)]

public class MyClass
{
```

```
// Warning: uint is not CLS-compliant
public uint Counter { get; set; }
}
```

3.4 Example: Non-Compliant vs Compliant Code

```
// ✗ Non-CLS-Compliant (VB.NET can't use this)
public ulong GetValue() { return 0; }

// ☑ CLS-Compliant Alternative
public long GetValue() { return 0; }
```

4. How CLS and CTS Work Together

4.1 Compilation Flow

1. **Source Code** → Written in C#/VB.NET/F#.
2. **Compiler** → Enforces CTS/CLS rules, outputs IL + metadata.
3. **Runtime** → Uses CTS to enforce type safety during execution.

4.2 Cross-Language Interop Example

- A **C# interface** can be implemented by a **VB.NET Class**.
- A **F# record** can be consumed by **C#** as a POCO.

5. Why Developers Should Care

5.1 Benefits

- ✓ **Language Flexibility** – Mix C#, F#, VB.NET in one project.
- ✓ **Library Reuse** – NuGet packages work across languages.
- ✓ **Future-Proofing** – Ensures compatibility with new .NET languages.

5.2 Pitfalls to Avoid

- **Unsigned types** (`uint`, `ulong`) break CLS compliance.
- **Case-sensitive identifiers** cause issues in VB.NET.

6. MSDN References

- [Common Type System \(CTS\)](#)
- [CLS Compliance Rules](#)

.Net Assemblies (.exe or .dll)

1. Definition and Role of Assemblies

- An **assembly** is the fundamental **deployment, versioning, and security unit** in .NET.
- It can be either an **executable (EXE)** or a **library (DLL)**.
- Contains **Intermediate Language (IL) code, metadata, and resources**.
- Serves as the **building block** of .NET applications, enabling modularity and reuse.

2. Types of Assemblies

1. Private Assemblies

- Deployed in the application's local directory.
- Used only by a single application.
- No versioning constraints (simple deployment).

2. Shared (Strong-Named) Assemblies

- Stored in the **Global Assembly Cache (GAC)**.
- Have a **strong name** (public key, version, culture).
- Used by multiple applications (e.g., `mscorlib.dll`).

3. Satellite Assemblies

- Contain **localized resources** (e.g., strings for different languages).
- Follow naming conventions (e.g., `MyApp.resources.dll`).

3. Physical Structure of an Assembly

An assembly is a **PE (Portable Executable) file** with the following components:

1. PE Header

- Contains metadata about the file format (COFF header).
- Specifies whether it is a **DLL or EXE**.

2. CLR Header

- Indicates **CLR version**, **entry point**, and **metadata location**.

3. Metadata Tables

- Describe **types, methods, fields, and dependencies**.
- Used by the CLR for **type safety** and **reflection**.

4. IL Code

- The compiled **Intermediate Language** instructions.
- Converted to **native code** at runtime by the JIT compiler.

5. Resources

- Embedded files (images, strings, configs).
- Accessed via `System.Resources.ResourceManager`.

6. Manifest

- Contains **assembly identity** (name, version, culture).
- Lists **referenced assemblies** and **security permissions**.

4. Logical Structure (Modules & Multi-File Assemblies)

- A **single-file assembly** contains all components in one EXE/DLL.
- A **multi-file assembly** splits code/resources across multiple modules (rarely used).
 - Example:

```
MainModule.dll (contains manifest)  
Helper.netmodule (IL only)  
Resources.resources (satellite file)
```

5. Assembly Manifest (Critical Metadata)

The manifest includes:

- **Assembly Name** (e.g., `MyApp`, `Version=1.0.0.0`).
 - **Public Key Token** (for strong-named assemblies).
 - **Referenced Assemblies** (dependencies).
 - **Security Requirements** (permissions requested).
- **Example (via `ildasm.exe`):**

```
.assembly MyApp {  
    .ver 1:0:0:0  
    .publickey = (...  
}  
.assembly extern mscorlib {  
    .ver 4:0:0:0  
}
```

6. Strong-Named Assemblies

- Signed with a **public/private key pair** for uniqueness.
- Prevents **DLL hijacking** and enables GAC deployment.
- Generated using:

```
sn -k MyKeyPair.snk
```

- Referenced in code:

```
[assembly: AssemblyKeyFile("MyKeyPair.snk")]
```

7. Global Assembly Cache (GAC)

- A **machine-wide repository** for shared assemblies.
- Located at `%windir%\Microsoft.NET\assembly`.
- Managed via:

```
gacutil /i MyAssembly.dll # Install  
gacutil /u MyAssembly     # Uninstall
```

8. Versioning and Side-by-Side Execution

- .NET enforces **versioning** to avoid "DLL Hell".
- Format: **Major.Minor.Build.Revision** (e.g., **1.0.2.45**).
- **Binding Policy:**
 - Configurable via **app.config** (e.g., redirects).

```
<dependentAssembly>  
  <assemblyIdentity name="MyLib" publicKeyToken="..." />  
  <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0" />  
</dependentAssembly>
```

9. Reflection: Inspecting Assemblies at Runtime

- The **System.Reflection** namespace allows dynamic inspection:

```
Assembly assembly = Assembly.LoadFrom("MyLib.dll");  
Type[] types = assembly.GetTypes();  
foreach (Type t in types) {  
    Console.WriteLine(t.FullName);  
}
```

10. Tools for Working with Assemblies

Tool	Purpose
<code>ildasm.exe</code>	Disassembles IL/metadata.
<code>sn.exe</code>	Generates strong-name key pairs.
<code>gacutil.exe</code>	Manages the GAC.
<code>pverify</code>	Validates IL for type safety.

11. MSDN References

- [Assemblies in .NET](#)
- [Strong-Named Assemblies](#)
- [Global Assembly Cache](#)

Execution Process of .NET Framework vs .NET Core Executables

(A Comprehensive Analysis from Startup to Runtime Execution)

1. Historical Context

1.1 .NET Framework (2002-Present)

- Original Windows-only runtime with **tight OS integration** (registry, GAC).
- Uses **CLR 2.0-4.x** with JIT compilation.
- Deployment: Requires framework installation on target machines.

2. .NET Framework Execution Process

2.1 Startup Sequence

1. Windows Loader

- Reads PE header of `.exe` and loads `mscoree.dll` (CLR shim).

2. CLR Bootstrap

- `mscoree.dll` loads the appropriate CLR version (via registry: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework`). The CLR implementation is in `clr.dll` (before .Net 4.0 it was named as `mscorwks.dll` or `mscorsvr.dll`).

3. CLR Initialization

- Creates **AppDomain**, loads `mscorlib.dll`, and prepares JIT compiler.

4. Assembly Loading

- Resolves dependencies from GAC or app directory (Fusion Log for troubleshooting).

5. JIT Compilation

- Converts IL to native code method-by-method.

6. Execution

- Runs compiled native code with services like GC, exception handling.

2.2 Key Components

Component	Role
<code>mscoree.dll</code>	CLR shim that selects runtime version.
<code>mscorlib.dll</code>	Core library (primitive types, GC, threading).
Fusion Engine	Resolves assembly dependencies (GAC, probing paths).
JIT Compiler	Converts IL to native code (standard or optimized).

2.3 Memory Model

- **Single AppDomain per Process:** No isolation by default.
- **Heap Management:** Generational GC with workstation/server modes.

3. MSDN References

- .NET Framework Architecture
 - CLR Internals
-

IL (Intermediate Language)

The Bridge Between C# and Machine Code

1. What is IL?

Definition:

- **IL (Intermediate Language)**, also called **MSIL (Microsoft IL)** or **CIL (Common IL)**, is a **low-level, platform-agnostic** instruction set generated by .NET compilers.
- Acts as the **output of C#/VB.NET compilation** and the **input to the JIT compiler**.

Key Characteristics:

- ✓ **Stack-based** (operations push/pop values from a virtual stack).
 - ✓ **Object-oriented** (supports classes, inheritance, interfaces).
 - ✓ **Self-describing** (includes metadata for types/methods).
-

2. IL Compilation Pipeline

1. **C# Code** → Written by developers (`Program.cs`).
2. **Compiler** → Generates **IL + Metadata** (`Program.dll`).
3. **JIT Compiler** → Converts IL to **native machine code** at runtime.

```
[C# Code] --> [C# Compiler] --> [IL + Metadata] --> [JIT Compiler] --> [Native Code]
```

3. Inspecting IL: Tools

3.1 ildasm.exe (Disassembler)

- Ships with Visual Studio.
- Command:

```
ildasm MyApp.exe
```

- Shows: **Types, methods, IL instructions, metadata.**

3.2 dotnet-ilanalyzer (Modern Alternative)

```
dotnet tool install -g dotnet-ilanalyzer  
ilanalyzer MyApp.dll
```

3.3 SharpLab.io (Online Viewer)

- Live C#-to-IL conversion: sharplab.io.

4. IL Instruction Set (Key Opcodes)

Category	Example Opcodes	Description
Load/Store	ldarg.0, stloc	Load arguments/store locals.
Arithmetic	add, sub, mul	Basic math operations.
Branching	br, beq, bgt	Conditional/unconditional jumps.

Category	Example Opcodes	Description
Object Ops	<code>newobj</code> , <code>callvirt</code>	Instantiation/method calls.
Metadata	<code>ldtoken</code> , <code>castclass</code>	Type manipulation.

5. IL vs. Assembly vs. Bytecode

Aspect	IL (C#)	Assembly (x86)	Java Bytecode
Abstraction	High (OOP-aware)	Low (CPU-specific)	High (JVM-based)
Platform	Cross-platform	CPU-specific	Cross-platform
Execution	JIT-compiled	Direct execution	JIT/AOT (JVM)

6. MSDN References

- [IL Instruction Set](#)
- [Metadata Standards](#)

Visual Studio Overview

- *The Ultimate IDE for Building .NET Applications*

1. Introduction to Visual Studio

1.1 What is Visual Studio?

- Primary IDE for .NET development, developed by Microsoft.
- Supports C#, F#, VB.NET, C++, and cross-platform technologies (**ASP.NET Core, Xamarin, Unity**).
- Available in **Community (free), Professional, and Enterprise** editions.

1.2 Key Features

- ✓ **Code Editor** (IntelliSense, Refactoring, Debugging)
 - ✓ **Project Templates** (Console, Web, Mobile, Cloud)
 - ✓ **Integrated Debugger** (Breakpoints, Step-through, Profiling)
 - ✓ **Extensions Ecosystem** (ReSharper, GitHub Copilot)
 - ✓ **Azure & DevOps Integration**
-

2. Visual Studio Editions Comparison

Edition	Target Users	Key Features
Community	Students, OSS Developers	Free, Full .NET Support
Professional	Small Teams, Freelancers	CodeLens, Azure Credits
Enterprise	Large Enterprises	Advanced Debugging, Live Unit Testing

Note: All editions support **.NET 6/7/8, ASP.NET Core, and Xamarin.**

3. Installation & Setup

3.1 Download

- [Visual Studio 2022](#) (Latest stable)

3.2 Workloads for .NET Development

Workload	Purpose
.NET Desktop	WPF, WinForms
ASP.NET & Web	Blazor, MVC, Web API

Workload	Purpose
Mobile (Xamarin)	iOS/Android Apps
Azure	Cloud Development

4. Key Components for .NET Developers

4.1 Solution Explorer

- Manages **projects, dependencies, and files**.
- Supports **multi-project solutions** (e.g., `MyApp.sln`).

4.2 Code Editor (IntelliSense & AI)

- **IntelliSense** (Code completion, parameter hints).
- **GitHub Copilot** (AI-powered suggestions).
- **Refactoring** (Rename, Extract Method, etc.).

4.3 Debugging Tools

- ✓ **Breakpoints & Step Debugging**
- ✓ **Watch & Immediate Window**
- ✓ **Performance Profiler** (CPU, Memory Usage)

4.4 NuGet Package Manager

- Install/update libraries (e.g., `Newtonsoft.Json`).

4.5 Integrated Terminal & Git

- **PowerShell, CMD, WSL** support.
- **Git GUI** (Commit, Push, Branch Management).

5. Project Templates for .NET

Template	Description
Console App	CLI Applications
Class Library	Reusable .dll
ASP.NET Core	Web API, MVC, Razor Pages
WPF/WinForms	Desktop GUI Apps
Xamarin.Forms	Cross-platform Mobile

6. Extensions for Productivity

Extension	Purpose
ReSharper	Advanced Refactoring
GitHub Copilot	AI Code Suggestions
EF Core Power Tools	Database Scaffolding

7. MSDN References

- [Visual Studio Docs](#)
- [Debugging in VS](#)

Introduction to OOP & C# Classes

1. What is Object-Oriented Programming (OOP)?

1.1 Core Principles (We'll Focus on These First)

Principle	Description	C# Example
Encapsulation	Bundling data + methods into a single unit (class). Hiding internal details.	private fields + public methods
Abstraction	Exposing only essential features while hiding complexity.	Interfaces, abstract classes

(We'll cover Inheritance/Polymorphism in later sessions.)

1.2 Why Use OOP?

- ✓ **Modularity:** Break code into reusable objects.
- ✓ **Maintainability:** Isolate changes to specific classes.
- ✓ **Real-World Modeling:** Objects mirror entities (e.g., Car, User).

2. Classes in C#: The Blueprint

2.1 Basic Class Structure

```
public class Car // Class declaration
{
    // Fields (data)
    private string _model;
    private int _currentSpeed;

    // Constructor (initialize object)
    public Car(string model)
    {
        _model = model;
        _currentSpeed = 0;
    }
}
```

```
// Method (behavior)
public void Accelerate(int speedIncrease)
{
    _currentSpeed += speedIncrease;
    Console.WriteLine($"{_model} sped up to {_currentSpeed} km/h!");
}
```

2.2 Key Components

Component	Purpose	Example
Fields	Store object state (usually private).	<code>private int _count;</code>
Properties	Controlled access to fields (get/set).	<code>public string Name { get; set; }</code>
Methods	Define behavior.	<code>public void Save() { ... }</code>
Constructor	Initialize new objects.	<code>public Car() { ... }</code>

3. Creating Objects (Instances)

3.1 Instantiation with **new**

```
Car myCar = new Car("Tesla Model 3"); // Calls constructor
myCar.Accelerate(20); // Method call
```

3.2 Memory Allocation

- Objects live on the heap.
- **myCar** is a reference to the object.

4. Encapsulation in Action

**4.1 Controlling Access with Modifiers

Modifier	Accessibility
private	Only within the same class.
public	Anywhere (no restrictions).
protected	Within class + derived classes (later).

4.2 Property vs. Public Field

```
// ✗ Avoid (no control)
public string Model;

// ☑ Preferred (encapsulation)
private string _model;
public string Model
{
    get { return _model; }
    set { if(value != null) _model = value else throw new ArgumentNullException(); }
}
```

5. Constructors Explained

5.1 Default Constructor

- Provided if no constructor is defined:

```
public class Book { } // Implicit: public Book() { }
```

5.2 Parameterized Constructors

```
public class Book
{
    public string Title { get; }
    public Book(string title) {
        this.Title = title;
    }
}
```

5.3 Constructor Chaining (`this`)*

```
public class Book
{
    public string Title { get; }
    public string Author { get; }

    public Book(string title) : this(title, "Unknown") { }

    public Book(string title, string author)
    {
        Title = title;
        Author = author;
    }
}
```

6. Best Practices for Beginners

- ✓ Favor properties over public fields.
- ✓ Keep fields **private** (expose via methods/properties).
- ✓ Use descriptive names (**Car**, not **C**).
- ✓ Start small: 1 class = 1 responsibility (SRP).

7. Example

7.1 Define a **BankAccount** Class

```
public class BankAccount
{
    private double _balance;
    public string Owner { get; }

    public BankAccount(string owner, double initialBalance)
    {
        Owner = owner;
        _balance = initialBalance;
    }

    public void Deposit(double amount) {
        _balance += amount;
    }

    public void Withdraw(double amount) {
        _balance -= amount;
    }

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

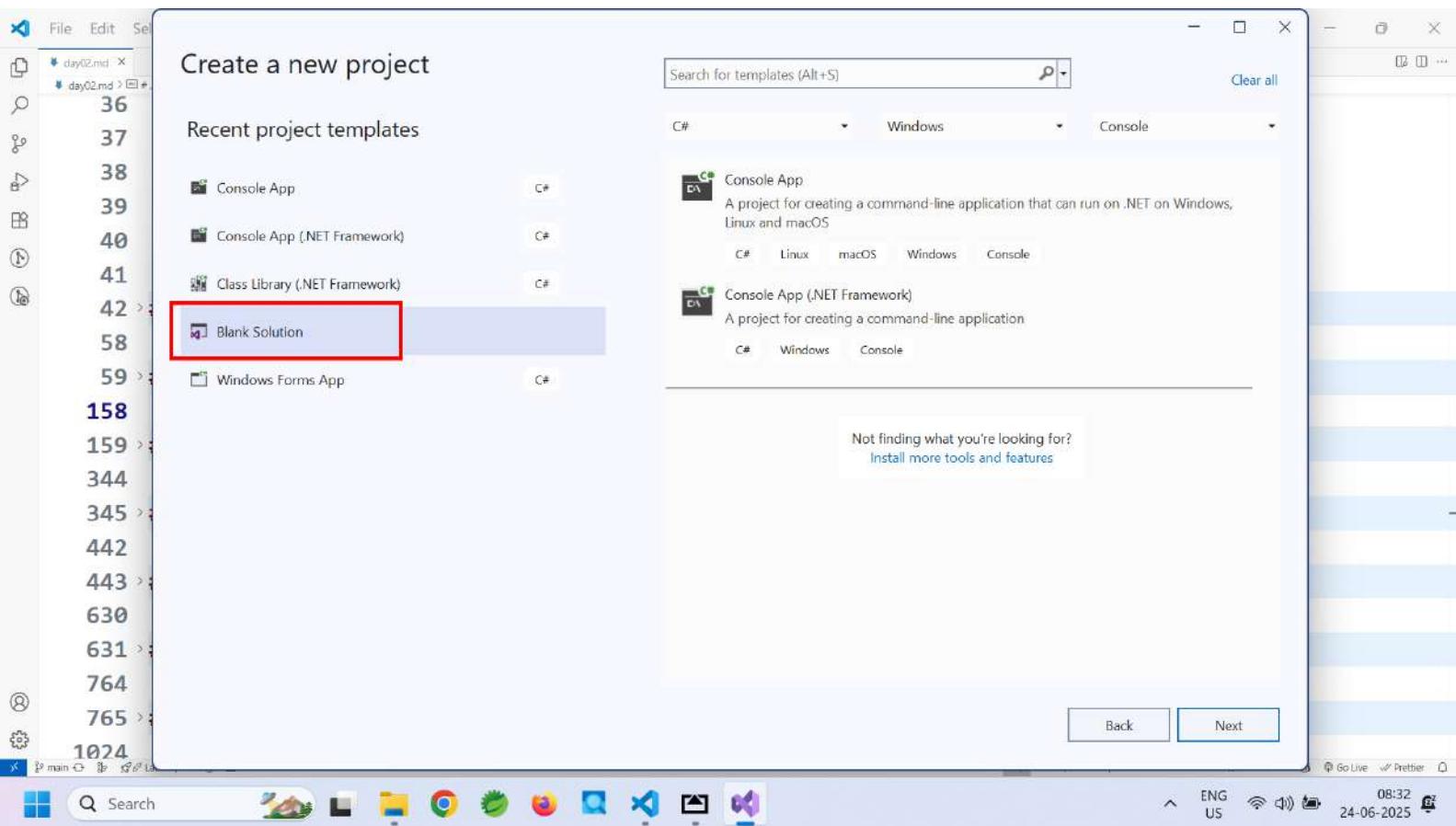
```
}
```

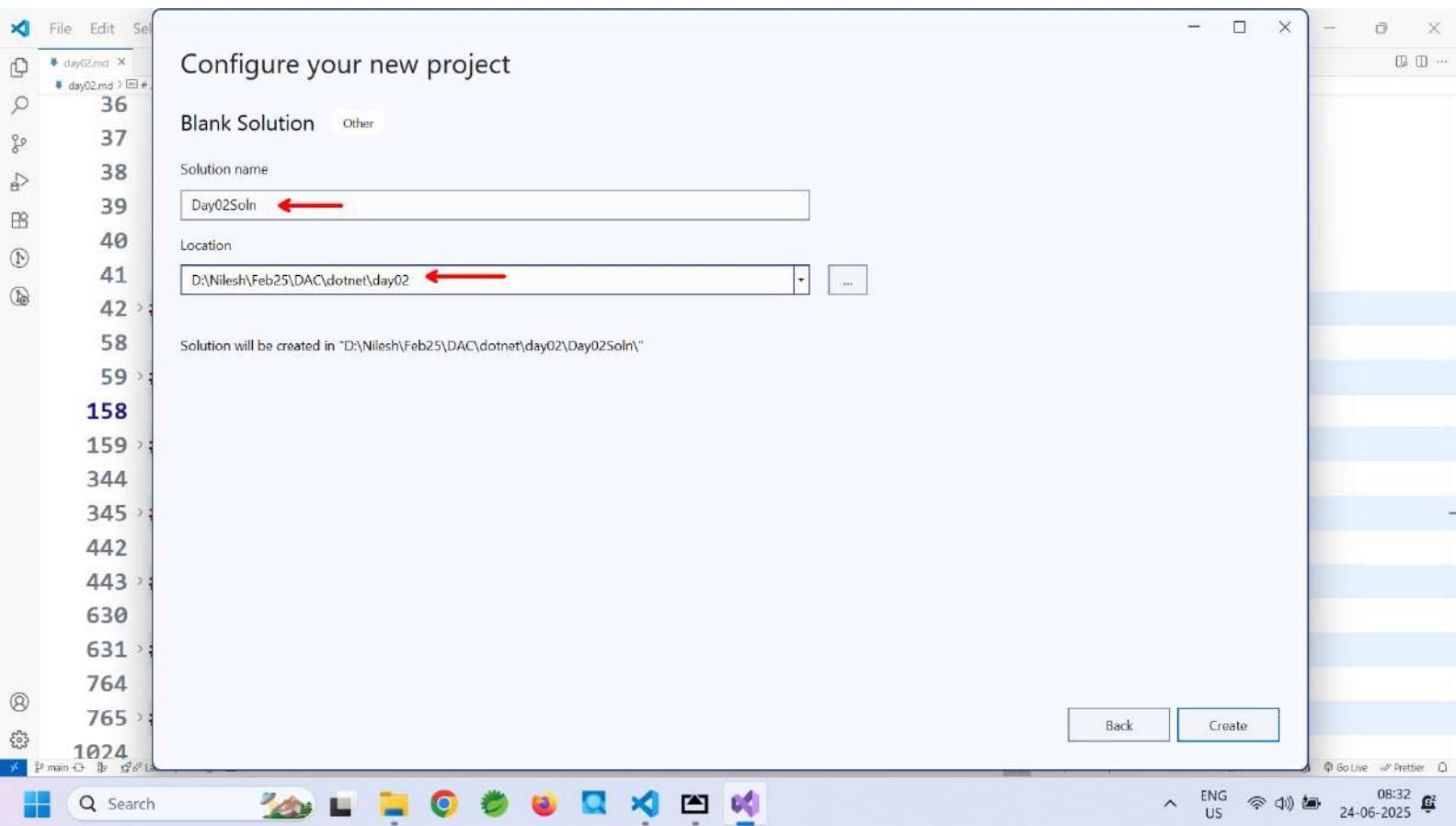
7.2 Test Your Class

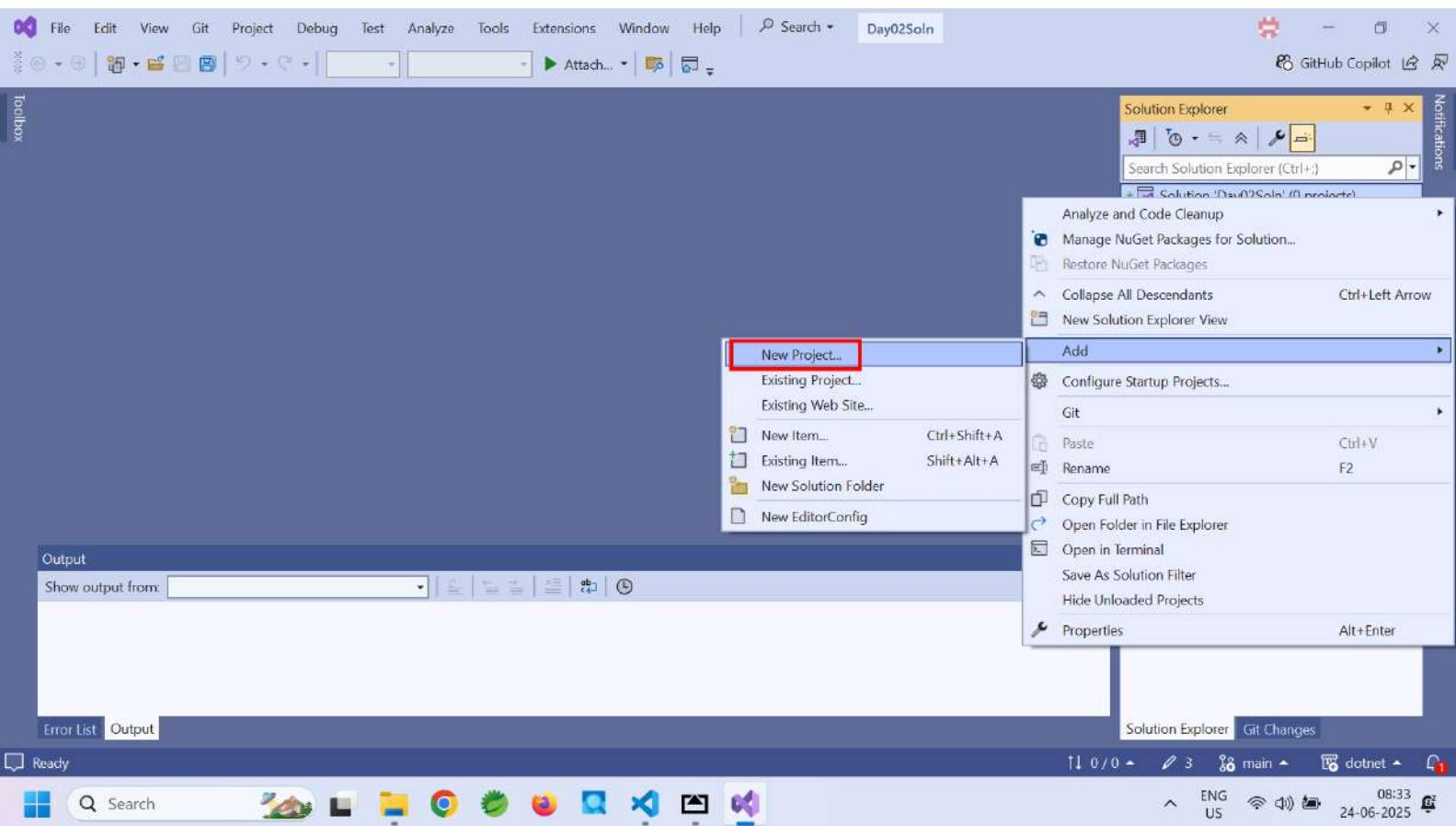
```
var account = new BankAccount("Alice", 1000);
account.Deposit(500);
Console.WriteLine($"{account.Owner}'s balance: ${account.GetBalance()}");
```

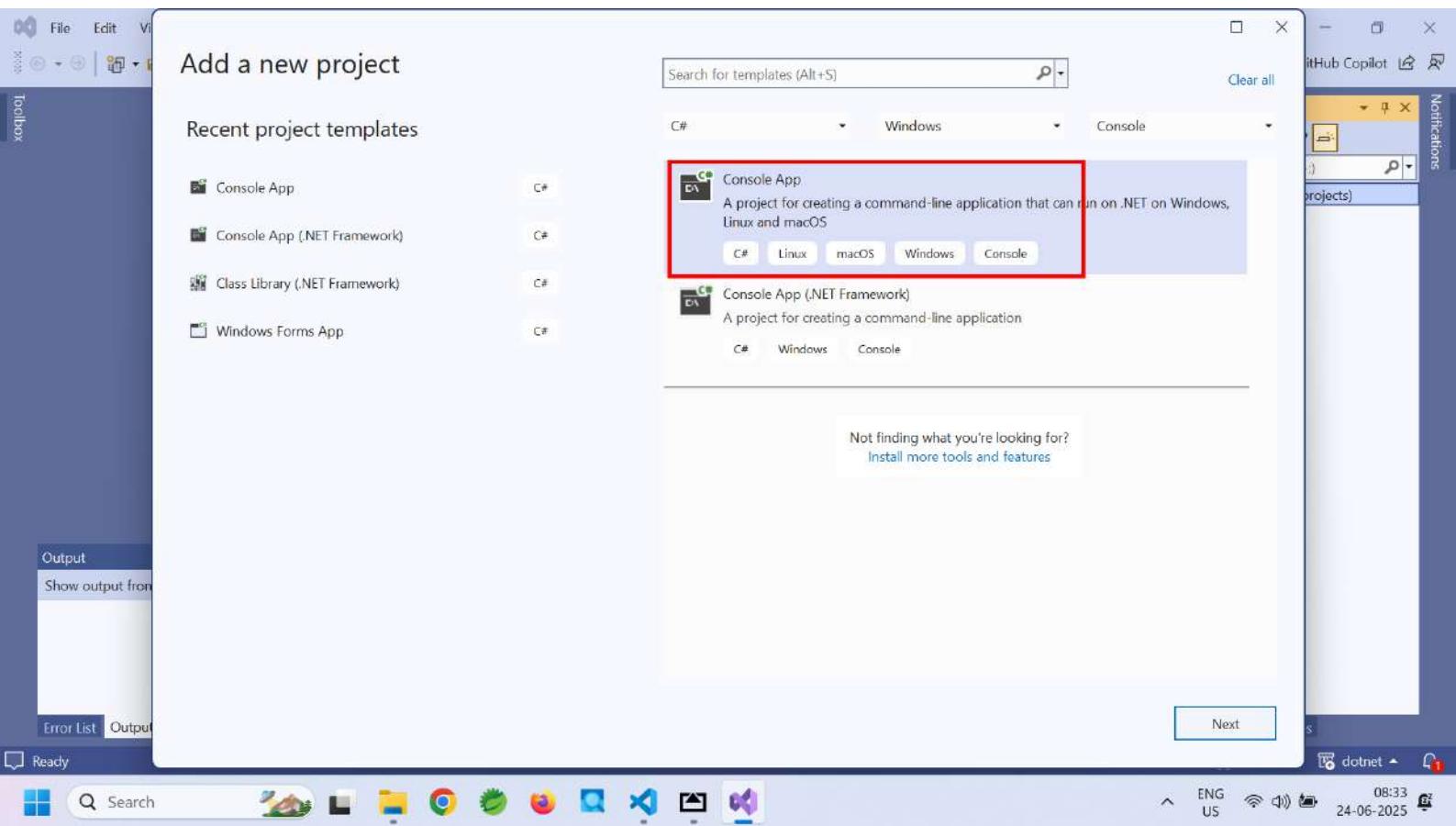
8. MSDN References

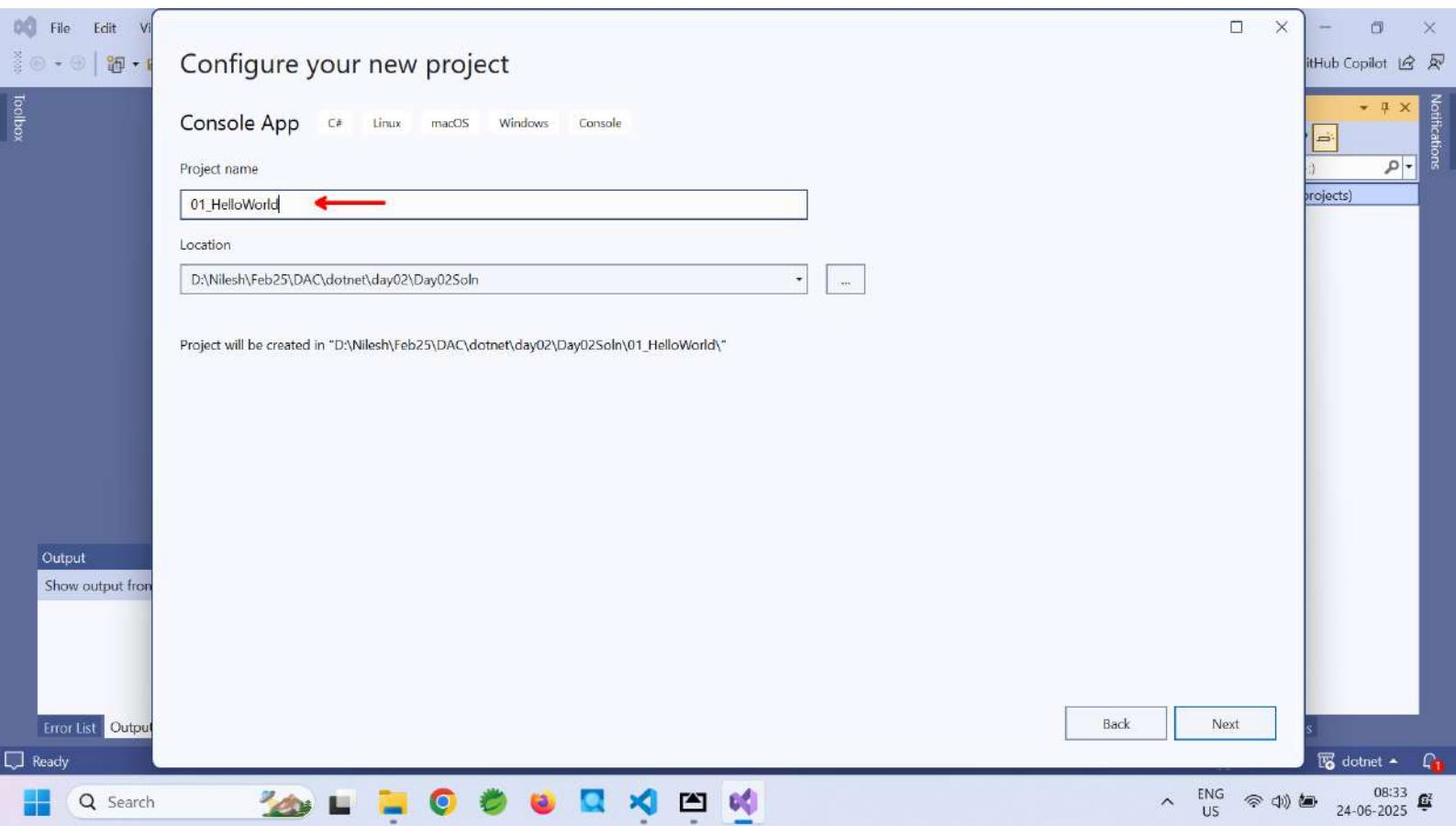
- [Classes \(C#\)](#)
 - [Properties \(C#\)](#)
-

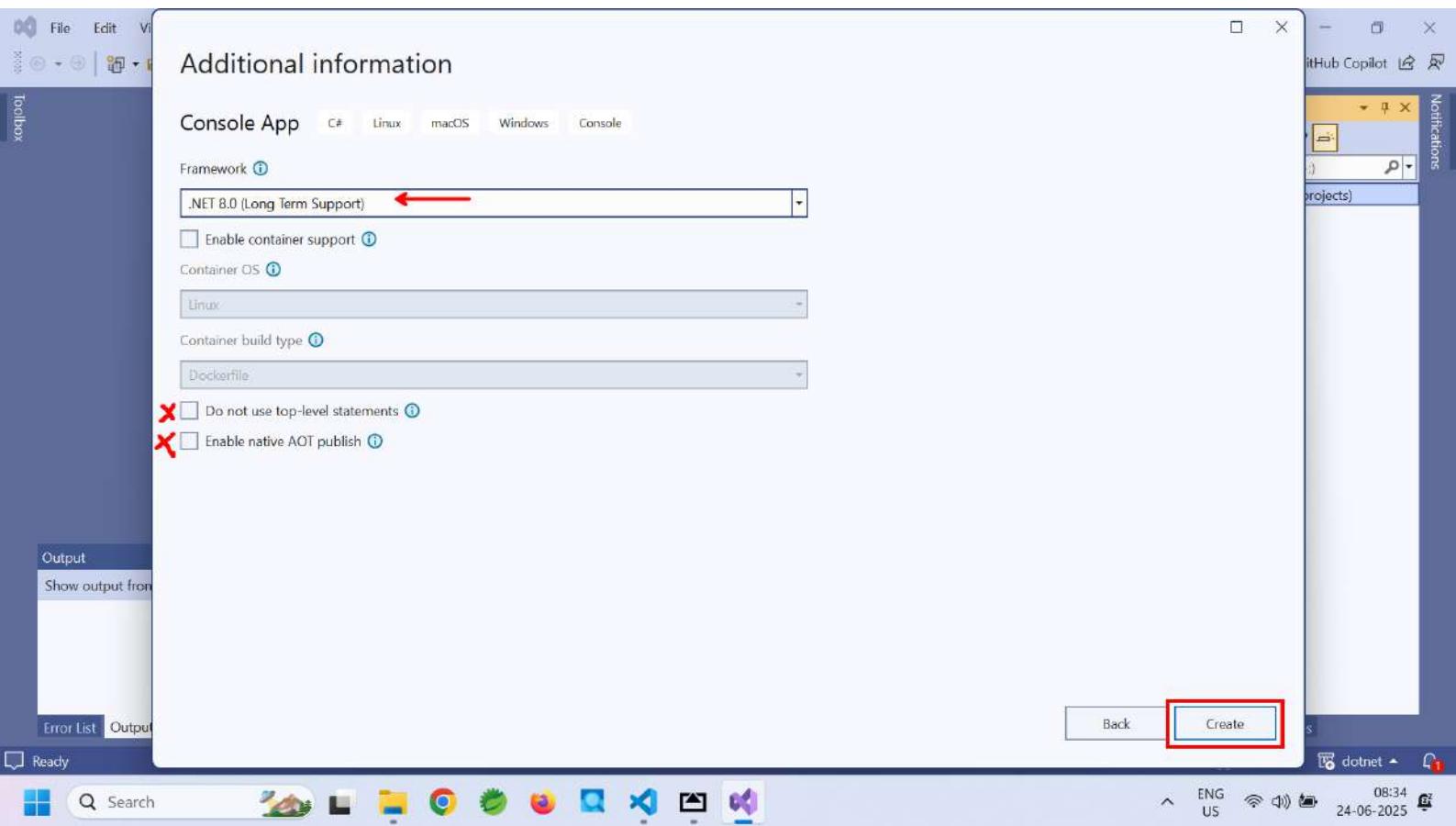












File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search Day02Soln

Program.cs

```
1 Console.WriteLine("Hello, World!");
```

Solution Explorer

Search Solution Explorer (Ctrl+.)

+ Solution 'Day02Soln' (1 of 1 project)

+ 01_Helloworld

+ Dependencies

+ C# Program.cs

Top-level statements

- Internally all these stmts are treated under a default class Main() method.

No issues found

Output

Show output from: Build

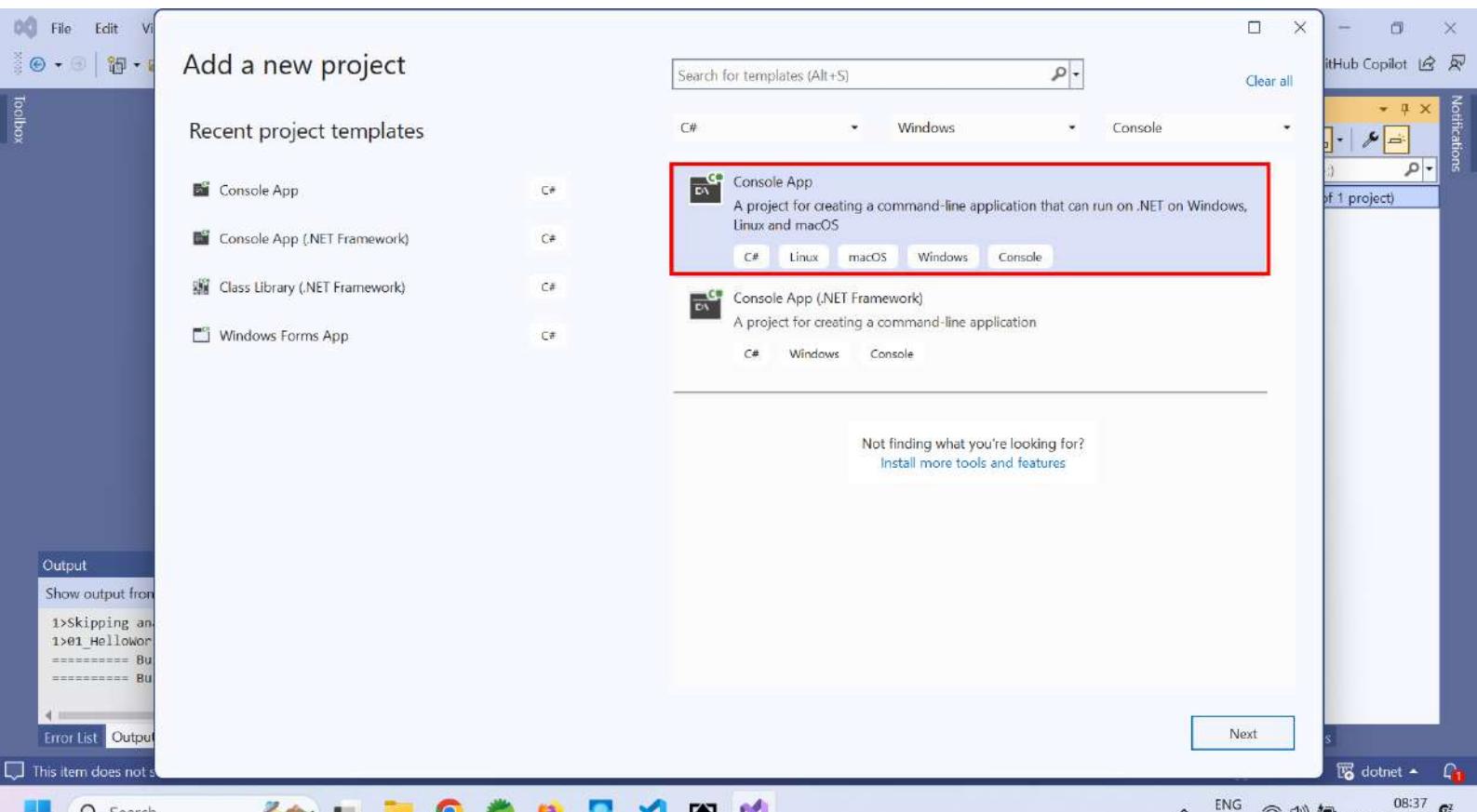
```
1> Skipping analyzers to speed up the build. You can execute 'Build' or 'Rebuild' command to run analyzers.
1> 01_Helloworld -> D:\Nilesh\Feb25\PAC\dotnet\day02\Day02Soln\01_Helloworld\bin\Debug\net8.0\01_Helloworld.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======
===== Build completed at 08:35 and took 02.008 seconds ======
```

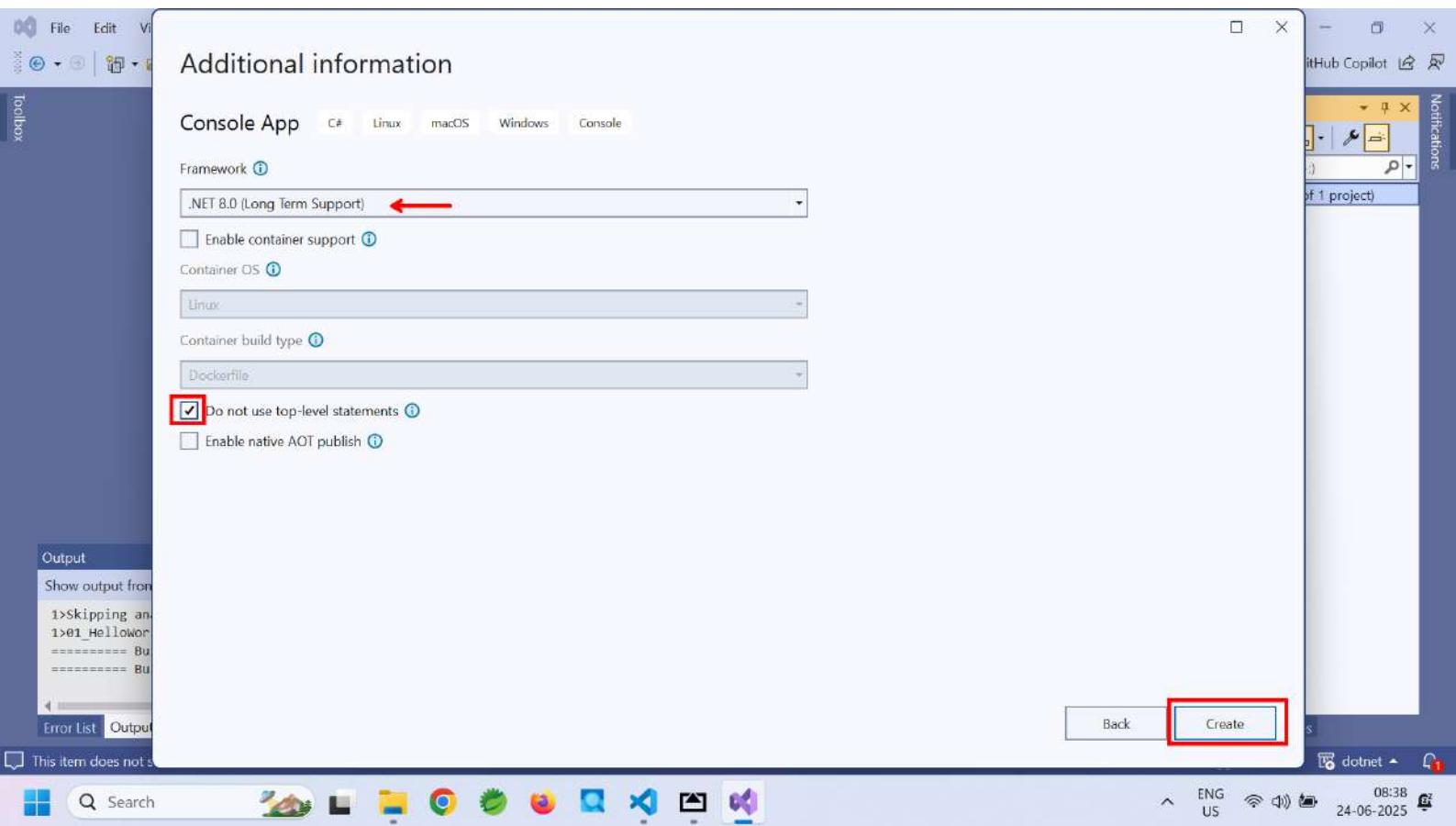
Error List Output

Build succeeded

Search

24-06-2025 08:35





File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day02Soln

Program.cs 02_Namespaces 02_Namespaces.Program Main(string[] args)

```
12  /*  
13   Namespaces are logical container.  
14   - Code organization -- Logical hierachial arrangement ↓  
15 */
```

System.Data.dll

Top Namespace
namespace System.Data; Nested Namespace

System.Data.SqlClient
SqlConnection
SqlCommand
...

System.Data.OleDb
OleDbConnection
OleDbCommand
...

System.Data.Odbc
OdbcConnection
OdbcCommand
...

- One namespace may span across multiple asms

Solution Explorer

Search Solution Explorer (Ctrl+.)

+ Solution 'Day02Soln' (2 of 2 projects)
 + 01_HelloWorld
 + Dependencies
 + C# Program.cs
 + 02_Namespaces
 + Dependencies
 + C# Program.cs

Typically namespaces are mapped to assemblies.

- One Asm (.dll) may have multiple namespaces.
- One Asm may have one namespace.

Error List Output

Item(s) Saved

Search

89 % No issues found

Line: 14 Ch: 59 SPC CRLF

11 0/0 10 08:39 main dotnet 24-06-2025 ENG US

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day02Soln

Program2.cs Program1.cs

02_Namespaces NewNamespace.Program2 Main(string[] args)

```
1 namespace NewNamespace;
2 // File-scoped namespace -- Reduce indentations (due to nesting)
3
4 internal class Program2
5 {
6     static void Main(string[] args)
7     {
8         Console.WriteLine("Hello, World!");
9     }
10}
11
12
```

when project has multiple classes with Main().
- Can select class to run in project settings i.e.
csc cmdline options.

Error List

Code	Description	Project	File	Line
CS0017	Program has more than one entry point defined. Compile with /main to specify the type that contains the entry point.	02_Namespaces	Program1.cs	49
CS0169	The field 'Node.data' is never used	02_Namespaces	Program1.cs	7
CS0169	The field 'Node.data' is never used	02_Namespaces	Program1.cs	28

Solution Explorer GitHub Copilot Notifications

Search Solution Explorer (Ctrl+.)

- + Solution 'Day02Soln' (2 of 2 projects)
 - + 01_HelloWorld
 - Dependencies
 - + C# Program.cs
 - + 02_Namespaces
 - Dependencies
 - + C# Program1.cs
 - + C# Program2.cs

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays a C# program named Program.cs. The code demonstrates two ways to handle arithmetic overflow:

```
29 static void Main(string[] args)
30 {
31     unchecked // default behaviour
32     {
33         // short range: -32768 to +32767
34         short num1 = 32767;
35         num1++; // overflow --> roll to opposite sign
36         Console.WriteLine($"num1 = {num1}");
37     }
38
39     checked // check for arithmetic overflow (data type range)
40     {
41         // and throw exception if overflow ←
42         short num2 = 32767;
43         num2++; // System.OverflowException: Arithmetic operation resu
44         Console.WriteLine($"num2 = {num2}");
45     }
46 }
47 }
```

A red annotation at the bottom left of the code block states: "In C# all exceptions are unchecked exceptions (compared to java)." A blue annotation below it states: "'checked'" & "'unchecked'" keyword are not exception types - they are for overflow check.

The Solution Explorer on the right shows the project structure:

- Solution 'Day02Soln' (4 of 4)
 - 01_HelloWorld
 - 02_Namespaces
 - Dependencies
 - Program1.cs
 - Program2.cs
 - 03_DataTypes
 - Dependencies
 - Program.cs
 - 04_StructEnum
 - Dependencies
 - Program.cs

10:18
24-06-2025

The screenshot shows a Microsoft Visual Studio interface with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Solution Explorer:** Shows a solution named "04_StructEnum" with projects "04_StructEnum" and "04_StructEnum.Program".
- Toolbox:** Standard Visual Studio toolbox.
- Task List:** Notifications and Diagnostic Tools.
- Code Editor:** Program.cs file containing the following code:

```
38 class Person
39 {
40     // convention - private fields are prefixed by _
41     private int _age;
42     private string _name;
43
44     // constructors -- homework
45     public int Age
46     {
47         get { return _age; }
48         set {
49             if (value < 0)
50                 throw new Exception($"Invalid Age: {value}");
51             _age = value;
52         }
53     }
54 }
55 internal class Program
```
- Annotations:** A red arrow points from the explanatory text below to the setter code in the Person class.
- Annotations Text:**
 - Why Properties?**
 - 1. Controlled Access:** programmer may provide getter and/or setter for a field. Writing only getter makes a read-only property.
 - 2. Data Validation:** programmer can check the value to set (in setter) before assigning to the field and throw exception if invalid value.
- Status Bar:** Shows "Ln: 76 Ch: 36 SPC CRLF".
- Taskbar:** Includes icons for Search, File Explorer, Task View, and others.
- System Tray:** Shows battery level, signal strength, and date/time (24-06-2025).

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day02Soln

Program.cs | Program.cs | Program.cs

04_StructEnum | 04_StructEnum | Main(string[] args)

```

80 Point p2 = new Point(2, 3);
81 Console.WriteLine($"p2: {{p2.X}, {p2.Y}}");
82
83 Point p3 = new Point();
84 p3.X = 5;
85 p3.Y = 8;
86 Console.WriteLine($"p3: {{p3.X}, {p3.Y}}");
87 #endregion
88
89 #region Class and Properties
90 Person pr1 = new Person();
91 //pr1.Age = -5; // System.Exception: Invalid Age: -5
92 pr1.Age = 5; // okay -- classic property
93 pr1.Name = "John"; // okay -- auto-implemented property
94 pr1.Display();
95 #endregion
96
97 #region Object Initializer
98 // Object can be initialized while allocating it - using properties.
99

```

Stack

Heap

p3

X	Y
5	8

Point obj

pr1

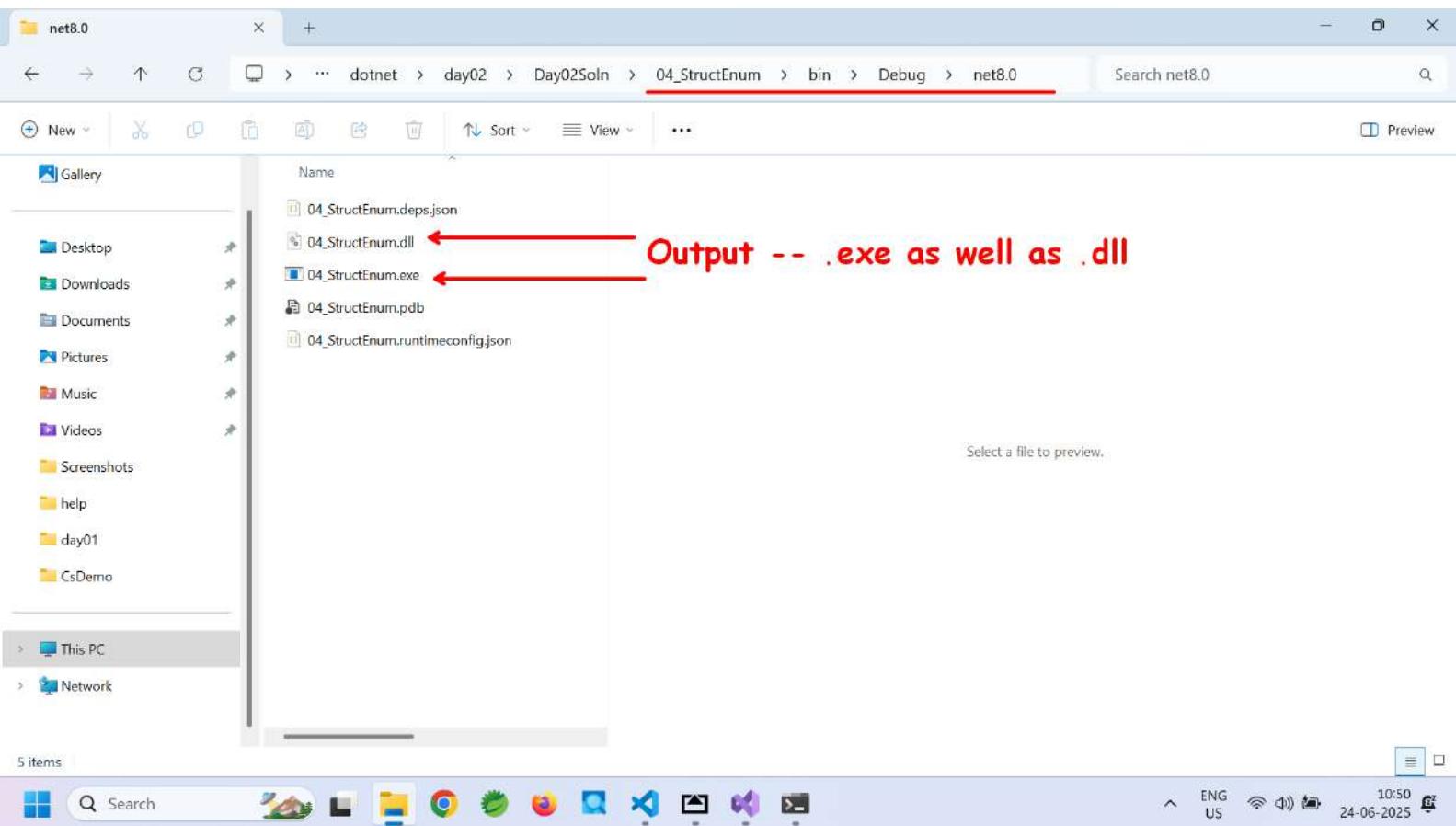
1000	
------	--

Person obj

5	*
_age	_name

"John"





The screenshot shows a Microsoft Visual Studio interface with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Search Bar:** Search ...
- Solution Explorer:** Shows a solution named "Day02Soln" with multiple projects and files.
- Toolbox:** Standard .NET components.
- Code Editor:** Program.cs file containing the following code:

```
90  static void Main1(string[] args){...}
106 static void Main2(string[] args){...}
132 static void Main(string[] args)
133 {
134     Weekday d1 = new Weekday(); // def
135     Console.WriteLine("d1 = " + d1);
136
137     Weekday d2 = Weekday.Wednesday;
138     Console.WriteLine("d2 = " + d2 + " ");
139
140     Weekday d3 = (Weekday)6;
141     Console.WriteLine("d3 = " + d3 + " ");
142
143     Weekday d4 = (Weekday)10;
144     Console.WriteLine("d4 = " + d4 + " ");
145 }
146
147 }
```

- Output Window:** Microsoft Visual Studio Debug window showing the execution results:

```
d1 = Monday
d2 = Wednesday -- ordinal = 2
d3 = Sunday -- ordinal = 6
d4 = 10 -- ordinal = 10

D:\Nilesh\Feb25\DAC\dotnet\day02\Day02Soln\04_StructEnum\bin\Debug\net8.0\04_StructEnum.exe (process 272
36) exited with code 0 (0x0).
Press any key to close this window . . .
```

- Status Bar:** Build succeeded.
- Taskbar:** Windows Start button, Search bar, Task View, File Explorer, Google Chrome, Mozilla Firefox, Microsoft Edge, Visual Studio, and others.
- System Tray:** ENG US, Wi-Fi, Battery, Date (24-06-2025), and a notification icon.

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day02Soln

Program.cs | Program2.cs | Program.cs | 05_Boxing | 05_Boxing.Program | Main(string[] args)

```

internal class Program
{
    static void Main(string[] args)
    {
        int num1 = 123;
        object obj = num1;
        // up-casting
        // int (value type) --> object (reference type) = Boxing
        int num2 = (int)obj;
        // down-casting
        // object (reference type) --> int (value type) = Unboxing
    }
}

```

Stack **Heap**

num1 **123**

obj **1000** → **123**

num2 **123**

boxing

unboxing

Boxing - allocates new obj on heap and copy data

Unboxing - copies data from heap to var

object

Boxing/Unboxing

- Needs more memory
- Needs time (copy)

C# Inheritance -- is-a

```
class Derived : Base {  
    ...  
}
```

```
1 Xyz company -- Salary Processing System  
2 ======  
3 1. Labors -- salary = payment hours * rate  
4 2. Managers -- salary + bonus  
5 3. Salesmans -- salary + comm  
6 4. Many employees -- Get total sal expense  
7  
8 class Person {  
9     // fields: name, age, id  
10    // methods: ctors, getter/setter, accept(), display()  
11 }  
12  
13 class Emp : Person {  
14     // fields: salary,  
15     // methods: ctors, getter/setter, accept(), display(), calcSal()  
16 }  
17  
18 class Labor : Emp {  
19     // fields: hours, rate  
20     // methods: ctors, getter/setter, accept(), display(), calcSal()
```

```
class Base {  
    public void Method1() {  
        // ...  
    }  
  
    public virtual void Method2() {  
        // ...  
    }  
  
    public virtual void Method3() {  
        // ...  
    }  
}  
  
class MyClass: Base {  
    public override void Method3() {  
        // ...  
    }  
}
```

```
class Derived : Base {  
    public new void Method1() {  
        // ...  
    }  
  
    public override void Method2() {  
        // ...  
    }  
  
    public new virtual void Method3() {  
        // ...  
    }  
}  
  
class Your:Derived {  
    public override void  
        Method3() { // ...  
    }  
}
```

```
Base b2=new MyClass()  
b2.Method3();//My  
Class  
  
Base b3=new Your();  
b3.Method3();//Base  
Class  
  
Derived b4=new Your()  
b4.Method3();//Your  
Class  
  
b4.Method2();  
//Derived  
  
Base b1=new Derived()  
b1.Method1(); //Base  
Class  
b1.Method2(); //Derive  
Class  
b1.Method3(); //Base  
Class
```

.NET

.NET Versions

- ** .NET Framework:** The original .NET platform, primarily for Windows desktop and web applications. It is no longer actively developed with new feature releases, but 4.8 is the last major version and continues to receive maintenance updates.
- ** .NET Core:** A cross-platform, open-source, and high-performance framework that was a complete re-architecture. It was designed for modern, cloud-native applications.
- ** .NET:** Starting with .NET 5.0, Microsoft unified .NET Core and the best of .NET Framework into a single, cross-platform product simply called ".NET". There is no .NET 4.x (after .NET Framework 4.8) or .NET Core 4.x. The numbering continued from .NET Core 3.1 to .NET 5.0.
- ** .NET Standard:** A formal specification of .NET APIs that are available on all .NET implementations (like .NET Framework, .NET Core, Xamarin, etc.). It enables developers to build libraries that can be used across different .NET platforms. It's not a runtime itself, but a contract. .NET Standard 2.1 is the last version, as the need for it diminished with the unification of .NET.

Release	.NET Framework	.NET Core	.NET	.NET Standard
Feb 2002	1.0			
Apr 2003	1.1			
Nov 2005	2.0			
Nov 2006	3.0			
Nov 2007	3.5			
Apr 2010	4.0			
Aug 2012	4.5		1.0, 1.1	
Oct 2013	4.5.1			
May 2014	4.5.2		1.2	

Release	.NET Framework	.NET Core	.NET	.NET Standard
Jul 2015	4.6			
Nov 2015	4.6.1		1.3	
Jun 2016		1.0		1.4, 1.5
Aug 2016	4.6.2		1.6	
Nov 2016		1.1		
May 2017	4.7			
Aug 2017		2.0	2.0	
Oct 2017	4.7.1			
Apr 2018	4.7.2			
May 2018		2.1		
Dec 2018		2.2		
Apr 2019	4.8			
Sep 2019		3.0	2.1	
Dec 2019		3.1		
Nov 2020			5.0	
Nov 2021			6.0	
Nov 2022			7.0	
Nov 2023			8.0	

C# Language versions

C# Version	Release Month-Year	Corresponding .NET Platform Version(s)	Most Important New Language Features
C# 1.0	Jan 2002	.NET Framework 1.0, 1.1	The foundational C# language: Classes, Structs, Interfaces, Delegates, Events, Properties, Attributes, basic error handling (try-catch-finally).
C# 2.0	Nov 2005	.NET Framework 2.0, 3.0	Generics , Partial Classes, Anonymous Methods, Iterators (<code>yield return</code>), Nullable Types (?), Covariance/Contravariance for delegates.
C# 3.0	Nov 2007	.NET Framework 3.5	LINQ (Language Integrated Query) , Lambda Expressions , Extension Methods, Anonymous Types, Object and Collection Initializers, Implicitly Typed Local Variables (<code>var</code>), Expression Trees.
C# 4.0	Apr 2010	.NET Framework 4.0	<code>dynamic</code> Keyword, Named and Optional Arguments, Generic Covariance and Contravariance (for interfaces and delegates), Embedded Interop Types.
C# 5.0	Aug 2012	.NET Framework 4.5	Async and Await (asynchronous programming support).
C# 6.0	Jul 2015	.NET Framework 4.6, .NET Core 1.0, 1.1	String Interpolation , Null-Conditional Operator (?.), Auto-Property Initializers, Expression-Bodied Members, <code>using static</code> , <code>nameof</code> operator, Exception Filters.
C# 7.0	Mar 2017	.NET Framework 4.7, .NET Core 2.0, 2.1, 2.2	Pattern Matching (<code>is</code> expression, <code>switch</code> statement enhancements), Tuples and Deconstruction , Local Functions, <code>out</code> variables, <code>ref</code> locals and returns, <code>throw</code> expressions.
C# 8.0	Sep 2019	.NET Core 3.0, 3.1	Nullable Reference Types , Default Interface Methods , Asynchronous Streams (<code>IAsyncEnumerable</code>), Indices and Ranges (^, ..), <code>using</code> declarations, Switch Expressions, Property Patterns, Positional Patterns.
C# 9.0	Nov 2020	.NET 5.0	Records , Top-level Statements , Pattern Matching Enhancements (type patterns, relational patterns, logical patterns: <code>and</code> , <code>or</code> , <code>not</code>), <code>init</code> only setters, Target-typed <code>new</code> expressions.
C# 10.0	Nov 2021	.NET 6.0	File-Scope Namespaces , Global using directives , Record structs, <code>const</code> interpolated strings, Enhanced Lambda Expressions, <code>With</code> expressions for structs.
C# 11.0	Nov 2022	.NET 7.0	Raw String Literals , Generic Attributes, List Patterns, <code>required</code> members, <code>static abstract</code> members in interfaces (for generic math), <code>file</code> local types.

C# Version	Release Month-Year	Corresponding .NET Platform Version(s)	Most Important New Language Features
C# 12.0	Nov 2023	.NET 8.0	Primary Constructors (for non-record types), Collection Expressions ([] for collections), Alias any type (<code>using</code> aliases for <code>any</code> type), <code>ref readonly</code> parameters, Inline arrays, Interceptors (experimental).

Namespaces in C# .NET

Introduction

Namespaces were introduced in the very first version of C# (2002) as a fundamental organizational feature. They were designed to solve the problem of naming collisions in large projects and to provide a logical hierarchy for organizing related functionality. The concept was borrowed from C++ but implemented with more rigor and better integration with the .NET type system.

Definition and Purpose

- Logical Organization:** Namespaces provide a way to organize related classes, structs, interfaces, enums and delegates into logical groups. This organization mirrors the way we organize files into folders in a file system.
- Name Collision Prevention:** They prevent naming conflicts by allowing two classes with the same name to coexist as long as they are in different namespaces. For example, both `System.IO.File` and `MyApp.IO.File` can exist simultaneously.
- Scope Delimitation:** Namespaces define a declarative region that helps limit the scope of names declared within them. This affects visibility and accessibility of types.

Namespace Hierarchy and Structure

- Dot Notation Hierarchy:** Namespaces use dot notation to create hierarchies (e.g., `System.Collections.Generic`). This doesn't imply inheritance - it's purely organizational.
- Recommended Conventions:** Microsoft guidelines suggest:
 - Starting with company name (e.g., `Microsoft`)

- Then product name (e.g., Office)
- Then functional area (e.g., Interop)

3. Physical vs Logical Organization: While namespaces suggest a physical organization, they don't enforce it. A single assembly can contain multiple namespaces, and a single namespace can span multiple assemblies.

Relationship with Assemblies

1. Many-to-Many Relationship

There's no strict 1:1 relationship between namespaces and assemblies. The System.Data.dll assembly contains multiple namespaces (System.Data, System.Data.Common, System.Data.SqlClient etc.)

2. Common Patterns

- Core functionality in root namespace (System.Data)
- Specialized functionality in child namespaces (System.Data.SqlClient)

3. Assembly Naming Conventions

Assemblies often take the name of their primary namespace (System.Data.dll for System.Data), but this isn't mandatory.

Using Namespaces in Code

1. The 'using' Directive

The 'using' keyword serves two purposes:

- As a directive to import namespaces
- As a statement for resource cleanup

2. Aliasing

Allows resolving naming conflicts or shortening long namespaces:

```
using Excel = Microsoft.Office.Interop.Excel;
```

3. Global Using (C# 10)

New in C# 10, global using directives apply to the entire project.

```
// GlobalUsings.cs
global using System;
global using System.IO;

// All other .cs files in the project
// No need to include using System; or using System.IO;
```

Namespace Design Guidelines

- 1. Logical Grouping:** Group types that are commonly used together and change together.
- 2. Depth Consideration:** While deep hierarchies are possible, 2-4 levels are typically most usable.
- 3. Versioning Considerations:** Namespaces should remain stable across versions to avoid breaking changes.

Advanced Namespace Features

- 1. File-Spaced Namespaces (C# 10):** Reduces indentation by declaring namespace scope for the entire file.

```
namespace MyNamespace;

class MyClass {
```

```
// ... code ...  
}
```

2. Implicit Global Usings: .NET 6+ projects can automatically include common namespaces.

- Frequently used namespaces like System, System.Collections.Generic, and System.Linq are implicitly imported.
- You can now use types from System.Collections.Generic (like IEnumerable) without writing `using System.Collections.Generic;` at the top of your file.

MSDN References

- [Namespaces \(C# Programming Guide\)](#)
- [Global Using Directive](#)

Data Types

1. Overview of Data Types C#

1.1 Categories of Data Types

Category	Description	Examples
Value Types	Stored on the stack (direct value).	<code>int, float, struct, enum</code>
Reference Types	Stored on the heap (accessed via reference).	<code>class, string, delegate, array</code>

1.2 Key Differences

Aspect	Value Types	Reference Types
Memory Location	Stack	Heap
Assignment	Copy entire value	Copy reference (pointer)

Aspect	Value Types	Reference Types
Default Value	0, false, etc.	null
Performance	Faster access	Slightly slower (indirection)

2. Value Types

2.1 C# - Built-in Primitive Types

Type	Size (Bytes)	Range	.NET Alias
byte	1	0 to 255	System.Byte
short	2	-32,768 to 32,767	System.Int16
int	4	-2.1B to 2.1B	System.Int32
long	8	-9.2Q to 9.2Q	System.Int64
float	4	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	System.Single
double	8	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	System.Double
decimal	16	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	System.Decimal
bool	1	true or false	System.Boolean
char	2	Unicode (U+0000 to U+FFFF)	System.Char

2.2 Type Conversions

1. Implicit Conversion (Widening)

- **Compiler-automated** when no data loss is possible.
- **Allowed for numeric types** if the target type has a larger range.

- Examples

```
int numInt = 100;
long numLong = numInt; // Implicit (int → long)

float price = 10.5f;
double total = price; // Implicit (float → double)
```

2. Supported Implicit Conversions

From	To
byte	short, int, long, float, double
int	long, float, double
char	int, long, float, double

3. Explicit Conversions (Narrowing)

- **Manual** using `(type)` syntax.
- **May cause data loss** (e.g., truncation, overflow).
- **3.2 Examples**

```
double pi = 3.14159;
int intPi = (int)pi; // Explicit (double → int), truncates to 3

long bigNum = 1_000_000_000;
int smallerNum = (int)bigNum; // Works if within int range
```

2.3 Conversion Methods

1. Convert Class

- Handles `null`, special cases, and cultural formats.

```
string input = "123";
int number = Convert.ToInt32(input); // Throws FormatException if invalid
```

2. Parse vs. TryParse

Method	**Behavior**	**Usage**
<code>`Parse`</code>	Throws exception on failure	<code>int.Parse("42")</code>
<code>`TryParse`</code>	Returns <code>bool</code> (safe for user input)	<code>int.TryParse("42", out var result)</code>

- Example

```
if (int.TryParse(Console.ReadLine(), out int age))
{
    Console.WriteLine($"Age: {age}");
}
else
{
    Console.WriteLine("Invalid input!");
}
```

2.4 Handling Overflow

- Use `checked` to throw exceptions on overflow.

```
checked
{
    int max = int.MaxValue;
    int overflow = (int)(max + 1); // throws OverflowException
}
```

- The default behavior is `unchecked` i.e. data overflows to opposite sign (positive to negative).

2.5 User-defined Value Types (`struct` and `enum`)

```
public struct Point
{
    public int X;
    public int Y;
}
```

```
public enum LogLevel { Info, Warning, Error }
// by default, underlying type is "int"
```

3. Reference Types

3.1 Built-in Reference Types

Type	Description	Example
<code>string</code>	Immutable Unicode text	<code>string s = "Hello";</code>

Type	Description	Example
object	Base type for all .NET objects	<code>object o = 10;</code>
Array	Fixed-size collection	<code>int[] nums = { 1, 2, 3 };</code>
Delegate	Function pointer	<code>Action<int> print = Console.WriteLine;</code>

3.2 Custom Reference Types (`class` and `interface`)

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

✓ **Use cases:** Complex objects with behavior.

4. Special Types

4.1 `System.Object` (Root Type)

- All types inherit from `object`.
- Methods:

```
Equals(), GetHashCode(), ToString(), GetType()
```

4.2 `System.ValueType` (Base for Value Types)

- Overrides `Equals()` and `GetHashCode()` for value comparison.

4.3 `System.Void` (For Methods with No Return)

- Used in reflection (`MethodInfo.ReturnType`).

5. Best Practices

- ✓ Prefer `int` over `Int32` (language aliases are idiomatic).
- ✓ Use `struct` for small, immutable data.
- ✓ Avoid large value types (copy overhead).

6. MSDN References

- Built-In Types (C#)
- Value Types vs. Reference Types

Boxing and Unboxing

Introduction

Boxing and unboxing have been fundamental concepts in C# since its initial release (2002) as part of the .NET Framework's type system. These mechanisms were designed to bridge the gap between value types and reference types, enabling unified type handling through the object-oriented paradigm.

Definition and Core Concepts

1. Boxing Definition

Boxing is the process of converting a value type to a reference type by wrapping the value inside a reference type like `System.Object`. This operation:

- Creates a new object on the managed heap
- Copies the value type's value into this object
- Returns a reference to the new object

```
int num1 = 123;  
object obj = num1; // boxing
```

2. Unboxing Definition

Unboxing is the reverse operation that extracts the value type from the object. This requires:

- Checking that the object instance is a boxed value of the correct type
- Copying the value from the heap back to the stack

```
int num2 = obj; // unboxing
```

- Unboxing requires exact type match:

```
object boxed = 42;  
long value = (long)boxed; // Runtime error
```

3. Type System Integration

These operations enable value types to participate in:

- Polymorphic behavior through object references
- Collections that store objects (like ArrayList)
- Reflection and late binding scenarios

Performance Implications

1. Memory Impact

- Boxing allocates memory on the heap (minimum 16 bytes for object header + value)
- Creates garbage collection pressure
- Unboxing doesn't allocate but requires type checking

2. CPU Overhead

- Boxing involves memory allocation and copying
- Unboxing requires type checking and copying
- Both operations are orders of magnitude slower than direct value type operations

3. Hidden (Implicit) Boxing

- Value types in non-generic collections
- Calling GetType() on value types
- String concatenation with value types

Common Usage Patterns

1. Legacy Collections:

Pre-generics (pre-.NET 2.0) collections like ArrayList required boxing:

```
ArrayList list = new ArrayList();
list.Add(42); // Boxing occurs
int value = (int)list[0]; // Unboxing occurs
```

2. Interface Implementation:

Value types implementing interfaces get boxed when cast to the interface type:

```
IComparable comparable = 5; // Boxing occurs
```

3. Reflection Scenarios:

Methods like MethodInfo.Invoke() box value type parameters

Modern Alternatives

1. **Generics (.NET 2.0+)**: Generic collections (List) eliminate boxing by working with value types directly
2. **Span (.NET Core 2.1+)**: Provides stack-only, boxing-free access to memory

MSDN References

- [Boxing and Unboxing \(C# Programming Guide\)](#)
 - [Performance Considerations](#)
 - [Type System Fundamentals](#)
-

C# Methods

1. Methods in C#

1.1 Definition

- A **method** is a block of code that performs a specific task and can be reused.
- Can return a value (**void** if no return).

1.2 Syntax

```
// Method with parameters and return type
public int Add(int a, int b) {
    return a + b;
}

// void method (no return)
public void Log(string message) {
    Console.WriteLine(message);
}
```

1.3 Method Components

Part	Example	Purpose
Access Modifier	public, private	Controls visibility.
Return Type	int, void	Specifies what the method returns.
Parameters	(int a, int b)	Inputs to the method.

2. Method Parameters

2.1 Named and Optional Arguments

- Named arguments enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list.
- The arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

```
void PrintInfo(string name, int age, string addr, string email) {
    Console.WriteLine($"Name: {name}, Age: {age}, Addr: {addr}, Email: {email}");
}

// positional args - args must be passed in sequence
PrintInfo("James Bond", 65, "London", "james@bond.com");

// named args - args can be passed in any order
PrintInfo(age: 65, email: "james@bond.com", name: "James Bond", addr: "London");

// mixed args - combination of args is possible - positional args work as long as their positions are correct
PrintInfo(name: "James Bond", 65, email: "james@bond.com", addr: "London");

// ERROR: Named argument 'email' is used out-of-position but is followed by an unnamed argument
PrintInfo(name: "James Bond", email: "james@bond.com", 65, addr: "London");
```

- Optional arguments enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.
- Optional parameters must appear after all required parameters (like C++).

```
void PrintInfo(string name, int age, string addr = "Anywhere", string email="Unknown")
{
    Console.WriteLine($"Name: {name}, Age: {age}, Addr: {addr}, Email: {email}");
}

// Usage
PrintInfo("James Bond", 65);
PrintInfo(age:65, name:"James Bond");
```

2.3 Parameter Types

Type	Syntax	Behavior
Value	int a	Copy of the value passed.
Reference (ref)	ref int a	Directly modifies the original variable.
Output (out)	out int result	Assigns a value before returning.
Input (in)	in Vector3 pos	Read-only reference (performance optimization).

1. **ref** Parameter

```
public void Swap(ref int x, ref int y) {
    int temp = x;
    x = y;
    y = temp;
}

// Usage:
```

```
int a = 5, b = 10;  
Swap(ref a, ref b); // a=10, b=5
```

2. **out** Parameter

```
public void Multiply(int x, int y, out int result) {  
    result = x * y;  
}  
  
// Usage:  
int num1=22, num2=7, res;  
Multiply(num1, num2, out res);
```

3. **in** Parameter

```
public double Calculate(in Vector3 point) {  
    return point.X + point.Y; // Cannot modify 'point'  
}
```

3. Local Functions (C# 7.0+)

3.1 Definition

- **Nested methods** inside another method.
- Useful for **helper logic** that shouldn't be exposed.

3.2 Syntax

```
public void ProcessData(List<int> data) {
    // Local function
    int Square(int x) {
        return x * x;
    }

    int data = {1, 2, 3, 4};
    foreach (int num in data)
        Console.WriteLine(Square(num));
}
```

- **Access outer method variables:**

```
public void PrintCount() {
    int count = 0;

    void Increment() {
        count++; // Modifies 'count'
    }
    Increment();
    Console.WriteLine(count); // 1
}
```

4. Static Local Functions (C# 8.0+)

4.1 Purpose

- **Prevents accidental captures** of outer variables (improves performance).
- Explicitly declares no dependency on enclosing scope.

4.2 Example

```
public void Calculate() {  
    int baseValue = 10;  
  
    // Static local function (cannot access 'baseValue')  
    static int Scale(int value, int factor) {  
        return value * factor;  
    }  
  
    Console.WriteLine(Scale(baseValue, 2)); // 20  
}
```

4.3 When to Use

- **Pure functions** (no side effects).
- **Performance-critical code** (avoid closure allocations).

5. Best Practices

- Use **out** for multiple returns (instead of tuples in simple cases).
- Prefer **local functions** over private methods for one-off helpers.
- Use **static locals** to avoid unintended closures.

6. MSDN References

- Methods (C#)
- Local Functions
- Parameter Modifiers

User-Defined Value Types

1. Introduction to User-Defined Value Types

1.1 Definition

- **Value types** store data directly (stack-allocated).
- **Two custom flavors:**
 - **struct**: Composite data type (e.g., `Point`, `DateTime`).
 - **enum**: Named constant set (e.g., `LogLevel`, `HttpStatusCode`).

1.2 Key Characteristics

Feature	Struct	Enum
Memory	Stack (unless boxed)	Stack (underlying integer)
Default	All fields zeroed	First member (0 by default)
Inheritance	No (sealed implicitly)	No
Use Case	Small, immutable data	Finite named options

2. Structs (Custom Value Types)

2.1 Definition & Syntax

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

    public Point(int x, int y) {
        this.X = x;
        this.Y = y;
    }
}
```

2.2 When to Use Structs

- ✓ **Small size** (< 16 bytes recommended).
- ✓ **Frequently copied** (e.g., coordinates, RGB colors).
- ✓ **Immutable patterns** (`readonly struct` have only getters for properties).

2.3 When to Avoid Structs

- ✗ **Large data** (copy overhead).
- ✗ **Polymorphism needed** (use `class` instead).

2.4 Performance Implications

- **Stack allocation** → Faster than heap (`class`).
- **No garbage collection** → Reduced GC pressure.

3. Struct vs. Class Decision Table

Criteria	Choose <code>struct</code>	Choose <code>class</code>
Size	Small (< 16B)	Large
Semantics	Value equality	Reference identity
Allocation	Stack	Heap
Mutation	Immutable preferred	Mutable

4. Advanced Struct Features

- Refer MSDN for more details.

1. Ref Structs (Stack-Only)

```
public ref struct StackOnlyStruct { ... }
```

2. **readonly struct (Immutable by Design)**

```
public readonly struct ImmutablePoint
{
    public int X { get; } // No setters allowed
}
```

3. **record struct (C# 10+)**

```
public record struct Point(double X, double Y) {
    public double GetDistanceFromOrigin() {
        return Math.Sqrt(X * X + Y * Y);
    }
}
```

5. Enums (Named Constants)

5.1 Definition & Syntax

```
```csharp
public enum LogLevel // Underlying type (default: int)
{
 Info = 1, // Explicit value
 Warning, // 2 (auto-incremented)
 Error = 10 // Custom value
}
```

```
}
```

## 5.2 Key Features

- ✓ **Type-safe**: Compiler prevents invalid values.
- ✓ **Bit flags support** (`[Flags]` attribute).
- ✓ **Underlying type control** (`byte`, `short`, etc.).

## 5.3 Enum Operations

```
// Parsing
LogLevel level = Enum.Parse<LogLevel>("Warning");

// Flags (bitwise)
[Flags]
public enum Permissions
{
 Read = 1,
 Write = 2,
 Execute = 4
}
var perms = Permissions.Read | Permissions.Write;
```

## 6. MSDN References

- [Structs \(C#\)](#)
- [Enums \(C#\)](#)

## Inheritance & Polymorphism

## 1. Inheritance: The "is-a" Relationship

### 1.1 Definition

- **Inheritance** allows a class (**derived class**) to inherit fields/methods from another (**base class**).
- Models hierarchical relationships (e.g., `Dog : Animal`).

### 1.2 Syntax

```
public class Animal // Base class
{
 public string Name { get; set; }
 public void Eat() {
 Console.WriteLine($"{Name} is eating.");
 }
}

public class Dog : Animal // Derived class
{
 public void Bark() {
 Console.WriteLine($"{Name} sounds Woof!");
 }
}
```

### 1.3 Key Rules

- ✓ **Single Inheritance:** A class can inherit from **only one** base class.
- ✓ **Transitivity:** If `C : B` and `B : A`, then `C` inherits from `A`.
- ✓ **Constructors:** Not inherited (but can be chained with `base()`).

### 1.4 Inheritance Types

1. **Single Inheritance:** One class inherited from one base class.

```
class Fruit { /* ... */ }
class Mango : Fruit { /* ... */ }
```

2. **Multiple Inheritance:** One class inherited from multiple base classes. Not supported in C# for the classes, but one class can implement multiple interfaces.

```
interface IPrintable { /* ... */ }
interface IFormattable { /* ... */ }
class Document : IPrintable, IFormattable { /* ... */ }
```

3. **Hierarchical Inheritance:** Multiple classes inherited from single base class.

```
class Person { /* ... */ }
class Employee : Person { /* ... */ }
class Student : Person { /* ... */ }
```

4. **Multi-level Inheritance:** Multiple levels of inheritance.

```
class Person { /* ... */ }
class Player : Person { /* ... */ }
class Cricketer : Player { /* ... */ }
```

5. **Hybrid Inheritance:** Any combination of the above inheritance types.

2. **Polymorphism: Many Forms**

## 2.1 Definition

- **Polymorphism** allows objects of different classes to be treated as objects of a common base class.
- Achieved via:
  - **Method overriding** (runtime polymorphism).
  - **Method overloading** (compile-time polymorphism).

## 2.2 Method Overriding (**virtual** + **override**)

```
public class Animal {
 public virtual void MakeSound() {
 Console.WriteLine("Some sound");
 }
}

public class Dog : Animal {
 public override void MakeSound() {
 Console.WriteLine("Woof!");
 }
}

// Usage:
Animal myDog = new Dog();
myDog.MakeSound(); // Output: "Woof!" (Runtime decision)
```

## 2.3 Method Overloading (Compile-Time)

```
public class Logger {
 public void Log(string message) { ... }
```

```
public void Log(int number) { ... } // Same name, different params
}
```

### 3. Abstract Classes & Methods

#### 3.1 Abstract Classes

- Cannot be instantiated.
- Define **partial implementations** for derived classes.

```
public abstract class Shape
{
 public abstract double Area(); // No implementation
}

public class Circle : Shape
{
 public double Radius { get; set; }
 public override double Area() => Math.PI * Radius * Radius;
}
```

#### 3.2 Interfaces vs. Abstract Classes

Feature	Interface	Abstract Class
Inheritance	Multiple	Single
Methods	No implementation	Partial implementation
Fields	Not allowed	Allowed

### 4. The **sealed** Keyword

- Prevents further inheritance:

```
public sealed class UltimateClass { } // Cannot be derived
public class Attempt : UltimateClass { } // ✗ Compile error
```

- Can also seal individual methods:

```
public override sealed void Method() { } // No further overrides
```

## 5. The **new** Keyword

- Hides base class method/property in derived class.

```
class Animal {
 public virtual void MakeSound() {
 Console.WriteLine("Generic animal sound");
 }
}

class Dog : Animal {
 public new void MakeSound() { //Hides the base class's MakeSound method
 Console.WriteLine("Woof!");
 }
 // method can also be "new virtual" in order to override it in Dog's sub-classes.
}
```

```
// In Main()
Animal myDogAsAnimal = new Dog();
```

```
myDogAsAnimal.MakeSound();
//Output: Generic animal sound (because myDogAsAnimal is an Animal reference and MakeSound() is "new" in Dog, not
"override").
```

## 6. Example: Payment System

```
public abstract class PaymentMethod
{
 /*Other members*/
 public abstract void ProcessPayment(double amount);
}

public class CreditCard : PaymentMethod
{
 public override void ProcessPayment(double amount) {
 Console.WriteLine($"Paid ${amount} via Credit Card");
 }
}

public class PayPal : PaymentMethod
{
 public override void ProcessPayment(double amount) {
 Console.WriteLine($"Paid ${amount} via PayPal");
 }
}

// Usage:
PaymentMethod payment = new CreditCard();
payment.ProcessPayment(100); // Polymorphic call
```

## 7. Casting in Inheritance Hierarchies

### 5.1 Upcasting (Implicit)

- Treating a derived class as its base type (always safe).

```
Dog dog = new Dog();
Animal animal = dog; // Upcast (Dog → Animal)
```

### 5.2 Downcasting (Explicit)

- Treating a base type as a derived type (requires check).

```
Animal animal = new Dog();

// Downcast (Animal → Dog) - Unsafe
Dog dog = (Dog)animal;

// Safer with `is` and casting:
if (animal is Dog) {
 Dog d = (Dog)animal;
 d.Bark();
}

// OR Shorthand typesafe downcasting:
if (animal is Dog d) {
 d.Bark();
}
```

### 5.3 **as** Operator (Safe Cast)

- Returns `null` (instead of throwing) if cast fails.

```
Animal animal = new Cat();
Dog dog = animal as Dog; // Returns null (not a Dog)
```

## 7. Best Practices

- **Favor composition over inheritance** when possible.
- **Use abstract** for incomplete base implementations.
- **Avoid deep inheritance hierarchies** (>3 levels).
- **Use is/as** for safe downcasting.

## 8. MSDN References

- [Inheritance \(C#\)](#)
- [Type Conversions \(C#\)](#)
- [Polymorphism \(C#\)](#)

SUNBEAM INFOTECH

# .NET

---

## System.Object Class

### Introduction

The `Object` class (`System.Object`) serves as the ultimate base class for all types in the .NET type system, introduced in the very first version of the framework (2002). As the root of the type hierarchy, it provides common functionality that every .NET type inherits, ensuring a consistent behavioral contract across all objects.

### Core Functionality and Members

#### 1. Common Methods

- All objects inherit these fundamental methods:
  - `Equals()`: Supports value equality comparison
  - `GetHashCode()`: Provides hash code generation
  - `ToString()`: Offers string representation
  - `GetType()`: Returns runtime type information

#### 2. Virtual Methods

- Key overridable methods that enable polymorphism:
  - `Equals()`: Default implementation uses reference equality
  - `GetHashCode()`: Should be overridden when Equals is overridden
  - `ToString()`: Default returns fully qualified type name
  - `Finalize()`: Cleanup method - Considered deprecated and should be avoided in favor of `IDisposable`

#### 3. Final Methods

- Non-overridable critical operations:
  - `GetType()`: Prevents tampering with type identity

- `MemberwiseClone()`: Provides shallow copying capability

## Type System Integration

### 1. Unified Type Hierarchy

- Value types inherit via `System.ValueType` (which inherits from `Object`)
- Reference types inherit directly or indirectly from `Object`
- Enables polymorphic treatment of all types

### 2. Boxing Mechanism

- Facilitates value type to reference type conversion:
  - Value types get boxed when cast to `Object`
  - Creates heap allocation and copy overhead

### 3. Type Safety Foundation

- Provides the basis for:
  - Runtime type checking
  - Reflection capabilities
  - Safe casting operations

## Common Usage Patterns

### 1. Generic Programming

Serves as constraint-less upper bound:

```
void Process(object item) { ... } // Accepts any type
```

## 2. Collections (Pre-Generics)

Non-generic collections (ArrayList, Hashtable) used Object as element type:

```
ArrayList list = new ArrayList();
list.Add(42); // Boxing occurs
```

## 3. Reflection Scenarios

Enables type-agnostic processing:

```
object instance = Activator.CreateInstance(someType);
```

## Best Practices

### 1. Method Overriding

- When overriding Equals():
  - Maintain reflexivity, symmetry, and transitivity
  - Override GetHashCode() consistently
  - Consider implementing IEquatable

### 2. ToString() Implementation

- Provide meaningful string representation
- Include all significant state information
- Keep culture-invariant for machine consumption

### 3. Type Checking

- Prefer pattern matching over direct GetType() checks:

```
if (obj is string s) { ... } // Modern approach
```

## Performance Considerations

### 1. Boxing Overhead

- Value type to Object conversion is expensive
- Generics (List) eliminate this overhead

### 2. Virtual Call Impact

- Virtual method calls have slight overhead
- Sealed classes can optimize this

### 3. Hash Code Generation

- Poor GetHashCode() implementations hurt hash tables
- Should be fast and produce well-distributed values

## MSDN References

- [Object Class \(System\)](#)
- [Object Lifetime](#)
- [Type System Fundamentals](#)

## Interfaces

### 1. What is an Interface?

### 1.1 Definition

- A **contract** that defines a set of methods/properties a class **must** implement.
- Pure abstraction (no implementation until C# 8.0).

### 1.2 Key Characteristics

- **No fields** (only methods, properties, events, indexers).
- **No constructors** (cannot be instantiated directly).
- **Multiple inheritance** (a class can implement many interfaces).

### 1.3 Syntax

```
public interface ILogger
{
 void Log(string message);
 string LogLevel { get; set; } // Property
}
```

## 2. Why Use Interfaces?

### 2.1 Design Benefits

- **Decoupling**: Code depends on abstractions, not concrete classes.
- **Polymorphism**: Different classes can be treated uniformly.
- **Testability**: Easy mocking for unit tests.

### 2.2 Real-World Analogy

- **Interface** → USB port (standardized contract).
- **Implementing Class** → Device (e.g., phone, laptop).

### 3. Implementing Interfaces

#### 3.1 Basic Implementation

```
public class FileLogger : ILogger {
 public string LogLevel { get; set; }
 public void Log(string message) {
 File.WriteAllText("log.txt", message);
 }
}
```

#### 3.2 Explicit Implementation

- Avoids naming conflicts when implementing multiple interfaces.

```
public class HybridLogger : ILogger, IDisposable
{
 void ILogger.Log(string message) { /* ILogger's Log */ }
 void IDisposable.Dispose() { /* IDisposable's Dispose */ }
}

// Usage:
ILogger logger = new HybridLogger();
logger.Log("Test"); // Calls ILogger.Log
```

### 4. Interface Inheritance

#### 4.1 Chaining Interfaces

```
public interface IAuditable
{
 void Audit();
}

public interface IAdvancedLogger : ILogger, IAuditable
{
 void LogError(Exception ex);
}
```

#### 4.2 Rules

- Interfaces can inherit other interfaces.
- Classes must implement all parent interface members.

#### 5. Interfaces vs. Abstract Classes

Feature	Interface	Abstract Class
Inheritance	Multiple	Single
Implementation	None (until C# 8.0)	Partial
Fields	✗	✓
Constructors	✗	✓

#### 6. Common .NET Interfaces

Interface	Purpose	Key Method
IEnumerable	Iteration	GetEnumerator()
IDisposable	Resource cleanup	Dispose()

Interface	Purpose	Key Method
IComparable	Sorting	CompareTo()
IComparer	Sorting	Compare()

## 7. Design Patterns with Interfaces

### 7.1 Strategy Pattern

```
public interface IPaymentStrategy {
 void ProcessPayment(double amount);
}

public class CreditCardPayment : IPaymentStrategy { ... }
public class PayPalPayment : IPaymentStrategy { ... }

// Usage:
IPaymentStrategy strategy = new CreditCardPayment();
strategy.ProcessPayment(100);
```

### 7.2 Dependency Injection

```
public class OrderService {
 private readonly ILogger _logger;
 // Constructor injection
 public OrderService(ILogger logger) {
 _logger = logger;
 }
}
```

## 8. Best Practices

- **Prefix with I** (e.g., `ILogger`).
- **Keep interfaces small** (Single Responsibility Principle).
- **Favor interfaces for cross-cutting concerns** (logging, caching).

## 9. MSDN References

- [Interfaces \(C#\)](#)
  - [Explicit Interface Implementation](#)
- 

# Generics in C# .NET

## Introduction

Generics were introduced in C# 2.0 (.NET Framework 2.0, 2005) as a revolutionary feature to address several limitations of the original type system. This implementation was influenced by templates in C++ but designed with .NET's runtime characteristics in mind.

## Core Concepts and Definitions

### 1. Generic Type Parameters

Generics allow the definition of type parameters (denoted by `<T>`) that serve as placeholders for actual types. These parameters enable:

- Creation of classes, interfaces, methods and delegates that work with any data type
- Compile-time type safety without sacrificing performance
- Elimination of runtime type checking and casting

### 2. Better Generics Implementation

- Unlike Java's type erasure, .NET implements generics at:
  - Runtime level (CLR understands generics)

- JIT compilation level (specialized native code generation)
- Metadata level (preserved in assemblies)

### 3. Performance Benefits

- Generics provide significant advantages over non-generic approaches:
  - Eliminate boxing for value types
  - Reduce runtime type checks
  - Enable more efficient code generation

## Generic Type Definitions

### 1. Generic Classes

Classes can be defined with one or more type parameters:

```
public class Box<T> {
 private T val;
 public void Set(T value) { this.val = value; }
 public T Get() { return this.val; }
 public void Display() {
 Console.WriteLine(val);
 }
}
```

```
Box<string> b1 = new Box<string>(); // generic type given = string
b1.Set("Secret");
string str = b1.Get();
// ...

Box<int> b2 = new Box<int>(); // generic type given = int
```

```
b2.Set(123);
int num = b2.Get();
// ...
```

## 2. Generic Methods

- Methods can have their own type parameters independent of the containing class.
- Non-generic class may have generic methods:

```
class Util {
 public void Swap<T>(ref T x, ref T y) {
 T t = x;
 x = y;
 y = t;
 }
}

// Usage
Util util = new Util();
double n1 = 1.1, n2 = 2.2;
util.Swap(ref n1, ref n2); // generic type is inferred = double
Console.WriteLine($"n1: {n1}, n2: {n2}");
```

- Generic class may have generic methods e.g. Set(), Get() in Box class.
- Generic class may have non-generic methods e.g. Display() in Box class.

## 3. Generic Interfaces

- Interfaces can define type parameters for implementing classes:

```
public interface IRepository<T> {
 IEnumerable<T> FindAll();
 void Add(T entity);
}
```

- Built-in examples: `IComparable<T>`, `IComparer<T>`, etc.
- Generic class/interface can be inherited into non-generic class.

```
class Emp : IComparable<Emp>
{
 public int Id { get; set; }
 public String Name { get; set; }
 public double Salary { get; set; }

 public int CompareTo(Emp? other) {
 if (other == null)
 return 1;
 int diff = this.Id - other.Id;
 return diff;
 }

 public override string ToString() {
 return $"Emp: Id={Id}, Name={Name}, Salary={Salary}";
 }
}
```

```
// Usage
Emp[] arr = new Emp[5]
{
 new Emp { Id = 2, Name = "John", Salary = 2000.0 },
 new Emp { Id = 3, Name = "Mike", Salary = 3000.0 },
 new Emp { Id = 4, Name = "Sarah", Salary = 4000.0 },
 new Emp { Id = 5, Name = "David", Salary = 5000.0 },
 new Emp { Id = 6, Name = "Emily", Salary = 6000.0 }
}
```

```
new Emp { Id = 5, Name = "Mark", Salary = 1500.0 },
new Emp { Id = 1, Name = "Steve", Salary = 3500.0 },
new Emp { Id = 4, Name = "Peter", Salary = 4000.0 },
new Emp { Id = 3, Name = "Tony", Salary = 3000.0 }
};

Console.WriteLine("Emps Sorted by Id: ");
Array.Sort(arr);
foreach (Emp x in arr)
 Console.WriteLine(x.ToString());
```

#### 4. Generic Delegates

- We will discuss this later.

### Constraints

#### 1. Constraint Types

- Constraints restrict what types can be used as arguments:
  - `where T : struct` (value type)
  - `where T : class` (reference type)
  - `where T : new()` (default constructor)
  - `where T : BaseClass` (specific base class)
  - `where T : ISomeInterface` (interface implementation)

#### 2. Examples

```
public static T Max<T>(T first, T second) where T : IComparable<T> {
 return first.CompareTo(second) > 0 ? first : second;
}
```

```
// Usage : If Emp implements IComparable<Emp>
Emp e1 = new Emp { Id = 1, Name = "Steve", Salary = 3500.0 };
Emp e2 = new Emp { Id = 4, Name = "Peter", Salary = 4000.0 };
Emp me = Max(e1, e2);
```

### 3. Multiple Constraints

- A type parameter can have multiple constraints:

```
public T CreateInstance<T>() where T : class, new()
{
 return new T();
}
```

### Limitations

- Generic type parameters cannot be used for:
  - Static fields/methods specific to the constructed type
  - Operator overloading
  - Certain reflection operations

## Advanced Generic Patterns

### 1. Covariance/Contravariance

- Introduced in C# 4.0 for interfaces/delegates:
  - Covariance (`out T`) allows more derived types
  - Contravariance (`in T`) allows less derived types

### 2. Generic Variance

- Enables more flexible type relationships:

```
IEnumerable<string> strings = new List<string>();
IEnumerable<object> objects = strings; // Covariant
```

### 3. Default Values

- The `default` keyword handles null/non-null cases:

```
T value = default(T); // null for classes, zero for structs
```

## Performance Considerations

### 1. Code Generation

- The JIT compiler generates:
  - Separate code for each (used) value type
  - Shared code for reference types

### 2. Memory Efficiency

- Generics eliminate:
  - Boxing overhead for value types
  - Redundant code for similar operations

### 3. Runtime Efficiency

- Provides:
  - Direct method calls without casting

- Optimized collections for value types

## Best Practices

### 1. Naming Conventions

- Single type parameters: T, TResult
- Multiple parameters: TSource, TResult
- Descriptive when needed: TEntity, IRepository

### 2. Constraint Usage

- Apply constraints:
  - As loosely as possible
  - Only when necessary

### 3. Design Considerations

- Prefer generic methods over whole generic classes when possible
- Consider generic base classes for shared functionality
- Document type parameter requirements

## MSDN References

- [Generics \(C# Programming Guide\)](#)
- [Generic Classes and Methods](#)
- [Constraints on Type Parameters](#)

---

## Delegates in C#

### 1. What is a Delegate?

### 1.1 Definition

- A **delegate** is a **type-safe function pointer** that references methods with a specific signature.
- Enables **callback mechanisms, event handling, and dynamic method invocation.**

### 1.2 Key Properties

- **Type-safe:** Compiler enforces method signatures.
- **Can reference static or instance methods.**
- **Multicast capable** (invokes multiple methods sequentially).

## 2. Delegate Declaration & Usage

### 2.1 Basic Syntax

```
// Step 1: Declare a delegate type
public delegate void LogMessage(string message);

// Step 2: Create a delegate instance (built-in method)
LogMessage logger = new LogMessage(Console.WriteLine);

// Step 3: Invoke
logger("Hello, delegates!"); // Calls Console.WriteLine
```

### 2.2 Assigning Methods

```
// Static method (user-defined method)
static void LogToFile(string msg) {
 File.WriteAllText("log.txt", msg);
}
```

```
// Step 2: Initialize delegate instance (to user-defined static method)
LogMessage fileLogger = new LogMessage(LogToFile);

// Step 3: Invoke
fileLogger("Hello, delegates!"); // Calls LogToFile()
```

```
// Define Instance method (user-defined method)
class NotificationService {
 // fields
 public void SendEmail(String msg) {
 // Send email
 }
}

// Step 2: Initialize delegate instance (to user-defined non-static method)
NotificationService service = new NotificationService();
LogMessage emailLogger = new LogMessage(service.SendEmail);
//OR
LogMessage emailLogger = service.SendEmail; // Implicit syntax - Internally allocates delegate object

// Step 3: Invoke
emailLogger("Hello, delegates!"); // Calls service.SendEmail()
```

### 3. Multicast Delegates

#### 3.1 Chaining Methods

- Use `+=` to add and `-=` to remove methods.

```
LogMessage multiLogger = Console.WriteLine;
multiLogger += LogToFile;
```

```
multiLogger += service.SendEmail;

multiLogger("This goes to both!"); // Calls all three methods
```

### 3.2 Return Values in Multicast

- Only the **last method's return value** is captured.

```
public delegate int MathOp(int x, int y);

public int Add(int a, int b) {
 return a + b;
}

MathOp operations = Add;
operations += Subtract;
operations += Multiply;

int result = operations(3, 4); // Returns 12 (Multiply's result)
```

- To access return values of individual delegates use GetInvocationList().

```
Delegate[] delegates = operations.GetInvocationList();
foreach (MathOp del in delegates) {
 int result = del(22, 7);
 Console.WriteLine(result);
}
```

## 4. Delegate Use Cases

#### 4.1 Callbacks

```
delegate void StatusHandler(string status);

public void ProcessData(StatusHandler callback) {
 // ...
 callback("Success!");
}

// Usage
ProcessData(Console.WriteLine);
```

#### 4.2 Strategy Pattern

```
public delegate int PaymentStrategy(double amount);

public class PaymentProcessor {
 public void Process(PaymentStrategy strategy, double amount) {
 int status = strategy(amount);
 Console.WriteLine($"Status: {status}");
 }
}

class Program {
 static void Main(string[] args) {
 string PayByCreditCard(double amount) {
 // ...
 return "Success";
 }

 string PayByUPI(double amount) {
 // ...
 }
 }
}
```

```
 return "Failed";
 }

 PaymentProcessor processor = new PaymentProcessor();
 processor.Process(PayByCreditCard, 2000.0);
 processor.Process(PayByUPI, 4000.0);
}
}
```

## 5. Best Practices

- Use **Action/Func** for common cases (avoid custom delegates) - (we'll cover later).
- Always check for **null** before invocation:

```
logger?.Invoke("Safe call");
```

- Prefer events for pub-sub patterns (we'll cover later).

## 6. MSDN References

- [Delegates \(C#\)](#)

---

## Func/Action/Predicate Delegates

### 1. Evolution

#### 1.1 Traditional Delegates

- Since C# 1.0 (2002)
- **Type-safe function pointers** that reference methods with specific signatures.

- Must be declared before use i.e. Required explicit delegate type declarations.
- Example:

```
delegate int MathOperation(int a, int b); // Custom delegate type
int Add(int a, int b) {
 return a + b;
}
MathOperation add = new MathOperation(Add);
```

## 1.2 Modern Generic Delegates

- In C# 3.0 (2007)
- Introduced **Func**, **Action**, and **Predicate** to reduce boilerplate.
- Part of the **System** namespace.

## 2. Generic Delegates (**Func**, **Action**, **Predicate**)

Delegate	Purpose	Signature
Action	For void methods	Action<T1, T2>
Func	For methods with return values	Func<T1, T2, TResult>
Predicate	For boolean conditions	Predicate<T>

### 3.1 Action Delegate

- **Purpose:** For methods that **return void**.
- **Overloads:** **Action**, **Action<T>**, **Action<T1, T2>**, ..., up to 16 parameters.
- **Example:**

```
Action<string> log = Console.WriteLine;
log("Hello, Action!");
```

### 3.2 Predicate Delegate

- **Purpose:** For methods that **return a boolean** i.e. test a condition.
- Legacy delegate: largely replaced by `Func<T, bool>`.
- **Example:**

```
boolean IsEven(int x) {
 return x % 2 == 0;
}
Predicate<int> condition = IsEven;
bool flag = condition(4); // true
```

### 3.3 Func Delegate

- **Purpose:** For methods that **return a value**.
- **Last type parameter:** Return type.
- **Example:**

```
int Add(int x, int y) {
 return x + y;
}
Func<int, int, int> add = Add;
int result = add(5, 7); // 12
```

## Comparison with Java's Functional interfaces

C# Delegate	Java Functional Interface	Description
Action<T>	Consumer<T>	Method with no return value (void)
Func<TResult>	Supplier<TResult>	Method with no arguments, returns a value
Func<T, TResult>	Function<T, TResult>	Method with one argument, returns a value
Func<T1, T2, TResult>	BiFunction<T1, T2, TResult>	Method with two arguments, returns a value
Predicate<T>	Predicate<T>	Method with one argument, returns boolean

### 3. When to Use Which

#### 3.1 Prefer Func/Action When:

- Needing **quick, inline & standard delegate definitions**.
- Working with **lambda expressions**.
- Using **LINQ** or functional programming patterns.

#### 3.2 Prefer Traditional Delegates When:

- Defining **event handlers** (e.g., `public delegate void EventHandler()`).
- Requiring **explicit naming** for API clarity.
- Handling **specialized signatures** (e.g., `ref/out` parameters).
- **Specialized signatures** not covered by `Func/Action`.

### 4. Performance Considerations

- **No runtime performance difference**: All delegates compile to similar IL.
- **Memory**: Traditional delegates may marginally increase metadata size.

### 5. Best Practices

- ✓ Use **Func/Action** for lambda expressions and LINQ.
- ✓ Reserve **traditional delegates** for events and public APIs.
- ✓ Replace **Predicate<T>** with **Func<T, bool>** for consistency.

## 6. MSDN References

- [Func Delegate](#)
  - [Action Delegate](#)
- 

# Anonymous Methods and Lambda Expressions

## 1. Introduction

### 1.1 Introduction Timeline

- **C# 2.0 (2005)**: Introduced **anonymous methods** for simplified delegate syntax.
- **C# 3.0 (2007)**: Added **lambda expressions**, superseding anonymous methods in most cases.

### 1.2 Purpose

- Reduce boilerplate code for delegate instantiation.
- Enable functional programming patterns (e.g., closures, higher-order functions).

## 2. Anonymous Methods

### 2.1 Definition

- Inline delegate definitions without a named method.
- Syntax: `delegate(parameters) { ... }`

### 2.2 Key Characteristics

- ✓ **No return type declaration** (inferred from context).
- ✓ **Can omit parameters** if unused (`delegate { ... }`).
- ✓ **Require explicit return** for non-void methods.

### 2.3 Example

```
// Traditional delegate (C# 1.0)
delegate void Printer(string s);
Printer pr1 = delegate(string msg) {
 Console.WriteLine(msg);
};
pr1("Hi");

// Parameterless delegate (System.Action type)
Action pr2 = delegate {
 Console.WriteLine("Hello");
};
pr2();
```

### 2.4 Limitations

- ✗ Cannot use `yield return`.
- ✗ No support for expression-bodied syntax.

---

## 3. Lambda Expressions

### 3.1 Definition

- Concise syntax for anonymous methods: `(parameters) => expression-or-block`.
- Two forms:
  - **Expression lambdas**: Single-line

```
x => x * x
```

- **Statement lambdas:** Multi-line

```
x => {
 int res = x * x;
 return res;
}
```

### 3.2 Type Inference

- Compiler infers types from context (e.g., `Func<int, int>` for `x => x + 1`).
- Explicit types can be specified: `(int x) => x + 1`.

### 3.3 Examples

Scenario	Lambda Syntax
Square a number	<code>x =&gt; x * x</code>
Filter even numbers	<code>n =&gt; n % 2 == 0</code>
Multi-line processing	<code>s =&gt; { s = s.Trim(); return s; }</code>

### 3.4 Common Delegate Types

Delegate	Lambda Example	Purpose
Action	<code>() =&gt; Console.WriteLine()</code>	Void methods
<code>Func&lt;T&gt;</code>	<code>() =&gt; 42</code>	Return value

Delegate	Lambda Example	Purpose
<code>Predicate&lt;T&gt; x =&gt; x &gt; 0</code>		Boolean conditions

## 4. Closures and Captured Variables

### 4.1 Definition

- **Closure:** A lambda/anonymous method that captures variables from its enclosing scope.
- **Lifetime:** Captured variables persist until the delegate instance is garbage-collected.

### 4.2 Example

```
public Func<int> CreateCounter()
{
 int count = 0;
 return () => ++count; // Captures 'count'
}

// Usage:
var counter = CreateCounter();
Console.WriteLine(counter()); // 1
Console.WriteLine(counter()); // 2
```

### 4.3 Pitfalls

- ✓ **Avoid modifying captured variables** in multi-threaded contexts.
- ✓ **Memory usage:** Long-lived delegates keep captured variables alive.

## 5. Lambda vs. Anonymous Method Comparison

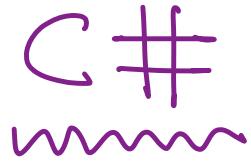
Feature	Lambda Expressions	Anonymous Methods
Syntax	<code>x =&gt; x + 1</code>	<code>delegate(int x) { return x + 1; }</code>
Type Inference	Full support	Limited
Expression-bodied	Yes	No
LINQ Compatibility	Preferred	Rarely used

## 6. Best Practices

- ✓ Prefer **lambdas** over anonymous methods (modern syntax).
- ✓ Avoid **complex logic** in lambdas (extract to methods if 3+ lines).

## 7. MSDN References

- [Anonymous Methods](#)
- [Lambda Expressions \(C#\)](#)
- [Closures](#)

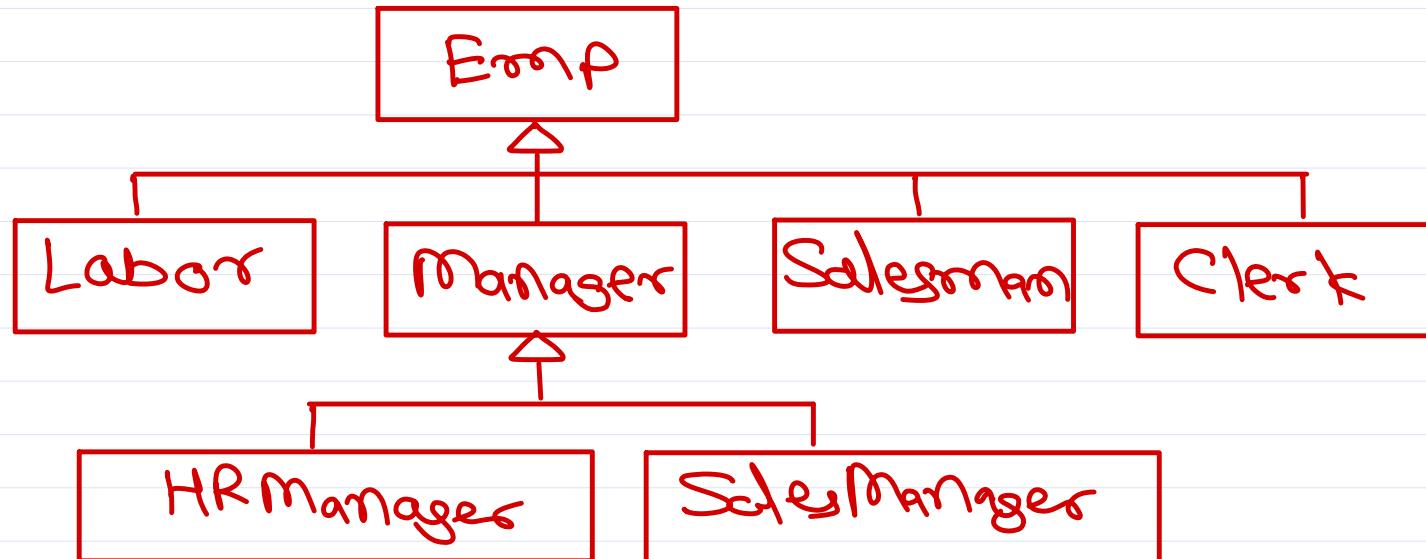


# C#.NET @ Sunbeam Infotech

***Trainer: Nilesh Ghule***



# Inheritance



classes

- fields + methods (reuse)
- virtual methods (polymorphism)
- objects

Object Oriented Analysis & Design (OOAD)

- Grady Booch .

\* SOLID

\* Design Patterns

abstract methods  
(contract)

conceptual entity  
(no objects)

abstract classes

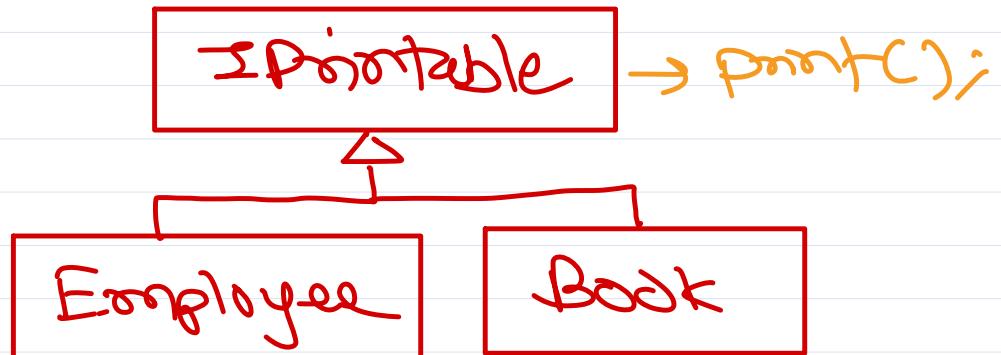
- fields + methods (reuse)
- virtual methods (polymorphism)



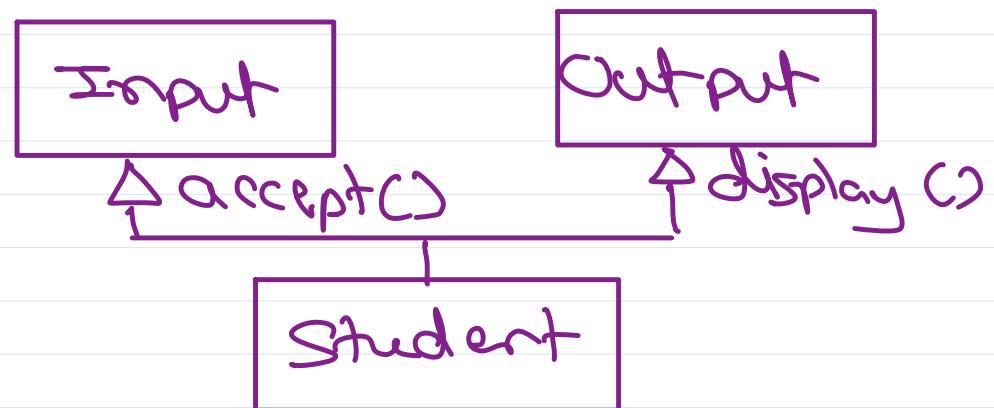
# Interfaces

## interfaces

- no reusability  
(no fields, methods,  
no ctors)
- all abstract methods  
(contract → standard)  
i.e. specification → guarantee  
that derived class has  
the functionality
- can group unrelated classes  
(Polymorphism)
- avoids fragile base class  
problem → Immutable
- no inheritance → implementation



one class may impl  
multiple interfaces .



# Delegates

delegate → like fn alias (in JS) but type safe.  
object oriented type safe function pointer.

C → fn pointer →

```
int sum (int a, int b){
 return a+b;
}
```

```
typedef int (*mathop)(int, int);
mathop ptr = sum;
res = ptr(22, 7);
call fn.
```

declare for ptr type ①

declare for ptr & initialize ②

1000

1000  
ptr



# Delegates

```
wid sum(int a, int b){
 cout<(a+b);
}
```

Step 1 → declare delegate (for ptr) type.

Syntax: delegate set-type delType Name (params & types)

```
delegate wid MathOp(int a, int b);
```

Step 2 → Create delegate obj & init it (with for add).

```
MathOp ptr = new MathOp(Sum); if static
```

→ Class Name . Sum  
→ Obj Name . Sum  
→ if non static  
if local fn

Step 3 → call the for

```
ptr(22, 7);
```





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

```
File Edit Selection View ... ← → 🔍 private
day02.md MCO2.md temp.txt
MCO2.md
104 class Root {
105 public new virtual void Greet() { }
106
107
108 class Middle : Root {
109 public override void Greet() { }
110 }
111
112 class Leaf : Middle
113 {
114 }
115
116 class Program
117 {
118 static void Main() {
119 Root obj = new Leaf();
120 obj.Greet();
121 }
122 }
```

A  $v\ f1()$  → C class vtable

B  $o\ f1()$  → f1

C  $n\ v\ f1()$  → f1 -- newslot

D  $o\ f1()$

The diagram illustrates the creation of an array in memory. A variable `emps` is assigned the value `1000`, which points to a heap-allocated array object. The array has a `Length` of `5` and a `data` field containing five `null` pointers.

```
static void Main(string[] args)
{
 Emp[] emps = new Emp[5];
 // array of 5 references created on heap
 // by default each ref is initialized to null
}
```

Length
5

data
0 null
1 null
2 null
3 null
4 null

## Object Slicing (Java/C#) -

When derived class object is assigned to base class ref,  
we can only access base class members through that ref.

```
class Person {
 public string name;
 // ...
}
class Student:Person {
 public int marks;
 // ...
}
```

```
Person p = new Student();
cw(p.name); // okay
cw(p.marks); // error
```

Exception - The overridden virtual  
members are accessed based on  
type of object (not the ref).

```
Student s = (Student) p;
cw(s.marks); // okay
```

The screenshot shows a Microsoft Visual Studio interface with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Search Bar:** Search for Day03\_Soln.
- Solution Explorer:** Shows the project 02\_OopInheritance.
- Toolbox:** Standard Visual Studio toolbox icons.
- Task List:** GitHub Copilot icon.
- Code Editor:** Program.cs file containing C# code for displaying employee details based on type (Manager, Salesman, Labor). The code uses type checking and casting.
- Output Window:** Shows the execution results:
  - Total Expense: 10700
  - \_02\_OopInheritance.Clerk Id: 1
  - \_02\_OopInheritance.Manager Id: 2
    - Bonus: 1000
  - \_02\_OopInheritance.Labor Id: 5
    - Hours: 30, Rate: 20
  - \_02\_OopInheritance.SalesManager Id: 3
    - Bonus: 600
  - \_02\_OopInheritance.Salesman Id: 4
    - Commission: 500

D:\Nilesh\Feb25\DAC\dotnet\day03\Day03\_Soln\02\_OopInheritance\bin\Debug\net8.0\02\_OopInheritance.exe (process 13120) exited with code 0 (0x0)

Press any key to close this window . . .
- Status Bar:** Shows build succeeded, ENG US, 09:44, 25-06-2025.

```
interface MyIntf {
 void Method();
}
```

```
class MyClass : MyIntf {
 public void Method() {
 // ...
 }
}
```

```
class SomeClass {
 // ...
}
class YourClass: SomeClass,
 MyIntf {
 // ...
}
```

In C# only one base class allowed.  
We may implement multiple  
interfaces.

## Generic Programming: Same logic (method/class) for any type

1. using Object class
2. using Generics (templates)

class Object --> namespace System

- root class of .NET class hierarchy
- virtual String ToString() { ... }
- virtual Boolean Equals(Object other) { ... }
- virtual int GetHashCode() { ... }
- virtual void Finalize() { ... }
- Type GetType() { ... } -- returns Metadata of class
- Object MemberwiseClone() { ... } -- returns shallow copy

## Generic Programming: Same logic (method/class) for any type

1. using Object class

2. using Generics (templates)

class Object --> namespace System

- root class of .NET class hierarchy
- virtual String ToString() { ... }
- virtual Boolean Equals(Object other) { ... }
- virtual int GetHashCode() { ... }
- virtual void Finalize() { ... }
- Type GetType() { ... } -- returns Metadata of class
- Object MemberwiseClone() { ... } -- returns shallow copy
- ...

Object - can hold object  
of any type (flexible).

object obj = 123;  
// boxing:inefficient  
.NET 1.0 - collections  
were non-generic  
e.g. ArrayList,  
Hashtable.

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day03\_Soln

Program.cs | Program.cs | Program.cs | Program.cs

05\_Generics | 05\_Generics | 05\_Generics | Main(string[] args)

Java Generics - Type Erasure  
- Every gen value is internally stored as "Object"

```
27 static void Main(string[] args) Java Generics - Type Erasure
28 Box<string> b1 = new Box<string>(); - Every gen value is internally stored as
29 b1.Set("Secret"); "Object"
30 //DateTime val = (DateTime)b1.Get(); // Compiler ERROR: Cannot convert type 'string' to 'Sy
31 string val = b1.Get(); string'
32 Console.WriteLine("b1 val: " + val); C# Generics - Type info - Compiler+CLR.
33
34 Box<int> b2 = new Box<int>(); - internally writes new classes by JIT.
35 b2.Set(123); // No Boxing
36 int val2 = b2.Get(); // No Unboxing
37 Console.WriteLine("b2 val: " + val2);
38
39 Box<double> b3 = new Box<double>(); // No Boxing
40 b3.Set(3.142); // No Unboxing
41 double val3 = b3.Get(); // No Unboxing
42 Console.WriteLine("b3 val: " + val3);
43
44
45
```

Output Error List Output

Item(s) Saved 11 0/0 20 main dotnet 11:36 25-06-2025 ENG US

**Generic constraints**

1. where T : InterfaceName
2. where T : class ←
3. where T : struct ←
4. where T : new() ←
5. where T : ClassName

The screenshot shows a Microsoft Visual Studio interface with a code editor displaying a C# program. Handwritten red text is overlaid on the code, defining 'Generic constraints' and listing five types of constraints:

```
140 }
141 4 references
142 class SomeClass
143 {
144 // ...
145 }
146 0 references
147 static void Main(string[] args)
148 {
149 // Generic Constraints -- where T is inherited from IComparable
150 static T GetMax<T>(T a, T b) where T : IComparable<T>
151 {
152 //return a > b ? a : b; // cannot use arithmetic/relative operators with gen
153 int diff = a.CompareTo(b);
154 return diff > 0 ? a : b;
155 }
156 string s1 = "ABC" s2 = "POR".
```

Handwritten notes in red:

- Generic constraints 1. where T : InterfaceName
- 2. where T : class ←
- 3. where T : struct ←
- 4. where T : new() ←
- 5. where T : ClassName

The code editor shows several tabs for 'Program.cs' and a status bar at the bottom indicating line 168, character 15, and other details.

# .NET

## Func/Action/Predicate Delegates

### 1. Evolution

#### 1.1 Traditional Delegates

- Since C# 1.0 (2002)
- **Type-safe function pointers** that reference methods with specific signatures.
- Must be declared before use i.e. Required explicit delegate type declarations.
- Example:

```
delegate int MathOperation(int a, int b); // Custom delegate type
int Add(int a, int b) {
 return a + b;
}
MathOperation add = new MathOperation(Add);
```

#### 1.2 Generic Delegates

- Since C# 2.0
- Allows generic parameters
- Examples:

```
delegate void Consumer<T>(T obj);
```

```
Consumer<string> print = Console.WriteLine;
print("Hello, World");
```

### 1.3 Pre-defined Generic Delegates

- In C# 3.0 (2007)
- Introduced **Func**, **Action**, and **Predicate** to reduce boilerplate.
- Part of the **System** namespace.

## 2. Generic Delegates (**Func**, **Action**, **Predicate**)

Delegate	Purpose	Signature
Action	For void methods	Action<T1, T2>
Func	For methods with return values	Func<T1, T2, TResult>
Predicate	For boolean conditions	Predicate<T>

### 2.1 Action Delegate

- **Purpose:** For methods that **return void**.
- **Overloads:** **Action**, **Action<T>**, **Action<T1, T2>**, ..., up to 16 parameters.
- **Example:**

```
Action<string> log = Console.WriteLine;
log("Hello, Action!");
```

### 2.2 Predicate Delegate

- **Purpose:** For methods that **return a boolean** i.e. test a condition.

- Legacy delegate: largely replaced by `Func<T, bool>`.
- **Example:**

```
boolean IsEven(int x) {
 return x % 2 == 0;
}
Predicate<int> condition1 = IsEven;
bool flag1 = condition1(4); // true
Predicate<int> condition2 = x => x % 2 != 0;
bool flag2 = condition2(4); // false
```

### 2.3 Func Delegate

- **Purpose:** For methods that **return a value**.
- **Last type parameter:** Return type.
- **Example:**

```
int Add(int x, int y) {
 return x + y;
}
Func<int, int, int> add = Add;
int result = add(5, 7); // 12
Func<int, int, int> subtract = (x,y) => x - y
result = subtract(7, 2); // 5
```

### 2.4 Comparison with Java's Functional interfaces

C# Delegate	Java Functional Interface	Description
-------------	---------------------------	-------------

C# Delegate	Java Functional Interface	Description
Action<T>	Consumer<T>	Method with no return value (void)
Func<TResult>	Supplier<TResult>	Method with no arguments, returns a value
Func<T, TResult>	Function<T, TResult>	Method with one argument, returns a value
Func<T1, T2, TResult>	BiFunction<T1, T2, TResult>	Method with two arguments, returns a value
Predicate<T>	Predicate<T>	Method with one argument, returns boolean

### 3. When to Use Which

#### 3.1 Prefer Func/Action When:

- Needing **quick, inline & standard delegate definitions**.
- Working with **lambda expressions**.
- Using **LINQ** or functional programming patterns.

#### 3.2 Prefer Traditional Delegates When:

- Defining **event handlers** (e.g., `public delegate void EventHandler()`).
- Requiring **explicit naming** for API clarity.
- Handling **specialized signatures** (e.g., `ref/out` parameters).
- **Specialized signatures** not covered by `Func/Action`.

### 4. Performance Considerations

- **No runtime performance difference**: All delegates compile to similar IL.
- **Memory**: Traditional delegates may marginally increase metadata size.

### 5. Best Practices

- ✓ Use **Func/Action** for lambda expressions and LINQ.
- ✓ Reserve **traditional delegates** for events and public APIs.
- ✓ Replace **Predicate<T>** with **Func<T, bool>** for consistency.

## 6. MSDN References

- Func Delegate
  - Action Delegate
- 

# Events

## 1. What are Events?

### 1.1 Definition

- **Events** are a language-level implementation of the **Observer pattern**, enabling objects to notify others when something happens.
- Built on top of **delegates**, but with added encapsulation and safety.

### 1.2 Key Characteristics

#### ✓ Publisher-Subscriber Model:

- **Publisher**: Raises the event.
- **Subscriber**: Responds to the event.
- ✓ **Decoupled Communication**: Publishers don't need to know about subscribers.

## 2. Event Declaration and Usage

### 2.1 Step 1: Define a Delegate

```
public delegate void PriceChangedHandler(decimal oldPrice, decimal newPrice);
```

## 2.2 Step 2: Declare the Event

```
public class Stock {
 // Event declaration (based on the delegate)
 public event PriceChangedHandler PriceChanged;

 private decimal _price;
 public decimal Price {
 get => _price;
 set {
 if (_price == value) return;
 decimal oldPrice = _price;
 _price = value;
 PriceChanged?.Invoke(oldPrice, _price); // Raise event
 }
 }
}
```

## 2.3 Step 3: Subscribe to the Event

```
var stock = new Stock { Price = 100 };

// Subscribe
stock.PriceChanged += (oldPrice, newPrice) =>
 Console.WriteLine($"Price changed from {oldPrice} to {newPrice}");

// Trigger event
stock.Price = 150; // Output: "Price changed from 100 to 150"
```

### 3. Built-in EventHandler Delegate

#### 3.1 Standard Pattern

- Instead of custom delegates, use .NET's standard:

```
public event EventHandler<PriceChangedEventArgs> PriceChanged;
```

- Requires an EventArgs subclass:

```
public class PriceChangedEventArgs : EventArgs
{
 public decimal OldPrice { get; }
 public decimal NewPrice { get; }
 public PriceChangedEventArgs(decimal oldPrice, decimal newPrice)
 {
 OldPrice = oldPrice;
 NewPrice = newPrice;
 }
}
```

#### 3.2 Usage with EventHandler

```
public class Stock
{
 public event EventHandler<PriceChangedEventArgs> PriceChanged;

 protected virtual void OnPriceChanged(PriceChangedEventArgs e)
 {
 PriceChanged?.Invoke(this, e); // 'this' is the sender
 }
}
```

```
 }

 public decimal Price
 {
 set
 {
 if (_price != value)
 {
 OnPriceChanged(new PriceChangedEventArgs(_price, value));
 _price = value;
 }
 }
 }
}
```

#### 4. Event Accessors (add/remove)

- Override default event behavior (e.g., thread-safe subscriptions):

```
private EventHandler _myEvent;
public event EventHandler MyEvent
{
 add => _myEvent += value;
 remove => _myEvent -= value;
}
```

#### 5. Best Practices

- ✓ Use `EventHandler<T>` for consistency.
- ✓ Always check for `null` before invoking:

```
PriceChanged?.Invoke(this, EventArgs.Empty);
```

- ✓ **Prefix events with `On`** for raising methods (`OnPriceChanged`).
- ✓ **Avoid long-running subscribers** (can block publishers).

## 6. Common Pitfalls

- ✗ **Memory leaks** (forget to unsubscribe).

```
// Unsubscribe when done:
stock.PriceChanged -= HandlePriceChange;
```

- ✗ **Overusing events** (simple callbacks may suffice).

## 7. MSDN References

- [Events \(C#\)](#)
- [EventHandler Delegate](#)

---

# Arrays

## Introduction

- Arrays represent one of the most fundamental data structures in C#.
- They provide efficient, fixed-size collections of elements that are stored contiguously in memory, offering O(1) access time to elements via indexing.
- The .NET Framework implements arrays as objects derived from `System.Array`, making them reference types that inherit all members from this base class.

## Array Types and Declaration

## 1. Single-Dimensional Arrays

- The simplest array form stores elements in a linear sequence:

```
int[] numbers = new int[5]; // Declaration with size
string[] names = { "Alice", "Bob", "Charlie" }; // Initialization
// traverse array using index
for(int i=0; i<names.Length; i++)
 Console.WriteLine(names[i]);
// OR - arrays are IEnumerable - traverse using foreach loop
foreach(string name in names)
 Console.WriteLine(name);
```

## 2. Multi-Dimensional Arrays

- Support matrix-like structures with rectangular:

```
//int[,] matrix = new int[3, 3]; // 3x3 matrix
int[,] matrix = new int[3, 3] {
 { 1, 2, 3 },
 { 4, 5, 6 },
 { 7, 8, 9 }
};
// access the members
for (int i = 0; i < matrix.GetLength(0); i++) {
 for (int j = 0; j < matrix.GetLength(1); j++)
 Console.Write(matrix[i, j] + " ");
 Console.WriteLine();
}
```

- Also supports jagged arrays.

```
int[][] jagged = new int[3][]; // Jagged (array of arrays)

// Initialize each row with an array of a specific length
jagged[0] = new int[] { 1, 2, 3 };
jagged[1] = new int[] { 4, 5 };
jagged[2] = new int[] { 6, 7, 8, 9 };

// Print the elements of the jagged array
for (int i = 0; i < jagged.Length; i++) {
 for (int j = 0; j < jagged[i].Length; j++)
 Console.Write(jagged[i][j] + " ");
 Console.WriteLine();
}
```

### 3. Specialized Array Types

- The .NET ecosystem includes optimized array variants:
  - `Buffer` for primitive type operations
  - `ArraySegment<T>` for partial array views
  - `Memory<T>` and `Span<T>` for safe memory access

## Memory Structure and Performance

### 1. Storage Characteristics

- Arrays allocate:
  - Contiguous memory blocks on the managed heap
  - Header information including length and sync block
  - Direct element storage (value types) or references (reference types)

### 2. Access Patterns

- Provide constant-time O(1) for:
  - Index-based read/write operations
  - Length lookup via Length property
  - Boundary checking (with runtime validation)

### 3. Performance Considerations

- Optimized for:
  - CPU cache locality (sequential access)
  - Vectorized operations via SIMD
  - Low-overhead iteration constructs

## System.Array Functionality

### 1. Core Members

- All arrays inherit essential methods:
  - Length property (total element count)
  - Rank property (number of dimensions)
  - Clone() method (shallow copy)
  - GetValue()/SetValue() reflection-style access

### 2. Sorting and Searching

- Static methods provide common algorithms:

```
Array.Sort(myArray); // Array elements should be IComparable<>
```

```
int index = Array.BinarySearch(sortedArray, value);
```

### 3. Advanced Operations (Refer Docs)

- Constrained copying (CopyTo)
- Value initialization (Clear, Fill)
- Structural operations (Reverse, Resize)

## Span and Memory

- Introduced in .NET Core 2.1 for:
  - Stack-allocated array slices
  - Unified memory access patterns
  - Reduced allocation overhead
  - Internally "ref struct" so can be used locally in methods (not as fields).
- Span Example:

```
using System;

public class SpanExample {
 public static void Main(string[] args) {
 // Create a span from an array
 int[] numbers = { 1, 2, 3, 4, 5 };
 Span<int> numberSpan = numbers.AsSpan();

 // Modify the span, which modifies the underlying array
 numberSpan[0] = 10;
 Console.WriteLine($"First element after modification: {numbers[0]}"); // Output: 10

 // Span can also be created using stackalloc
 Span<int> stackSpan = stackalloc int[3] { 6, 7, 8 };
 foreach (int number in stackSpan) // Iterate through span
 Console.WriteLine(number); // Output: 6, 7, 8
 }
}
```

```
}
```

## Best Practices and Guidelines

### 1. Size Management

- Prefer known sizes at creation
- Use List for dynamic sizing
- Consider stackalloc for small temporary arrays

### 2. Type Safety

- Avoid object[] for heterogeneous data
- Prefer generic methods when possible
- Validate array parameters in public APIs

### 3. Performance Optimization

- Minimize bounds checking in hot paths
- Use fixed buffers for interop scenarios
- Consider parallel operations for large arrays

## MSDN References

- [Arrays \(C# Programming Guide\)](#)
- [System.Array Class](#)
- [Memory and Span](#)

---

## .NET Collections

## Introduction

- The initial non-generic collections (ArrayList, Hashtable) were replaced by type-safe generic collections in .NET 2.0 (2005), with further optimizations in .NET Core and modern .NET versions.
- The System.Collections and System.Collections.Generic namespaces contain the core collection types that form the foundation of data management in .NET applications.

## Collection Overview

### 1. Key Interfaces

- IEnumerable: Foundation for iteration capability
- ICollection: Adds modification operations
- IList: Provides index-based access
- IDictionary< TKey, TValue >: Key-value pair storage
- ISet: Mathematical set operations

### 2. Primary Implementations

- List: Dynamic array implementation
- Dictionary< TKey, TValue >: Hash table implementation
- Queue/Stack: FIFO/LIFO structures
- LinkedList: Doubly-linked list
- HashSet/SortedSet: Optimized set operations

### 3. Specialized Collections

- SortedList< TKey, TValue >: Hybrid list/dictionary
  - BlockingCollection: Thread-safe producer/consumer
  - ImmutableList< T >: Thread-safe persistent structures
- Refer hierarchy diagram as well.

## Core Collection Types

## 1. List

The most commonly used collection representing a dynamically resizable array:

- Implements IList, ICollection, IEnumerable
- O(1) indexed access
- O(n) insertion/removal (except at end)
- Automatically handles capacity growth
- Example:

```
List<string> names = new List<string>();
names.Add("Alice");
names.AddRange(new[] {"Bob", "Charlie"});
foreach (string item in names)
 Console.WriteLine(item);
```

## 2. Dictionary< TKey, TValue >

- Hash-table based key-value store.
- Near O(1) lookup/insert/delete
- Requires good hash distribution
- Implements IDictionary< TKey, TValue >
- Example

```
Dictionary<int, string> employees = new();
employees.Add(101, "John Doe");
employees[102] = "Jane Smith";
int key = 101; // input from user
string name = employees[key];
System.Console.WriteLine(name);
```

### 3. HashSet

- Contains unique elements only
- O(1) membership testing
- Union/Intersect/Except operations
- Example:

```
HashSet<int> primes = new() { 2, 3, 5, 7 };
primes.Add(11); // Returns true
primes.Add(5); // Returns
foreach (int item in primes)
 Console.WriteLine(item);
```

### 4. Queue and Stack

- Queue: Enqueue/Dequeue operations - FIFO collection
- Stack: Push/Pop operations - LIFO collection
- Both provide O(1) operations
- Example:

```
Queue<string> requests = new();
requests.Enqueue("Request1");
string next = requests.Dequeue();
```

## Performance Characteristics

### 1. Time Complexity Considerations

- Array-based (List): Fast indexed access

- Node-based (LinkedList): Efficient inserts
- Hash-based (Dictionary): Best for lookups

## 2. Memory Overhead Analysis

- Reference types add per-element overhead
- Capacity vs Count management
  - always Capacity  $\geq$  Count.
  - `list.EnsureCapacity(min);` // ensures min capacity, doing reallocation if needed.
  - `list.TrimAccess();` // release excess allocation, so that Capacity = Count.

## 3. Thread Safety Options

- Concurrent collections (`ConcurrentDictionary`)
- Synchronization wrappers (lock statements)
- Immutable collections

# Specialized Collection Types

## 1. Sorted Collections

- `SortedList<TKey, TValue>`: Memory-efficient ordered dictionary
- `SortedSet`: Balanced tree implementation
- `SortedDictionary<TKey, TValue>`: Faster inserts than `SortedList`

## 2. Concurrent Collections

- `BlockingCollection`: Producer/Consumer patterns
- `ConcurrentBag`: Unordered thread-safe collection
- `ConcurrentQueue/Stack`: Lock-free implementations

## 3. Immutable Collections

- Thread-safe by design
- Structural sharing for efficiency
- Builder pattern for batch modifications
- Example

```
using System;
using System.Collections.Immutable;
public class ImmutableListExample {
 public static void Main(string[] args) {
 // Create an immutable list
 ImmutableList<string> colors = ImmutableList.Create("Red", "Green", "Blue");
 // Iterate and print the original list
 Console.WriteLine("Original List:");
 foreach (var color in colors)
 Console.WriteLine(color);
 // Create a new list by adding an item
 ImmutableList<string> newColors = colors.Add("Orange");
 // Print the new list
 Console.WriteLine("\nNew List (with Orange added):");
 foreach (var color in newColors)
 Console.WriteLine(color);
 }
}
```

## Best Practices

### 1. Collection Selection Guidelines

- Prefer generic collections over non-generic
- Choose based on access patterns

1. Access by Index: ArrayList (non-generic), List<>
2. Access by Index with fast Insert/Delete (in middle): LinkedList<>
3. FIFO: Queue<>
4. LIFO: Stack<>
5. Thread Safety: ConcurrentQueue<>, ConcurrentStack<>, ConcurrentBag<>, ...
6. Immutability: ImmutableList<>, ImmutableQueue<>, ImmutableStack<>, ...
7. Key-Value Pairs (Fast lookup): Dictionary< TKey, TValue >, SortedList< TKey, TValue >, NameValueCollection.
8. Uniques: HashSet<>
9. Fixed size, Index access: Arrays

## 2. Capacity Management

- Pre-size collections when possible
- Monitor growth operations
- Trim excess memory when appropriate

## 3. Enumeration Safety

- Avoid modification during enumeration - a modification will throw InvalidOperationException.
- Use snapshots when needed

```
List<string> myList = new List<string> { "apple", "banana", "cherry" };

foreach (string item in myList.ToList()) // Creates a copy
{
 if (item == "banana")
 myList.Remove(item); // Modifies the original, but not the copy
}
```

## Modern Enhancements and Features

## 1. Span and Memory Support

- Slice operations on arrays/lists
- Stack-only allocation options
- Unified processing of different memory types

## 2. Collection Expressions (C# 12)

Simplified initialization syntax:

```
List<int> numbers = [1, 2, 3, 4, 5];
int[] moreNumbers = [..numbers, 6, 7, 8];
```

```
Dictionary<string, int> dict = new() { ["a"] = 1, ["b"] = 2 };
```

## 3. Performance-Optimized Variants

- Pooled collections
- Struct-based implementations
- SIMD-accelerated operations

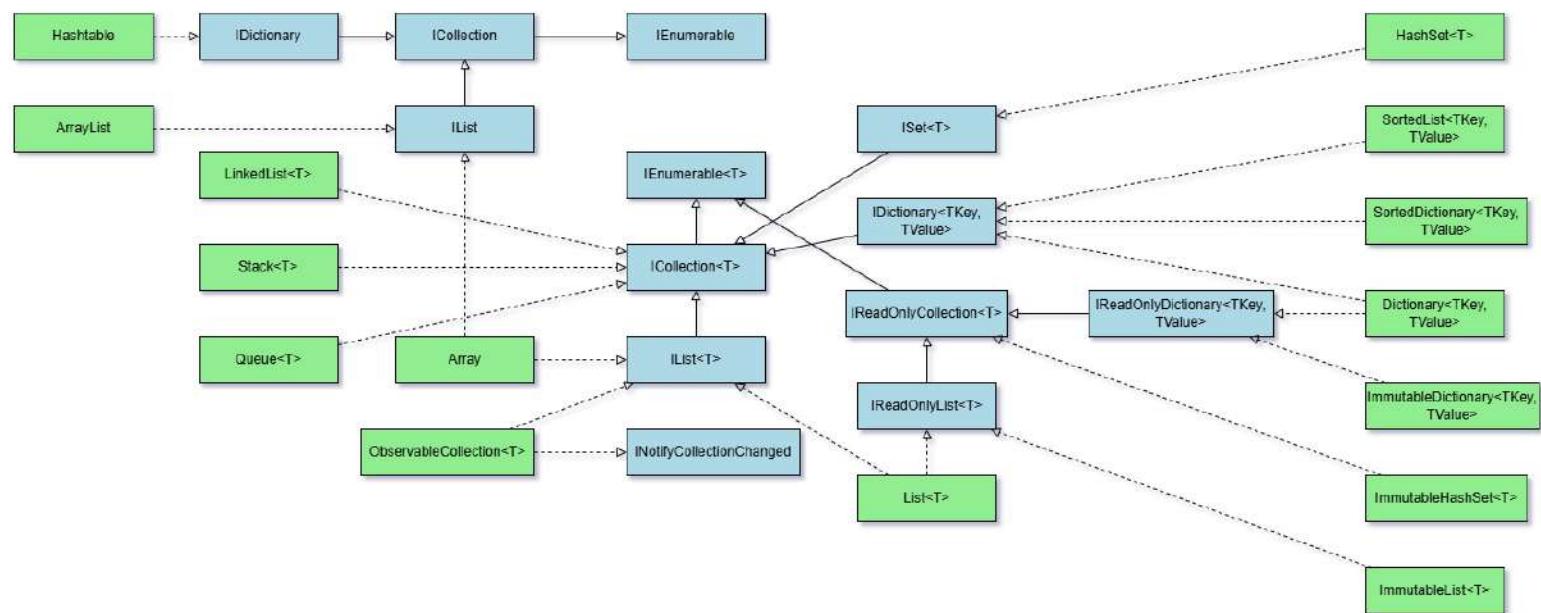
## MSDN References

- [Collections \(C# Programming Guide\)](#)
- [System.Collections.Generic](#)
- [Choosing a Collection Class](#)

---

## Class Member Access specifiers

1. **public**: Members declared with public are accessible from anywhere, without any restrictions.
2. **private**: Members declared with private are only accessible from within the same class where they are defined. This is the default access level for class members if no access specifier is specified.
3. **protected**: Members declared with protected are accessible from within the same class and from derived classes (classes that inherit from the current class).
4. **internal**: Members declared with internal are accessible only from within the same assembly (project or DLL).
5. **protected internal**: Members declared with protected internal are accessible from within the same assembly or from derived classes, even if they are in a different assembly.



File Edit Selection View ... ← → ○ private

MCQ3.md x day03.md day04.md

MCQ3.md  
9 / ---

98

99 5. What will happen when this code is executed?

100

```
101 ``csharp
102 public class Test {
103 sealed int secret = 42;
104 public void Reveal() {
105 Console.WriteLine(secret);
106 }
107 }
108
109 public class Program {
110 public static void Main() {
111 Test test = new Test();
112 test.Reveal();
113 }
114 }
115
116 }
```

Error: "sealed" cannot be used for fields.

constant fields in C#:

1. const keyword - compile time const
  - must be init at declaration
2. readonly keyword - runtime const
  - can initialized only once - either in field initializer or in constructor

```
class MyClass {
 const double PI = 3.14;
 readonly int COUNT; // = 5
 public MyClass() {
 COUNT = 5;
 }
}
```

Ln 96, Col 1 (13 selected) Spaces: 4 - UTF-8 - CRLF - Markdown - Go Live - Prettier

Search web & PC

08:12 26-06-2025

```
127
128
129 ``csharp
130 using System;
131 interface IJump {
132 void Jump();
133 }
134 abstract class Animal {
135 public virtual void Jump() {
136 Console.WriteLine("Animal jumps ");
137 }
138 public abstract void Sound();
139 }
140 class Dog : Animal, IJump {
141 public override void Jump() {
142 Console.WriteLine("Dog jumps ");
143 }
144 public override void Sound() {
145 Console.WriteLine("bark ");
146 }
147 }
148 class Program {
149 static void Main() {
150 Animal myPet = new Dog();
151 myPet.Jump();
152 myPet.Sound();
153 }
154 }
155 }
```

Annotations on the code:

- An orange arrow points from the word "override" at line 141 to the first "override" keyword at line 142.
- An orange arrow points from the word "override" at line 141 to the second "override" keyword at line 143.
- An orange arrow points from the word "impl" at line 134 to the implementation of the `Jump()` method at line 134.
- A red arrow points from the word "Common implementation" at line 533 to the implementation of the `Jump()` method at line 134.
- Red arrows point from the call to `myPet.Jump()` at line 152 and the call to `myPet.Sound()` at line 153 to the corresponding method implementations in the `Dog` class.

Diagram illustrating the execution flow of a delegate invocation list:

```

 static void Main(string[] args)
 {
 int Add(int x, int y)
 {
 return x + y;
 }

 MathRetOp op = Add;
 op += delegate (int x, int y) { return x - y; };
 op += (x, y) => x * y;

 int res = op(22, 7);
 Console.WriteLine("Result: " + res);
 }

```

The code defines a delegate type `MathRetOp` and creates an instance `op` pointing to the `Add` method. It then adds two more methods to the invocation list: a delegate lambda and another anonymous lambda. Finally, it calls `op(22, 7)` and prints the result.

**Annotations:**

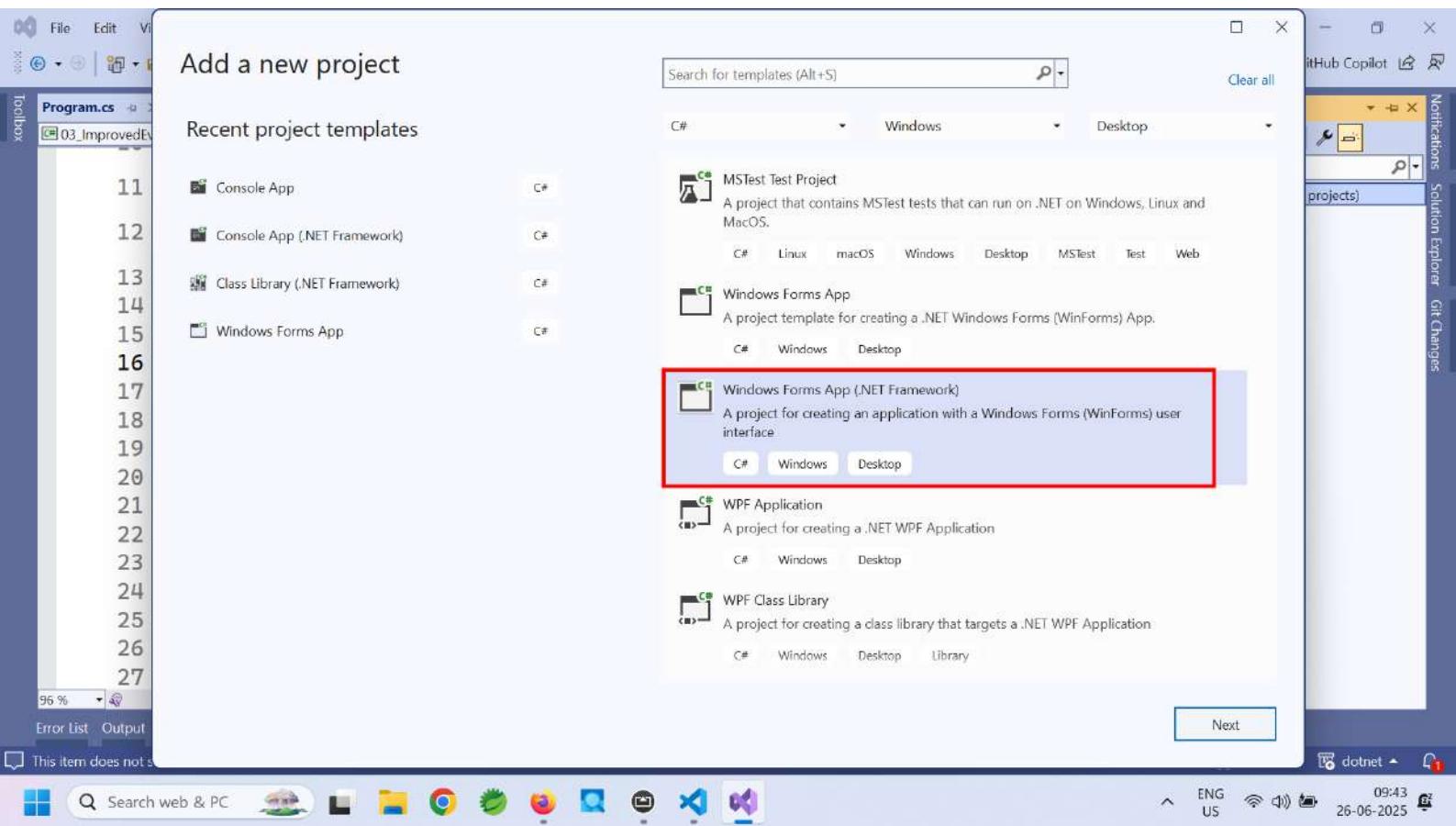
- op**: Points to the variable `op`.
- delegate obj**: Points to the `MathRetOp` object.
- invocation list**: Points to the list of methods.
- 1000**: Value returned by the `Add` method.
- 2000**: Value returned by the second added method (`x - y`).
- 3000**: Value returned by the third added method (`x * y`).
- result of last fn call.**: Points to the line `int res = op(22, 7);`.
- internally loops through invocation list and call each fn from that list..**: Describes the execution flow of the invocation list.

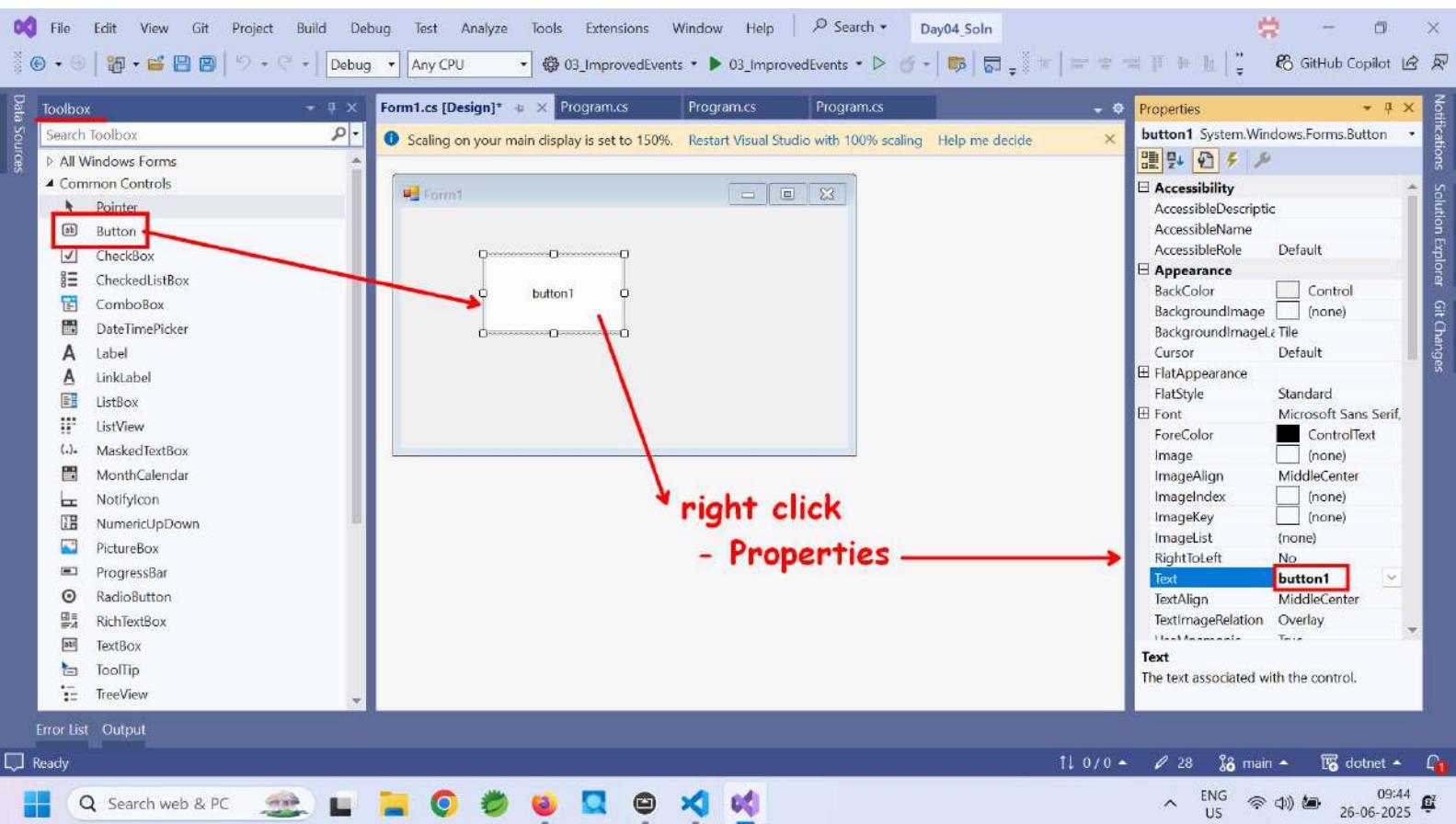
The screenshot shows a Microsoft Visual Studio interface with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Solution Explorer:** Shows projects: 01\_Delegates, 02\_Events (selected), and 04\_Soln.
- Toolbox:** Standard Visual Studio toolbox icons.
- Task List:** GitHub Copilot icon.
- Code Editor:** Program.cs file, line 111, containing:

```
111 event --> delegate -- fn pointer (multicast: multiple actions)
- may have user defined delegate i.e. fn signature.
- recommended to have a standard signature for delegate.
 void EventAction(SomeClass target, SomeEventArgs e);
object for which event occurred.
extra info about the event - Some class inherited from EventArgs.
```
- Output Window:** Shows build logs:

```
Build started at 09:20...
1>----- Build started: Project: 02_Events, Configuration: Debug Any CPU -----
1>Skipping analyzers to speed up the build. You can execute 'Build' or 'Rebuild' command to run analyzers.
1>02_Events -> D:\Nilesh\Feb25\DAC\dotnet\day04\day04_Soln\02_Events\bin\Debug\net8.0\02_Events.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
===== Build completed at 09:20 and took 00.218 seconds =====
```
- Status Bar:** Shows build status (Build succeeded), current file (main), and system information (ENG US, 09:22, 26-06-2025).





File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day04\_Soln

Form1.cs [Design] Program.cs Program.cs Form1.Designer.cs [Design]

```
11 namespace _04_ButtonClickEvent
12 {
13 public partial class Form1 : Form
14 {
15 public Form1()
16 {
17 InitializeComponent();
18 this.button1.Click += Bu
19 }
20
21 private void Button1_Click(o
22 {
23 MessageBox.Show("I am cl
24 }
25 }
26 }
```

1 namespace \_04\_ButtonClickEvent
2 {
3 partial class Form1
4 {
3 references
4
5 /// <summary>
6 /// Required designer variable
7 /// </summary>
8 private System.ComponentModel.
9
10 /// <summary>
11 /// Clean up any resources being
12 /// </summary>
13 /// <param name="disposing">t
14 protected override void Disp
15 {
16 if (disposing && (compon
17 {
18 components.Dispose();
19 }
20 }
21 }
22 }

partial class can be split into multiple .cs files of the same project.



The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, and a search bar. The toolbar below has icons for file operations like Open, Save, and Print, along with GitHub Copilot and Notifications buttons.

The main code editor window displays `Form1.cs` under the tab `Form1.cs [Design]`. The code is part of the namespace `_04_ButtonClickEvent`. It defines a partial class `Form1` that inherits from `Form`. The constructor `Form1()` calls `InitializeComponent()` and subscribes to the `button1.Click` event. A red box highlights this line. Below it is a code block for the `Button1_Click` event handler, which displays a message box. A red box highlights this entire block.

Annotations in red text are overlaid on the code:

- `this.button1.Click += Button1_Click;` is labeled "subscribing to event."
- The `Button1_Click` event handler block is labeled "implementing event handling fn"

The status bar at the bottom shows "96 %", "No issues found", "Ln: 18 Ch: 33 SPC CRLF", and "Ready". The taskbar at the bottom includes icons for Start, Search web & PC, File Explorer, Google Chrome, Mozilla Firefox, Task View, and Visual Studio, along with system status icons for battery, signal, and date/time.

```
11 namespace _04_ButtonClickEvent
12 {
13 public partial class Form1 : Form
14 {
15 public Form1()
16 {
17 InitializeComponent();
18 this.button1.Click += Button1_Click;
19 }
20
21 private void Button1_Click(object sender, EventArgs e)
22 {
23 MessageBox.Show("I am clicked!!!");
24 }
25
26 }
27 }
```

The screenshot shows the Microsoft Visual Studio IDE with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Solution Explorer:** Shows the project structure with "05\_GenericDelegates" selected.
- Toolbox:** Standard Visual Studio toolbox.
- Code Editor:** The "Program.cs" file is open. The code demonstrates a user-defined generic delegate `Supplier<T>` and its usage.

```
10 static void Main(string[] args)
11 {
12 #region Consumer<> demo
13 Consumer<string> print = Console.WriteLine;
14 // Console class -> static void WriteLine(string line);
15 print("Hello, World!");
16 #endregion
17 delegate T Supplier<T>(); <-- user defined generic delegate
18 #region Supplier<> demo
19 int RandomInt()
20 {
21 Random r = new Random();
22 return r.Next(100);
23 }
24 Supplier<int> random = RandomInt;
25 Console.WriteLine("Random Num: " + random());
26 #endregion
27
28 #region Function<> demo
```

Annotations in red and orange highlight specific parts of the code:

  - A red box surrounds the line `delegate T Supplier<T>();` with the text "`<-- user defined generic delegate`".
  - A blue box surrounds the type parameter `T` in `T=int`.
  - A pink box surrounds the implementation of `RandomInt()`, which returns the value `1000`.
  - An orange box surrounds the variable `random` in `Supplier<int> random = RandomInt;`, which also has the value `1000`.

The status bar at the bottom shows: 96%, No issues found, Ln: 8 Ch: 27 SPC CRLF, Error List, Output, Build succeeded, 10:33, ENG US, 26-06-2025.

The screenshot shows a Microsoft Visual Studio IDE window. The title bar reads "Day04\_Soln". The menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, and a Search field. The toolbar has icons for Undo, Redo, Cut, Copy, Paste, Find, Replace, and others. The solution explorer on the left shows a project named "06\_Arrays" with files Program.cs, Form1.cs, and Form1.cs [Design]. The properties window on the right shows "Main3(string[] args)". The code editor displays the following C# code:

```
62
63 static void Main(string[] args)
64 {
65 int AddAll(params int[] arr)
66 {
67 int sum = 0;
68 foreach (int num in arr)
69 sum += num;
70 return sum;
71 }
72
73 Console.WriteLine("Addition of Two Nums: " + AddAll(2, 4));
74 Console.WriteLine("Addition of Four Nums: " + AddAll(1, 2, 3, 4));
75 }
76 }
77
78 }
```

To the right of the code, handwritten-style red annotations explain the concept of variable arguments:

- Method with variable args.
- keyword "params"
- arg type must be "array"
- array of any type
- e.g. int, string, object, ...

The screenshot shows a Microsoft Visual Studio interface with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Search Bar:** Search for "Day04\_Soln".
- Toolbars:** Standard toolbar with icons for New, Open, Save, Print, etc.
- Toolbox:** Data Sources, Toolbox.
- Solution Explorer:** Shows a project named "06\_Arrays" with files "Program.cs", "Form1.cs", and "Form1.cs [Design]".
- Properties Window:** Not visible in the screenshot.
- Task List:** Not visible in the screenshot.
- Code Editor:** Displays "Program.cs" with the following code:

```
generic array objects
{
 static void Main(string[] args)
 {
 // flexible - can store any type
 object[] arr = new object[5];
 arr[0] = 123;
 arr[1] = 3.14;
 arr[2] = "string";
 arr[3] = DateTime.Now;
 arr[4] = true;
 // to access each elem (and their methods) -
 foreach (object obj in arr)
 {
 if (obj is int i)
 Console.WriteLine("int elem = " + i);
 else if (obj is double d)
 Console.WriteLine("double elem = " + d);
 else if (obj is string s)
 Console.WriteLine("string elem = " + s);
 }
 }
}
```
- Output Window:** Shows the output of the debug session:

```
int elem = 123
double elem = 3.14
string elem =
datetime elem = 26-06-2025 11:18:45
unknown type elem = True

D:\Nilesh\Feb25\DAC\dotnet\day04\Day04_Soln\06_Arrays\bin\Debug\net8.0\06_Arrays.exe (process 19244) exited with code 0 (0x0).
Press any key to close this window . . .
```
- Visual Studio Debug Window:** Shows the state of variables in memory:

```
int elem = 123
double elem = 3.14
string elem =
datetime elem = 26-06-2025 11:18:45
unknown type elem = True
```
- Taskbar:** Shows the Windows Start button, a search bar, and pinned icons for File Explorer, Edge, Google Chrome, Mozilla Firefox, and Visual Studio Code.
- System Tray:** Shows battery level, signal strength, and the date and time (11:18, 26-06-2025).

Handwritten annotations in red text are overlaid on the code editor area:

- "Unefficient: Boxing & Unboxing"

# .NET

## Partial classes

### 1. Introduction

- A partial class in C# allows you to split the definition of a class, struct, or interface across multiple source files.
- When the application is compiled, these parts are combined into a single class. This feature is useful for organizing large classes, especially when multiple developers are working on the same class, or when dealing with automatically generated code.

### 2. Key Features

- Splitting Class Definition: A partial class's definition is divided into multiple files, each marked with the partial keyword.
- Compilation: The compiler combines all the partial class definitions into a single class during compilation.
- Same Namespace and Class Name: All parts of a partial class must belong to the same namespace and have the same name.
- Accessibility: All parts of a partial class must have the same accessibility (e.g., public, private, internal).
- Partial Structures and Interfaces: The partial keyword can also be used with structures and interfaces.

### 3. Benefits

- Organization: Large classes can be broken down into smaller, more manageable files.
- Collaboration: Multiple developers can work on different parts of the same class simultaneously.
- Generated Code: Partial classes are often used with automatically generated code (e.g., in Visual Studio) to add custom logic without modifying the generated code directly.
- Readability: Splitting a class into logical parts can make the code easier to read and understand.

### 4. Example:

```
// File: Person.cs
public partial class Person {
```

```
public string FirstName { get; set; }
public string LastName { get; set; }
}
```

```
// File: Person.Methods.cs
public partial class Person {
 public void DisplayName() {
 Console.WriteLine($"Name: {FirstName} {LastName}");
 }
}
```

```
// Usage:
Person person = new Person();
person.FirstName = "John";
person.LastName = "Doe";
person.DisplayName(); // Output: Name: John Doe
```

## MSDN References

- [Partial Classes and Members](#)

## Partial Methods

### 1. Introduction

- A partial method is a special type of method that allows you to split its declaration and implementation across multiple parts of a partial class.
- If the implementation is not provided, the method and all calls to it are removed at compile time, making it useful for scenarios like code generation or optional functionality.

## 2. Key characteristics:

- A partial method has a declaration (signature) in one part of a partial class and an optional definition (implementation) in the same or another part.
- If the implementation is not provided, the method and all calls to it are removed during compilation. This is a key feature for code generation and optional features.
- Partial Class Requirement: Partial methods must be declared within a partial class or struct. Partial methods **not possible** in interfaces.
- Partial methods must have a void return type.
- Partial methods cannot have access modifiers like public, private, etc.
- They cannot have out parameters

## 3. Example

```
public partial class MyPartialClass {
 partial void OnNameChanged(string oldName, string newName);
}

public partial class MyPartialClass {
 partial void OnNameChanged(string oldName, string newName) {
 Console.WriteLine($"Name changed from {oldName} to {newName}");
 }
}

public class Example {
 public void ChangeName(string newName) {
 string oldName = "Initial Name";
 // If OnNameChanged is implemented, this line will call the implementation
 // If it's not, this line will be removed.
 OnNameChanged(oldName, newName);
 }
}
```

## static Keyword

## Introduction

- Traditionally, class `static` members represent `shared` members i.e. they are shared among all objects of the class.
- Indicates a member belongs to the type itself rather than to specific instances.
- Static members are allocated memory once when the program starts and exist for the application's lifetime.
- Apart from static members, `static` is also useful in many other cases.
- All possible uses of `static` are given below.

### 1. Static Fields

- Class-level variables shared across all instances:

```
public class Counter {
 public static int TotalCount; // Shared across all instances
 public Counter() {
 TotalCount++;
 }
}
```

### 2. Static Methods

- Utility methods that don't require instance data:

```
public class MathUtils {
 public static double CalculateCircleArea(double radius) {
 return Math.PI * radius * radius;
 }
 // ...
}
// Usage: double area = MathUtils.CalculateCircleArea(5);
```

### 3. Static Constructors

- Run once when the class is first accessed.
- No access specifier (implicitly private).

```
public class ConfigLoader {
 public static readonly string ConnectionString;
 static ConfigLoader() {
 ConnectionString = ConfigurationManager.AppSettings["DBConnection"];
 }
}
```

- App.config

```
<configuration>
<connectionStrings>
 <add name="DBConnection"
 connectionString="Data Source=serverName;Initial Catalog=databaseName;User Id=userName;Password=password;"
 providerName="System.Data.SqlClient" />
 </connectionStrings>
</configuration>
```

### 4. Static Readonly vs Const

- Const keyword represent compile-time constant (replaced by compiler). Cannot be static.
- Readonly keyword represent run-time constant - can be initialized only once - field initializer or constructor.

```
public class AppConstants {
 public const double PI = 3.14159; // Compile-time constant
```

```
 public static readonly DateTime StartupTime = DateTime.Now; // Runtime constant
}
```

## 5. Static Classes

- Contain only static members and cannot be instantiated.
- Typically used to implement helper/utility classes.

```
public static class StringUtils {
 public static string SwapCase(string value) {
 StringBuilder swapped = new StringBuilder(value.Length);
 foreach (char c in value) {
 if (char.IsUpper(c))
 swapped.Append(char.ToLower(c));
 else if (char.IsLower(c))
 swapped.Append(char.ToUpper(c));
 else
 swapped.Append(c);
 }
 return swapped.ToString();
 }
}

// Usage:
string result = StringUtils.SwapCase("SunBeam InfoTech");
```

## 6. Extension Methods (C# 3.0+)

- Enable adding methods to existing types without inheritance or modification
- Guidelines:

- Must be defined in a static class
  - First parameter uses `this` modifier
  - Appear as instance methods on target type
- Example:

```
public static class StringExtensions {
 public static bool IsValidEmail(this string input) {
 return Regex.IsMatch(input, @"^[@\s]+@[^\s]+\.[^\s]+$");
 }

 // Usage:
 string email = "test@example.com";
 bool isValid = email.IsValidEmail();
```

## 7. Static Local Functions (C# 8.0+)

- Prevents accidental capture of enclosing scope variables:

```
public void ProcessData() {
 int baseValue = 10;
 static int AddNumbers(int a, int b) {
 // Cannot access baseValue here
 return a + b;
 }
}
```

## 8. Operator Overloading

- Define custom behavior for operators on custom types

- Characteristics
  - Must be declared as public static
  - At least one parameter must be containing type
  - Certain operators must be overloaded in pairs ( $==/!=$ ,  $</>$ , etc.)
- Example:

```
public struct Vector {
 public double X, Y;

 public static Vector operator +(Vector a, Vector b) {
 return new Vector { X = a.X + b.X, Y = a.Y + b.Y };
 }

 public static bool operator ==(Vector a, Vector b) {
 return a.X == b.X && a.Y == b.Y;
 }

 public static bool operator !=(Vector a, Vector b) {
 return !(a == b);
 }
}
```

```
// Usage
Vector v1 = new Vector() { X = 3, Y = 2 };
Vector v2 = new Vector() { X = 1, Y = 4 };
Vector v3 = v1 + v2; // invokes overloaded operator+
// ...
if(v1 == v2)
 Console.WriteLine("Same");
else
 Console.WriteLine("Different");
```

## 9. Static Abstract Members (C# 11+)

- Enables static polymorphism in interfaces
- Key Features:
  - Interface can require implementing types to provide static members
  - Enables generic math scenarios
  - Supports operators, methods, and properties
- Example:

```
public interface IAddable<T> where T : IAddable<T> {
 static abstract T operator +(T left, T right);
}

public struct Vector : IAddable<Vector> {
 public int X, Y;

 public static Vector operator +(Vector left, Vector right) {
 return new Vector { X = left.X + right.X, Y = left.Y + right.Y };
 }
}
```

## 10. Static Imports (C# 6.0+)

- Import static members directly into scope
- Highlights:
  - Avoid repetitive class name qualification
  - Works with both types and enums

- Can lead to naming conflicts if overused
- Example:

```
using static System.Math;
using static System.Console;

double radius = 10;
double area = PI * Pow(radius, 2);
WriteLine($"Area: {area}");
```

## 11. Module Initializers (C# 9.0+)

- Run code when assembly loads
- Highlights:
  - Marked with `[ModuleInitializer]` attribute
  - Must be static void parameterless method
  - Execution order not guaranteed
- Example:

```
internal static class ModuleInit {
 [ModuleInitializer]
 public static void Initialize() {
 // Runs when assembly loads
 Console.WriteLine("Module initialized");
 }
}
```

## 12. Singleton Design Pattern

- Ensures a class has only one instance with global access point:
- Example

```
public sealed class Logger {
 private static readonly Logger _instance = new Logger();

 private Logger() {} // Private constructor

 public static Logger Instance {
 get { return _instance; }
 }

 public void Log(string message) {
 Console.WriteLine($"{DateTime.Now}: {message}");
 }
}

// Usage:
Logger.Instance.Log("Application started");
```

## Best Practices and Considerations

### 1. Extension Methods

- Keep in dedicated namespace
- Follow naming conventions (Extensions suffix)
- Document behavior clearly

### 2. Operator Overloading

- Only overload where operation is intuitive
- Maintain mathematical invariants

- Provide corresponding instance methods

### 3. Static Imports

- Use sparingly to avoid confusion
- Prefer for commonly used constants (Math.PI)
- Avoid with frequently conflicting names

### 4. Static Abstracts

- Primarily for advanced generic scenarios
- Consider performance implications
- Document requirements thoroughly

### 5. Memory Allocation

- Static members are allocated in a special high-frequency heap
- Exist for the application's entire lifetime
- Cannot be garbage collected

### 6. Thread Safety

- Static fields are shared across threads
- Requires synchronization for mutable state
- Prefer immutable static data where possible

### 7. Testing Challenges

- Static dependencies make unit testing difficult
- Consider dependency injection for testable code
- Use interfaces when mocking is needed

### 8. static: Appropriate Use Cases

- Utility methods that don't need instance data
- Shared configuration values

- Factory methods
- Extension methods

## 9. Anti-Patterns to Avoid

- Overusing static for state management
- Creating "god classes" with many static methods
- Using static as a shortcut to avoid proper DI

## MSDN References

- [Static \(C# Reference\)](#)
  - [Static Classes and Members](#)
  - [Extension Methods](#)
  - [Operator Overloading](#)
  - [Static Imports](#)
  - [Static Abstract Members](#)
  - [Singleton Implementation](#)
- 

# Nullable Types

## 1. Introduction

### 1.1 Definition

- Added in C# 2.0.
- Primarily designed for **value types** (e.g., `int`, `DateTime`) to hold `null`.
- The **reference types** can be null. C# 8.0+, reference types can be explicitly defined with the `? suffix`.
- Useful for:
  - Database fields (where values can be `NULL`).
  - Optional parameters.

## 1.2 Syntax

```
int? nullableInt = null; // Shorthand (preferred)
Nullable<double> nullableDouble = 3.14; // Full syntax
```

## 1.3 Underlying Representation

- The `Nullable<T>` struct wraps value types:

```
public struct Nullable<T> where T : struct
{
 public T Value { get; }
 public bool HasValue { get; }
}
```

## 2. Checking for `null`

### 2.1 `HasValue` Property

- `HasValue` is true/false indicate value is present/absent.

```
int? age = 25;
if (age.HasValue)
 Console.WriteLine($"Age: {age.Value}");
```

### 2.2 `GetValueOrDefault()`

- Returns the value or a default (avoiding `InvalidOperationException`).

```
int? count = null;
int safeCount = count.GetValueOrDefault(); // 0
int customDefault = count.GetValueOrDefault(100); // 100
```

### 3. Null-Coalescing Operator (??)

- Provides a fallback value if the left-hand side is `null`.

```
int? userId = null;
int actualId = userId ?? -1; // -1
```

- `??` can be chained for multiple values.

```
string name = GetName();
string showName = GetDisplayName();
string displayName = name ?? showName ?? "Anonymous";
```

### 4. Null-Conditional Operator (?.)

#### 4.1 Safe Member Access

- Short-circuits to `null` if the object is `null`.

```
Person person = null;
int? length = person?.Name?.Length; // null (no exception)
```

#### 4.2 Combining with Null-Coalescing

```
int nameLength = person?.Name?.Length ?? 0;
```

#### 5. Null-Forgiving Operator (!)

- Suppresses nullable warnings (use cautiously!).

```
string name = null!; // Assert: "I know this is null"
```

- Common Usage

```
public string Process(string? input) {
 return input!.ToUpper(); // Trust developer's judgment
}
```

---

#### 6. Casting Nullable Types

##### 6.1 Explicit Cast (Throws if null)

```
int? nullableNum = 42;
int num = (int)nullableNum; // Works
```

```
int? badNum = null;
int crash = (int)badNum; // ✗ InvalidOperationException
```

## 6.2 as Operator

```
object obj = "Hello";
string? str = obj as string; // Safe (returns null if fails)
```

```
object obj = null;
int? num = obj as int?; // num will be null obj is null - No exception
```

## 7. Arithmetic with Nullables

- Operations return `null` if any operand is `null`.

```
int? a = 10;
int? b = null;
int? sum = a + b; // null
```

## 8. Best Practices

- ✓ Use `?.` and `??` to avoid `NullReferenceException`.
- ✓ Prefer `GetValueOrDefault()` over direct `.Value` access.
- ✓ Limit `!` operator (only when certain of non-null).

## 9. MSDN References

- Nullable Value Types
  - Null-Conditional Operator
- 

## var and Anonymous Types

### 1. var Keyword (C# 3.0+)

#### Introduction

- Introduced in C# 3.0 (2007) as part of LINQ, **var** enables implicit typing where the compiler determines the variable type at compile-time. It does not create dynamically typed variables - C# remains statically typed.

#### Key Characteristics

1. **Type Inference:** Compiler determines type from initialization expression
2. **Static Typing:** Still enforces type safety at compile time
3. **Local Variables Only:** Cannot be used for fields, parameters, or return types
4. **Requires Initialization:** Must declare and initialize in one statement

#### Valid Use Cases

```
var numbers = new List<int>(); // Clear type from initialization
var name = "John Doe"; // Obvious string type
var result = Calculate(); // When type is evident from method call
```

#### Invalid Use Cases

```
var value; // Error: must be initialized
public var Id { get; set; } // Error: can't use for properties
```

```
var x = null; // Error: can't infer type
```

## Best Practices

1. Use when type is obvious from right-hand side
2. Avoid when type isn't immediately clear
3. Prefer explicit types for public API signatures
4. Consider readability in team environments

## 2. Anonymous Types (C# 3.0+)

### Definition and Purpose

- Anonymous types are compiler-generated, immutable reference types typically used in LINQ queries for temporary results. They contain read-only properties inferred from initialization.

### Key Features

1. **Compiler-Generated:** Created at compile time
2. **Immutable:** Properties are read-only
3. **Reference Type:** Allocated on heap despite value-type syntax
4. **Limited Scope:** Primarily for local use in methods

### Creation Syntax

```
var person = new { Name = "Alice", Age = 30 };
var product = new { ID = 1001, Price = 19.99m };
```

### Type Characteristics

1. **Property Inference:** Names and types from initialization
2. **Equals/GetHashCode:** Overridden for value equality
3. **ToString:** Generates property listing

#### Common LINQ Usage

```
var query = from p in products
 select new { p.Name, DiscountedPrice = p.Price * 0.9 };
```

#### Advanced Scenarios

##### 1. Nested Anonymous Types

```
var order = new {
 ID = 1001,
 Customer = new { Name = "Bob", Email = "bob@example.com" }
};
```

##### 2. Array of Anonymous Types

```
var people = new[] {
 new { Name = "Alice", Age = 30 },
 new { Name = "Bob", Age = 25 }
};
```

##### 3. Combined Usage in LINQ

The `var` keyword becomes essential when working with anonymous types from LINQ queries:

```
var results = from e in employees
 where e.Salary > 50000
 select new { e.Name, e.Department };
```

## 4. Important Limitations

### 1. Anonymous Type Limitations

- Cannot add methods/events
- Only contains read-only properties
- Cannot be passed as parameters (without using `dynamic`)

### 2. `var` Limitations

- Not the same as JavaScript `var` (still static typing)
- Doesn't work with `dynamic` types
- Can't be used in explicit interface implementation

## 5. Performance Considerations

### 1. Anonymous Types

- No runtime performance penalty
- Generated class has optimized Equals/GetHashCode
- Garbage collected like regular reference types

### 2. `var` Keyword

- Zero runtime impact (compile-time only)
- Doesn't affect generated IL
- No memory or CPU overhead

## MSDN References

- `var` Keyword
- Anonymous Types
- Type Inference

# LINQ (Language Integrated Query)

## 1. Introduction to LINQ

### Historical Context

- Introduced in C# 3.0 (.NET Framework 3.5, 2007), LINQ revolutionized data querying in .NET by bringing SQL-like syntax to C#.
- Provides a unified model for querying various data sources including collections, databases, XML, and more.

### Key Benefits

1. **Declarative Syntax:** Express what you want, not how to get it
2. **Type Safety:** Compile-time checking of queries
3. **IntelliSense Support:** Full IDE integration
4. **Standardized Patterns:** Consistent across data sources

## 2. Core LINQ Operators

### Filtering Operations

```
var highScores = students.Where(s => s.Score > 90);
var topStudent = students.FirstOrDefault(s => s.Score == 100);
```

### Projection Operations

```
var names = students.Select(s => s.Name);
var studentDetails = students.Select(s => new { s.Name, s.Age });
```

### Sorting Operations

```
var orderedStudents = students.OrderBy(s => s.Name);
var multiLevelSort = students.OrderBy(s => s.Grade).ThenByDescending(s => s.Score);
```

### Grouping Operations

```
var studentsByGrade = students.GroupBy(s => s.Grade);
var gradeAverages = students.GroupBy(s => s.Grade)
 .Select(g => new { Grade = g.Key, Avg = g.Average(s => s.Score) });
```

### Aggregation Operations

```
var totalScore = students.Sum(s => s.Score);
var averageAge = students.Average(s => s.Age);
var maxScore = students.Max(s => s.Score);
```

## 3. Query Syntax vs Method Syntax

### Query Syntax (SQL-like)

```
var results = from s in students
 where s.Score > 80
 orderby s.Name
 select new { s.Name, s.Score };
```

### Method Syntax (Lambda-based)

```
var results = students
 .Where(s => s.Score > 80)
 .OrderBy(s => s.Name)
 .Select(s => new { s.Name, s.Score });
```

### When to Use Each

1. **Query Syntax:** Better for joins, complex queries
  2. **Method Syntax:** Better for method chaining, simple queries
  3. **Mixed Syntax:** Can combine both styles
- 4. Deferred Execution**

#### Lazy Evaluation Concept

- Queries aren't executed until enumerated
- Enables query composition and optimization
- Can lead to multiple enumerations if not careful

#### Immediate Execution Methods

```
foreach(var s in students) // enumeration
 System.Console.WriteLine(s);

var list = students.ToList(); // Forces execution
var array = students.ToArray();
var count = students.Count();
```

## 5. Performance Considerations

### Optimization Techniques

1. Add `.AsParallel()` for CPU-bound operations
2. Use `.Any()` instead of `.Count() > 0` for existence checks
3. Consider `.ToLookup()` for repeated key-based access
4. Pre-size collections when possible with `.ToList(capacity)`

### Common Pitfalls

1. N+1 queries in nested iterations
2. Multiple enumerations of same query
3. Unnecessary sorting operations
4. Inefficient joins on large collections

## 6. Advanced LINQ Patterns

### Joins Between Collections

```
var studentCourses = from s in students
 join c in courses on s.CourseId equals c.Id
 select new { s.Name, c.CourseName };
```

## Partitioning Operations

```
var firstPage = students.Take(20);
var secondPage = students.Skip(20).Take(20);
```

## Set Operations

```
var distinctAges = students.Select(s => s.Age).Distinct();
var commonStudents = classA.Intersect(classB);
```

## 7. Best Practices

### Do's

1. **Chain methods properly** (filter first, then project)
2. **Use meaningful variable names** in queries
3. **Consider readability** over cleverness
4. **Profile performance** of complex queries

### Don'ts

1. **Don't misuse LINQ** for complex business logic
2. **Avoid deep nesting** of queries
3. **Don't ignore deferred execution** implications
4. **Don't mix LINQ with side effects**

## MSDN References

- [LINQ Overview](#)

- Standard Query Operators
- LINQ Performance

## File and Stream I/O

### 1. Fundamental Concepts

- The System.IO namespace has been a core part of .NET since version 1.0 (2002), providing comprehensive APIs for file system operations and stream-based I/O.
  - .NET Framework 2.0 (2005): Added FileSystemWatcher and improved serialization
  - .NET Framework 4.0 (2010): Introduced memory-mapped files
  - .NET Core 3.0 (2019): Added high-performance System.IO.Pipelines
  - .NET 6 (2021): Improved async file operations

#### 1.1 Core Namespaces

- `System.IO`: Basic file and directory operations
- `System.Text`: Encoding/decoding text streams
- `System.IO.Compression`: For ZIP/GZIP handling

#### 1.2 Stream Based Architecture

- The .NET I/O system is built on a stream-based model that provides:
  - **Abstraction layer** over various storage mediums
  - **Uniform interface** for sequential byte access
  - **Buffering capabilities** for performance optimization

## 2. File System Operations

- The File and Directory classes provide static methods for common operations:
  - File: Create, copy, delete, move, and open files
  - Directory: Create, move, and enumerate directories

- Path: Cross-platform path manipulation methods

## 2.1 File Class (Static Methods)

```
// Basic operations
File.WriteAllText("data.txt", "Hello World\n");
File.AppendAllText("data.txt", "Bye World\n");
string content = File.ReadAllText("data.txt");
bool exists = File.Exists("data.txt");

// Advanced scenarios
File.Copy("source.txt", "dest.txt", overwrite: true);
File.SetAttributes("data.txt", FileAttributes.Hidden);
```

## 2.2 Directory Management

```
// Directory operations
Directory.CreateDirectory("logs");
var files = Directory.EnumerateFiles("docs", "*.*");

// Special folders
string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

## 3. Stream-Based Operations

### 3.1 Core Stream Classes

Class	Purpose
FileStream	Physical file access

Class	Purpose
MemoryStream	In-memory byte storage
BufferedStream	Performance optimization
NetworkStream	Network communication
CryptoStream	For encryption/decryption

### 3.2 Traditional Synchronous IO

- Traditional blocking operations suitable for:
  - Small files
  - Non-UI applications
  - Simple scripting scenarios

### 3.3 Write/Read Binary Files

```
// Writing to file
using (FileStream fs = File.Create("data.bin"))
using (BinaryWriter writer = new BinaryWriter(fs))
{
 writer.Write(42);
 writer.Write(3.14);
}

// Reading from file
using (FileStream fs = File.OpenRead("data.bin"))
using (BinaryReader reader = new BinaryReader(fs))
{
 int number = reader.ReadInt32();
 double pi = reader.ReadDouble();
}
```

### 3.4 Write/Read Text Files

```
// Writing text
using (StreamWriter writer = new StreamWriter("log.txt", append: true))
{
 writer.WriteLine($"{DateTime.Now}: Application started");
}

// Reading lines
using (StreamReader reader = new StreamReader("data.csv"))
{
 while (!reader.EndOfStream)
 {
 string line = reader.ReadLine();
 // Process line
 }
}
```

### 3.5 Stream Processing Patterns

#### A. Basic Stream Usage

- The standard disposable pattern for streams ensures:
  - Proper resource cleanup
  - Exception safety
  - Buffer management

#### B. Buffering Strategies

- Critical for performance optimization:

- Default buffer sizes (4KB typically)
- Custom buffering for large files
- Flush considerations for writers
- Example:

```
using (FileStream fileStream = new FileStream("output.txt", FileMode.Create)) {
 using (BufferedStream bufferedStream = new BufferedStream(fileStream)) {
 using (StreamWriter streamWriter = new StreamWriter(bufferedStream))
 {
 streamWriter.WriteLine("First line of text.");
 streamWriter.WriteLine("Second line of text.");
 }
 }
}
```

#### C. Position and Seeking

- Random access capabilities:
  - Position property tracking
  - Seek() for arbitrary access
  - Length monitoring

### 4. Serialization and Formatters

- Serialization is converting each given object into sequence of bytes.
- This sequence of bytes can be written into any stream e.g. FileStream, NetworkStream, etc.
- Deserialization is converting sequence of bytes back to the object.

#### 4.1 Binary Serialization

- [Serializable] attribute enables:

- Object graph preservation
  - Type fidelity
  - Compact binary representation
- [NonSerialized] attribute on field:
    - makes that field non serialized (like transient in Java).
- Note:
    - BinaryFormatter is deprecated as Deserialization is insecure (may cause attack).
    - Removed in .NET 9. Needs to be enabled explicitly in .NET 8 project settings (still with a warning).
      - <EnableUnsafeBinaryFormatterSerialization>true</EnableUnsafeBinaryFormatterSerialization>
- Example:

```
[Serializable]
public class Person { /* ... */ }

// Serialize
BinaryFormatter formatter = new BinaryFormatter();
using (FileStream fs = File.Create("person.bin"))
{
 formatter.Serialize(fs, new Person());
}

// Deserialize
using (FileStream fs = File.OpenRead("person.bin"))
{
 Person p = (Person)formatter.Deserialize(fs);
}
```

#### 4.2 XML Serialization

- XmlSerializer provides:
  - Human-readable output
  - Schema generation
  - Interoperability
- Example:

```
XmlSerializer serializer = new XmlSerializer(typeof(Person));

// Serialize to file
using (TextWriter writer = new StreamWriter("person.xml"))
{
 serializer.Serialize(writer, new Person());
}

// Deserialize from file
using (TextReader reader = new StreamReader("person.xml"))
{
 Person p = (Person)serializer.Deserialize(reader);
}
```

#### 4.3 JSON Serialization (System.Text.Json)

- Modern default choice with:
  - System.Text.Json (high performance)
  - Newtonsoft.Json (rich features)
  - Async streaming support
- Example:

```
// Serialize
string json = JsonSerializer.Serialize(new Person());
File.WriteAllText("person.json", json);

// Deserialize
string jsonText = File.ReadAllText("person.json");
Person p = JsonSerializer.Deserialize<Person>(jsonText);
```

## 5. Advanced Scenarios

### 5.1 Asynchronous I/O

- Modern non-blocking pattern offering:
  - Better scalability
  - UI responsiveness
  - Efficient resource utilization
- Example:

```
async Task ProcessFileAsync() {
 using (StreamReader reader = new StreamReader("largefile.txt")) {
 string content = await reader.ReadToEndAsync();
 // Process content
 }
}
```

### 5.2 File System Watcher

- FileSystemWatcher provides events for:
  - File creations/modifications
  - Directory changes

- Rename operations
- Example:

```
FileSystemWatcher watcher = new FileSystemWatcher("C:\\\\WatchFolder");
watcher.Created += (s, e) => Console.WriteLine($"Created: {e.Name}");
watcher.EnableRaisingEvents = true;
```

### 6.3 Memory-Mapped Files

- For extremely large files:
  - Efficient random access
  - Shared memory between processes
  - Native OS integration
- Example:

```
using (var mmf = MemoryMappedFile.CreateFromFile("large.bin"))
using (var accessor = mmf.CreateViewAccessor()) {
 int value = accessor.ReadInt32(position: 0);
}
```

## 6. Best Practices

### 6.1 Resource Management

1. **Always dispose streams** (use `using` blocks)
2. **Flush writers** when needed (or use auto-flush)
3. **Handle exceptions** (`FileNotFoundException`, `UnauthorizedAccessException`)

## 6.2 Performance Considerations

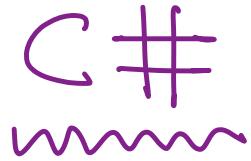
1. **Buffer sizes**: Optimal defaults exist (typically 4KB)
2. **Async vs Sync**: Use async for UI apps and services
3. **File sharing**: Consider [FileShare](#) modes for concurrent access

## 6.3 Security

1. **Validate file paths** (avoid path traversal attacks)
2. **Handle sensitive data** carefully in memory
3. **Set proper permissions** when creating files

## MSDN References

- [File and Stream I/O](#)
- [System.Text.Json](#)
- [Asynchronous File Access](#)



# C#.NET @ Sunbeam Infotech

***Trainer: Nilesh Ghule***



# File IO

• .Net IO → based on Streams

Stream → flow of bytes

↳ obj in which data can be written  
or data can be read from.

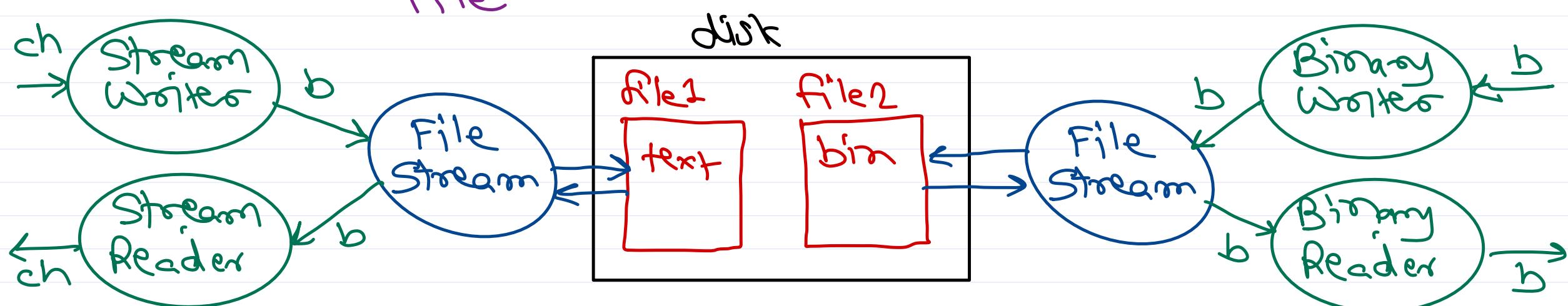
↳ FileStream, NetworkStream, CryptoStream

↳ Disk File

↳ Socket

↳ encrypt/decrypt

↳





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

A screenshot of a code editor showing a file named 'temp.cs'. The code is a list of numbered comments. Red annotations have been added to explain partial classes and methods.

```
1 # .NET
2
3 ### **Partial classes** ...
55
56 ### **Partial Methods** ...
94
95 ### **`static` Keyword** ...
417
418 ### **Nullable Types** ...
571
572 ### **var and Anonymous Types** ...
704
705 ### **LINQ (Language Integrated Query)** ...
863
864 ### **File and Stream I/O** ...
1172
1173 ### **Exception Handling** ...
1387
1388 ### **Resource Management & Garbage Collection** ...
1528
1529 ### **Interfaces Advanced Features** ...
1667
```

**partial class MyClass {**  
 **partial void Fun();**  
 **// ...**  
**}**

**partial class MyClass {**  
 **void Fun() {**  
 **// ...**  
 **}**  
**}**

**MyClass obj = new MyClass();**  
**obj.Fun();**

A screenshot of the Microsoft Visual Studio IDE interface. The main window displays a C# code editor with two files open: Program.cs and 02\_Statics.StringUtil.cs. The code in Program.cs includes an extension method 'Display' for the MyClass class. The code in 02\_Statics.StringUtil.cs includes a static utility method 'SwapCase' for strings. A red annotation with the text '← Extension methods.' points to the 'Display' method in Program.cs. The status bar at the bottom shows the current file is 'Program.cs', line 36, character 15, with 36 total lines. The magnifier icon in the status bar indicates the code is being viewed at 100% zoom.

```
22 }
23 }
24 static class MyClassUtil
25 {
26 // extension method of the MyClass
27 // rule 1: the first arg must be object of MyClass with "this" keyword
28 // rule 2: must be static method of the static class (top-level)
29 public static void Display(this MyClass obj) ← Extension methods.
30 {
31 Console.WriteLine("MyClass.Display() called: " + obj.field);
32 }
33 }
34
35 // Common Practice: Utility extension methods are added into static helper classes
36 // and a separate class library is created -- which can be reused in multiple projects.
37 static class StringUtil
38 {
39 public static string SwapCase(this string s)
40 {
41 StringBuilder sb = new StringBuilder(s.Length);
42 foreach (char c in s)
43 {
44 if(Char.IsUpper(c))
```

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays a C# code editor for a file named `Program.cs`. The code defines a `Vector` class with properties `X` and `Y`, and a static method `operator+(Vector v1, Vector v2)` that returns a new `Vector` object with summed `X` and `Y` values. A comment at the top states `// - not all operators can be overloaded.`

Annotations in red text and blue boxes are overlaid on the code:

- A blue box highlights the call `c = Vector.operator+(a, b)`.
- Red text below it reads `Main ->`.
- More red text follows:
  - `Vector a = new Vector() { ... };`
  - `Vector b = new Vector() { ... };`
  - `Vector c = a + b;`
  - `cw(c);`

The status bar at the bottom shows the current line (Ln: 72), column (Ch: 24), and other system information like battery level and date (28-06-2025).

```
// - not all operators can be overloaded.
class Vector
{
 public int X { get; set; }
 public int Y { get; set; }
 public static Vector operator+(Vector v1, Vector v2)
 {
 Vector res = new Vector()
 {
 X = v1.X + v2.X,
 Y = v1.Y + v2.Y
 };
 return res;
 }
 public override string ToString()
 {
 }
}
```

When any class implements `IAddable<T>` interface, it must have `operator+()` implemented.

For this, `operator+()` declared `abstract static`.

```
53 }
54
55 2 references
56 interface IAddable<T> where T : IAddable<T>
57 {
58 2 references
59 abstract static T operator +(T v1, T v2);
60
61 // operator overloading in C# class/struct
62 // - must be static method named as operator(sign). "abstract static"
63 // - number of args must be as per the operator requirement i.e. unary, binary.
64 // - not all operators can be overloaded.
65 15 references
66 class Vector : IAddable<Vector>
67 {
68 8 references
69 public int X { get; set; }
70 8 references
71 public int Y { get; set; }
72 2 references
73 public static Vector operator+(Vector v1, Vector v2) ✓
```

## Design Pattern -- Common solutions to Well-known problems : OO development

- Problem: Global variable in Procedural Lang (like C).
  - Not allowed in OO lang (like C++, Java).
  - Need global vars so that it conserve resources e.g. ConnectionPool, Logger, ...
- Solution: Singleton Design Pattern
  - Can create single object of the class
  - Same object accessible throughout the program. |

The screenshot shows a Microsoft Visual Studio interface with a C# code editor. The code is as follows:

```
internal class Program
{
 static void Main(string[] args)
 {
 //Emp data --> empno, ename, job, mgr, hire, sal, comm, deptno
 // in database -- KING's mgr --> NULL
 // in database -- non-salesman emp's comm --> NULL
 }
}
```

Handwritten annotations in red text are overlaid on the code:

- @Entity -- java jpa entity**
- class Emp {**
- int empno;**
- String ename;**
- String job;**
- Double mgr; // wrapper class**
- ref can be null.**
- // boxing/unboxing**
- // ...**

The Visual Studio interface includes a toolbar, menu bar, and various tool windows like Error List and Output.

`int num = 99999999; ---> text fil: [8 chars] ---> bin file: [4 bytes]`

~~99999999~~

~~10101010~~

.Net IO → based on Streams

human readability

efficient store

& process.

Stream → flow of bytes

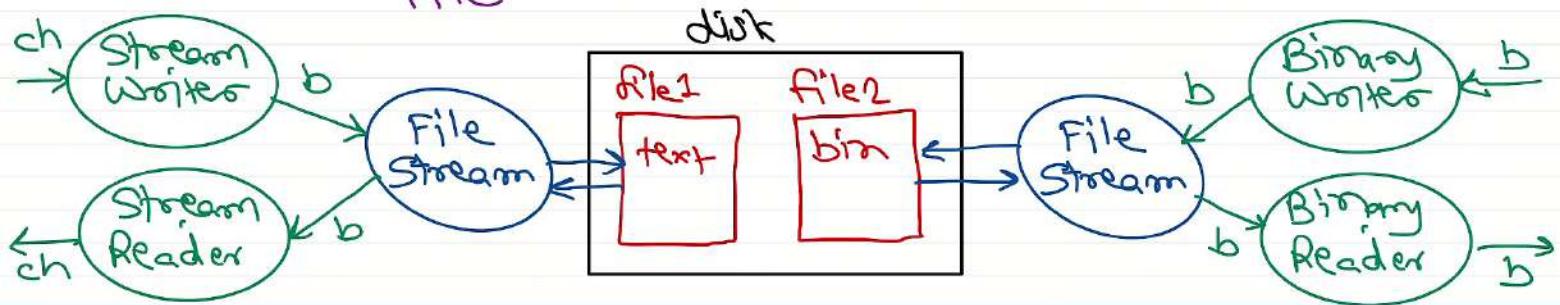
Stream → obj in which data can be written  
or data can be read from.

→ FileStream, NetworkStream, CryptoStream

Disk  
file

Socket

encrypt/decrypt



`int num = 99999999; ---> text fil: [8 chars] ---> bin file: [4 bytes]`

~~99999999~~

~~10101010~~

.Net IO → based on Streams

human readability

efficient store

& process.

Stream → flow of bytes

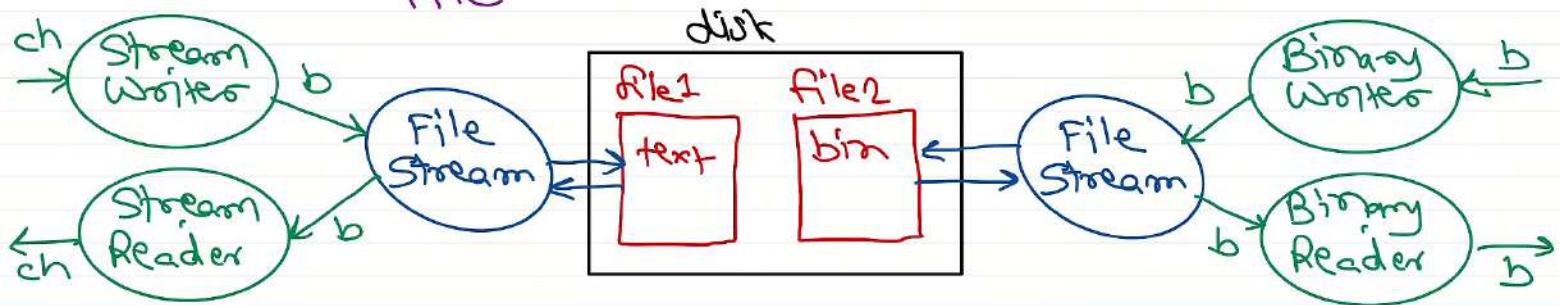
Stream → obj in which data can be written  
or data can be read from.

File Stream, Network Stream, Crypto Stream

Disk  
file

Socket

encrypt/decrypt



File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help | Search | Day05\_Soln

Program.cs | Program.cs | Program.cs | Program.cs | Main(string[] args)

```

72 fs.Close();
73 }
74 static void Main(string[] args)
75 {
76 // C# class -> obj serialize
77 // - [Serializable]
78 // - paramless public ctor
79 }
80 }
```

**Serialization in .NET - formatters**

1. BinaryFormatter
2. XmlSerializer
3. JsonSerializer
4. ...

```

graph LR
 CObj1[C# obj] --> BF[Binary Formatter]
 CObj2[C# obj] --> BF
 BF -- "binary format
metadata+data
(bytes)" --> FS[File Stream]
 FS -- "metadata+data
(bytes)" --> CObj3[C# obj]
 FS -- "metadata+data
(bytes)" --> F[File]
 F --> CObj4[C# obj]
```

binary format  
metadata+data  
(bytes)

Disk

File Stream

File

C# obj

Binary Formatter

C# obj

File Stream

File

Item(s) Saved

96% 0/0 4 Error List Output 11 0/0 24 88 main dotnet 13:22 28-06-2025 ENG US

The screenshot shows a Microsoft Visual Studio interface with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Search Bar:** Day05\_Soln
- Toolbox:** Toolbox icon on the left.
- Toolbars:** Standard toolbar with icons for New, Open, Save, Print, etc.
- Project Explorer:** Shows a project named "05\_FileIO".
- Solution Explorer:** Shows multiple files: Program.cs, Program.cs, Program.cs, and Program.cs.
- Task List:** GitHub Copilot icon.
- Diagnostic Tools:** Diagnostic Tools icon.
- Properties:** Properties icon.
- Split View:** Split View icon.
- Git Changes:** Git Changes icon.
- Code Editor:** The main window displays the following C# code:

```
0 references
76 static void Main(string[] args)
77 {
78 string path = "file3.bin";
79 Dept d1 = new Dept { DeptId = 10, Dname = "Accounting", Location = "New York" };
80
81 BinaryFormatter bf = new BinaryFormatter();
82 using (FileStream fs = new FileStream(path, FileMode.OpenOrCreate, FileAccess.Write))
83 {
84 list
85 bf.Serialize(fs, d1);
86 Console.WriteLine("Dept File Saved.");
87 } // fs closed
88
89 using (FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read))
90 {
91 List<Dept> list
92 Dept d2 = (Dept)bf.Deserialize(fs);
93 Console.WriteLine("Dept Read: " + d2);
94 } // fs closed
95 }
```

The code demonstrates the use of the `BinaryFormatter` class to serialize and deserialize objects of type `Dept` to a binary file named `file3.bin`.

**Status Bar:** 96%, 0/0, 24, main, dotnet, 13:41, ENG US, 28-06-2025.

# .NET

## Exception Handling

### 1. Introduction

- Exception handling has been a fundamental part of C#, providing a structured way to handle runtime errors.
- The .NET exception model is based on a hierarchy of exception classes, all deriving from System.Exception.

### 2. Core Exception Handling Constructs

#### 2.1 try-catch Blocks

- The fundamental structure for catching exceptions:

```
try {
 // Code that might throw exceptions
 File.ReadAllText("nonexistent.txt");
}
catch (FileNotFoundException ex) {
 Console.WriteLine($"File not found: {ex.FileName}");
}
catch (Exception ex) {
 Console.WriteLine($"General error: {ex.Message}");
}
```

#### 2.2 finally Block

- Guarantees execution for cleanup:

```
FileStream file = null;
try {
 file = File.Open("data.txt", FileMode.Open);
 // Process file
}
finally {
 file?.Dispose(); // Always executes
}
```

### 2.3 using Statement

- Simplified resource cleanup (implements IDisposable):

```
using (var file = File.Open("data.txt", FileMode.Open)) {
 // Automatically disposed when block exits
}
```

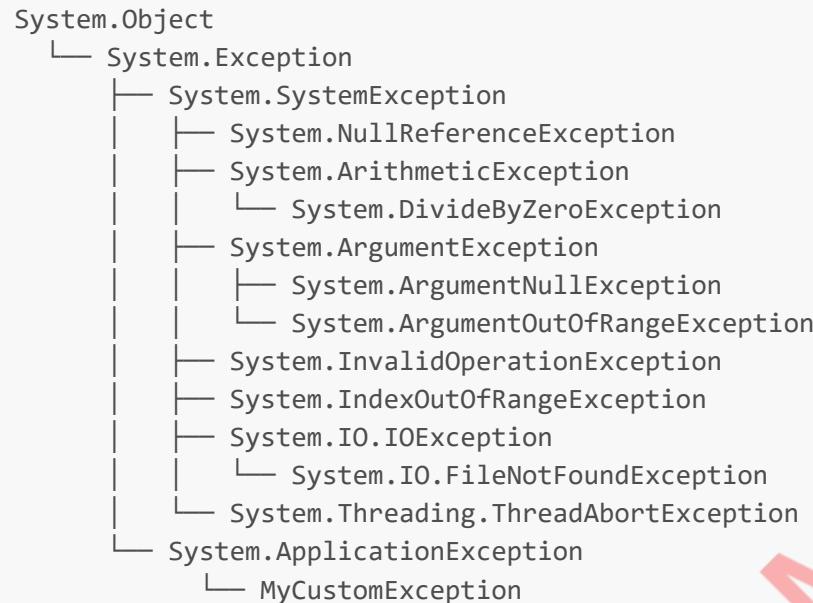
## 3. Exception Types Hierarchy

### 3.1 Common Exception Types

```
// System Exceptions
catch (ArgumentNullException ex) { ... }
catch (ArgumentException ex) { ... }
catch (InvalidOperationException ex) { ... }

// IO Exceptions
catch (FileNotFoundException ex) { ... }
catch (DirectoryNotFoundException ex) { ... }
```

```
// Custom Exceptions
catch (MyCustomException ex) { ... }
```



### 3.2 Creating Custom Exceptions

```
public class InventoryException : ApplicationException {
 public int ItemId { get; }
 public InventoryException(int itemId, string message) : base(message) {
 ItemId = itemId;
 }
}
```

```
public class InventoryManagement {
 // fields
 public InventoryItem GetItemDetails(int itemId) {
 // ...
 if(notFound)
 throw new InventoryException(itemId, "Item is not in stock");
 // ...
 }
}
```

```
// Usage
try {
 InventoryManagement im = new InventoryManagement();
 // ...
 InventoryItem item = im.GetItemDetails(id);
 // ...
}
catch(InventoryException e) {
 Console.WriteLine(e.ToString());
}
```

## 4. Advanced Exception Handling

### 4.1 Exception Filters (C# 6)

- Conditional catch blocks:

```
try { ... }
catch (HttpRequestException ex) when (ex.StatusCode == 404) {
```

```
 Console.WriteLine("Resource not found");
 }
```

#### 4.2 AggregateException

- Handling multiple exceptions:

```
try {
 Parallel.ForEach(items, ProcessItem);
}
catch (AggregateException ae) {
 foreach (var ex in ae.InnerExceptions) {
 Console.WriteLine(ex.Message);
 }
}
```

### 5. Best Practices

#### 5.1 Do's

```
// Catch specific exceptions
catch (SqlException ex) { ... }

// Provide meaningful messages
throw new InvalidOperationException("Connection pool exhausted");

// Log complete exception details
logger.LogError(ex, "Processing failed for {ItemId}", itemId);
```

## 5.2 Don'ts

```
// Avoid empty catch blocks
catch (Exception) { }

// Don't catch without action
catch (Exception ex) {
 throw; // Just rethrowing is usually pointless
}

// Avoid exception-based flow control -- This is not good practice
try {
 using (StreamReader reader = new StreamReader("your_file.txt")) {
 while (true) {
 string line = reader.ReadLine();
 if (line == null) // Alternative check for end of stream
 break;
 // Process the line
 }
 }
 catch (EndOfStreamException ex) {
 // Handle the end of stream condition
 Console.WriteLine("End of stream reached: " + ex.Message);
 }
}
```

## 6. Performance: Exception Costs

- Stack trace generation is expensive
- Avoid exceptions in normal flow
- Use Try-pattern for expected cases:

```
if (int.TryParse(input, out var number)) {
 // Success case
}
else {
 // Handle invalid input
}
```

## MSDN References

- Exception Handling
- Best Practices
- Creating Custom Exceptions

## Resource Management & Garbage Collection

### 1. Introduction

- The .NET runtime provides automatic memory management through its garbage collector (GC). The CLR memory management system handles:
  - Allocation of objects on the managed heap
  - Reclamation of unused memory
  - Compaction of surviving objects

### 2. Garbage Collection Fundamentals

#### 2.1 Generational Collection

- The heap is divided into generations:
  - **Generation 0:** Short-lived objects (most collections occur here)
  - **Generation 1:** Buffer between short-lived and long-lived objects
  - **Generation 2:** Long-lived objects
  - **Large Object Heap (LOH):** Objects > 85KB (collected less frequently)

## 2.2 Collection Triggers

- GC runs when:
  1. Generation 0 reaches its budget/threshold
  2. System memory is low
  3. AppDomain is unloading
  4. Explicitly called via `GC.Collect()`

## 2.3 Garbage Collection Process

1. Marking Phase: The GC identifies live objects by traversing the object graph starting from roots (like global variables and stack references). It marks reachable objects as "live" and identifies unreachable objects as candidates for collection.
2. Relocating Phase (optional): If the GC determines there's significant fragmentation in the heap, it updates references of live objects to new compacted locations before the compacting phase.
3. Compacting Phase: The GC reclaims the memory occupied by dead objects and compacts the remaining live objects into a contiguous block of memory. This reduces fragmentation and improves memory locality.

## 2.4 GC Flavors

- **Workstation GC:** Optimized for UI apps (lower latency)
- **Server GC:** Optimized for throughput (multiple CPU cores)

## 3. Deterministic Resource Cleanup

### 3.1 IDisposable Interface

- It's an interface in the System namespace.

```
public interface IDisposable {
 void Dispose();
}
```

- It contains a single method: Dispose().
- Implementing IDisposable signals that a class holds resources that need explicit cleanup.

### 3.2 Standard Disposable Pattern

```
public class Resource : IDisposable {
 private bool _disposed = false;
 public void Dispose()
 {
 // Dispose managed resources
 _disposed = true;
 GC.SuppressFinalize(this);
 }
 ~Resource() {
 if(!_disposed)
 Dispose();
 }
}
```

### 3.3 Usage Patterns

```
// Explicit disposal
var resource = new Resource();
try { /* use resource */ }
finally { resource.Dispose(); }
```

```
// Using statement (preferred)
using (var resource = new Resource()) {
 // Automatic disposal
}
```

## 4. Finalization (Destructors)

### 4.1 Finalizer Syntax

```
public class Resource {
 ~Resource() { // Finalizer
 // Cleanup unmanaged resources
 }
}
```

### 4.2 Finalization Process

1. Object becomes unreachable
2. GC queues object for finalization
3. Finalizer thread calls finalizer
4. Memory reclaimed in next GC cycle

### 4.3 Performance Impact

- Finalizable objects survive first collection
- Require additional GC work
- Add latency to cleanup process

## 5. Best Practices

### 5.1 Resource Management

1. Implement `IDisposable` for types holding:
  - File handles

- Database connections
  - Native resources
2. Prefer `using` over manual `try-finally`
3. Avoid finalizers unless absolutely necessary

## 5.2 GC Optimization

1. Minimize large object allocations
2. Consider object pooling for frequent allocations
3. Use `structs` for small, short-lived data
4. Avoid unnecessary references

## 5.3 Anti-Patterns

1. Calling `GC.Collect()` explicitly
2. Empty finalizers
3. Resurrecting (reuse/reassign) objects in finalizers
4. Overusing object pinning (`fixed` keyword)

## MSDN References

- Garbage Collection
- IDisposable Pattern
- Memory Management

## Indexers and Iterators

### 1. Indexers

#### Concept

- Indexers allow objects to be indexed like arrays, providing a way to access elements using the `[]` notation. They are essentially properties named "this" that take parameters.

**Example**

```
// consider a user defined stack of strings
public class DemoStack {
 // ...
 public string this[int index] {
 get {
 if (index < 0 || index > _top)
 throw new IndexOutOfRangeException();
 return _items[index];
 }
 set {
 if (index < 0 || index > _top)
 throw new IndexOutOfRangeException();
 _items[index] = value;
 }
 }
}
```

**Usage**

```
var stack = new DemoStack<string>(5);
stack.Push("First");
stack.Push("Second");
Console.WriteLine(stack[0]); // Accesses "First"
stack[1] = "Updated"; // Modifies "Second"
```

**2. Iterators****Concept**

- Iterators provide a way to traverse collections using `foreach` by implementing `IEnumerable` or `IEnumerable<T>`. The `yield` keyword simplifies iterator implementation.

#### Implementation Example

```
public class DemoStack : IEnumerable {
 // ...
 public IEnumerator GetEnumerator() {
 for (int i = _top; i >= 0; i--) {
 yield return _items[i]; // Lazy evaluation
 }
 }
}
```

#### Usage

```
foreach (var item in stack) {
 Console.WriteLine(item); // Prints from top to bottom
}
```

### 3. Key Features

#### Indexer Characteristics

- Can be overloaded with different parameter types
- Can have multiple dimensions (`[x,y]`)
- Can be defined in interfaces
- Support get-only or set-only accessors

## Iterator Characteristics

1. `yield return` provides lazy evaluation
2. Maintains state between iterations
3. Can implement complex traversal logic
4. Compiler generates state machine

## 4. Best Practices

### 1. Indexers:

- Validate index parameters
- Consider performance for large collections
- Document expected behavior

### 2. Iterators:

- Prefer `IEnumerable<T>` over `IEnumerable`
- Avoid modifying collections during iteration
- Consider thread safety

## MSDN References

- [Indexers \(C#\)](#)
- [Iterators \(C#\)](#)
- [yield \(C# Reference\)](#)

## Interfaces Advanced Features

### 1. Default Interface Methods (C# 8.0+)

#### Introduction

- Traditionally purely abstract, interfaces gained implementation capability in C# 8.0 (.NET Core 3.0, 2019) to support API evolution without breaking changes.

## Key Characteristics

1. **Backward Compatibility:** Add functionality without forcing implementation
2. **Multiple Inheritance:** Classes can inherit multiple default implementations
3. **Explicit Invocation:** Default methods must be called through interface reference

```
public interface ILogger {
 void Log(string message); // Traditional abstract method

 // Default implementation
 void LogError(string error) {
 Log($"ERROR: {error}");
 }
}

class ConsoleLogger : ILogger
{
 public void Log(string message) => Console.WriteLine(message);
 // No need to implement LogError
}

// Usage:
ILogger logger = new ConsoleLogger();
logger.LogError("Something failed"); // Uses default implementation
```

## 2. Static Abstract Members (C# 11+)

### Generic Math Support

- Enables abstract static methods in interfaces, primarily for numeric scenarios:

```
public interface IAddable<T> where T : IAddable<T>
{
 static abstract T operator +(T left, T right);
 static virtual T Zero => default;
}

public struct Point : IAddable<Point>
{
 public int X, Y;

 public static Point operator +(Point left, Point right) =>
 new Point { X = left.X + right.X, Y = left.Y + right.Y };

 public static Point Zero => new Point();
}
```

#### Key Benefits

1. **Type-Safe Operator Overloading**
  2. **Generic Algorithm Support**
  3. **Numerics Without Boxing**
- 3. Version-Resilient Interfaces**

1. Add new methods as defaults
2. Never remove existing methods
3. Use extension methods for utility functions

**4. Interface Full Member Support**

- Interfaces can declare all member types:

```
public interface IObservable {
 event EventHandler Changed;
 string Name { get; set; }
 int Id { get; }
}
```

- All these must be implemented in derived classes.

## 5. Diamond Inheritance Resolution

- Default methods handle multiple inheritance cases:
- Example1:

```
class Intf1 {
 public void Fun() {
 Console.WriteLine("Intf1.Fun() called.");
 }
}
class Intf2 {
 public void Fun() {
 Console.WriteLine("Intf2.Fun() called.");
 }
}
class MyClass : Intf1, Intf2 {

}

// Usage : e.g. in Main()
Intf1 obj1 = new MyClass();
obj1.Fun(); // Intf1.Fun() called.

MyClass obj = new MyClass();
obj.Fun(); // Compiler error
```

- Example2:

```
interface IA { void M() => Console.WriteLine("A"); }
interface IB : IA { void IA.M() => Console.WriteLine("B"); }
interface IC : IA { void IA.M() => Console.WriteLine("C"); }
class D : IB, IC
{
 // Must provide implementation to resolve ambiguity - Otherwise compiler error
 void IA.M() => Console.WriteLine("D");
}
```

## 5. Best Practices

1. **Small, Focused Interfaces (ISP)**
2. **Default Methods for Backward Compatibility**
3. **Explicit Implementation for Clarity**
4. **Avoid State in Interfaces**
5. **Prefer Abstract Classes for Common Implementation**

## MSDN References

- Default Interface Methods
- Static Abstract Members
- Interface Design Guidelines

## dynamic Keyword

### 1. Introduction

- Introduced in **.NET Framework 4.0 (2010)** alongside C# 4.0 to simplify interoperability with **dynamic languages** (e.g., Python, JavaScript) and **COM objects**.

- Before `dynamic`, developers used **reflection** or explicit casting, which was verbose and error-prone.
- The `dynamic` keyword enables **late binding**, where type resolution happens at **runtime** instead of compile-time.
- Part of the **Dynamic Language Runtime (DLR)**, which sits atop the CLR to support dynamic operations.

## 2. Key Concepts & Definitions

- **Dynamic Typing:** The type of a `dynamic` variable is resolved at runtime, bypassing compile-time checks.
- **RuntimeBinder:** The component that resolves member invocations on `dynamic` objects (throws `RuntimeBinderException` on failures).
- **IDynamicMetaObjectProvider:** Interface used for custom dynamic behavior (e.g., `ExpandoObject`, `DynamicObject`).
- **DLR (Dynamic Language Runtime):** A layer that provides services for dynamic languages in .NET (e.g., caching call sites).
- **Use Cases:**
  - COM interop (e.g., Microsoft Office automation).
  - Consuming REST APIs with unpredictable schemas.
  - Dynamic data structures (e.g., `ExpandoObject`).

## 3. Advantages

- **Flexibility:** Simplifies interaction with weakly-typed systems (e.g., JSON, COM).
- **Reduces Boilerplate:** Avoids complex reflection code.
- **Improves Readability:** Cleaner syntax for dynamic operations.

## 4. Disadvantages

- **No Compile-Time Safety:** Errors (e.g., missing methods) only surface at runtime.
- **Performance Overhead:** Dynamic dispatch is slower than static typing due to runtime resolution.
- **Tooling Limitations:** IDE features like IntelliSense don't work for `dynamic` types.

## 5. How `dynamic` Differs from `var` and `object`

- `var`: Still **statically typed** (type inferred at compile-time).
- `object`: Requires explicit casting and offers no dynamic dispatch.
- `dynamic`: Defers all type checks to runtime.

## 6. Examples

```
// Example 1: Basic dynamic usage
dynamic obj = GetExternalData(); // Could be JSON/COM
Console.WriteLine(obj.Name); // Resolved at runtime
```

```
// Example 2: ExpandoObject (dynamic dictionary)
dynamic person = new ExpandoObject();
person.Name = "Alice";
person.Age = 30;
```

```
// Example 3: COM Interop (e.g., Excel)
dynamic excel = Microsoft.Office.Interop.Excel;
excel.Application app = new excel.Application();
```

## 7. RuntimeBinderException

- Thrown when a member (method/property) doesn't exist at runtime.
- Example:

```
dynamic value = "hello";
value.Foo(); // RuntimeBinderException: 'string' does not contain 'Foo'
```

## 8. Performance Considerations

- Avoid `dynamic` in performance-critical paths (e.g., tight loops).

- Cache dynamic calls if repeated (e.g., via `Func<dynamic, object>` delegates).

## MSDN References

- [dynamic \(C# Reference\)](#)
  - [DynamicObject Class](#)
  - [DLR Overview](#)
- 

## Reflection

### 1. Introduction

- Reflection provides the ability to inspect and interact with type information at runtime. This powerful feature enables:
  - Dynamic type discovery
  - Late binding
  - Runtime code analysis
  - Self-modifying applications

### 2. Core Concepts

#### 2.1 Type Metadata

- Type metadata is binary information that describes your type (class, struct, enum, ...). It's stored alongside the compiled code (IL) in assemblies.
- **Type Metadata**
  - Name: The fully qualified name of the type.
  - Visibility: Whether the type is public, private, etc.
  - Base Class: The type that the current type inherits from.
  - Interfaces: Interfaces implemented by the type.
  - Members: Information about the methods, fields, properties, events, and nested types within the type.
  - Attributes/Flags: Additional descriptive information about the type e.g. `IsAbstract`, `IsSealed`, etc.
- **Member Metadata**
  - Name: The name of the member (e.g., method name, field name).

- Type: The data type of the member (e.g., int, string, a custom type).
- Visibility: Whether the member is public, private, etc.
- Attributes: Additional descriptive information about the member.
- This metadata is loaded at runtime (when assembly is loaded).

## 2.2 Type Discovery

- The `System.Type` object holds type metadata.
- It can be accessed in one of the following ways:

```
Type classType = typeof(ClassName); // Compile-time known type
Type objType = someObject.GetType(); // Runtime type
Type typeByName = Type.GetType("System.Int32"); // By name
```

## 2.3 Assembly Inspection

- Information from current assembly can be accessed as follows:

```
Assembly assembly = Assembly.GetExecutingAssembly();
foreach (Type type in assembly.GetTypes())
 Console.WriteLine(type.FullName);
```

- An assembly can be loaded explicitly as follows:

```
Assembly assembly = Assembly.LoadFrom(@"AssemblyPath");
foreach (Type type in assembly.GetTypes())
 Console.WriteLine(type.FullName);
```

### 3. Member Inspection

#### 3.1 Exploring Members

```
Type type = typeof(MyClass);

// Get all public methods
MethodInfo[] methods = type.GetMethods();

// Get specific property
 PropertyInfo[] props = type.GetProperties();

// Get all fields
 FieldInfo[] field = type.GetFields(BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |
 BindingFlags.Static);

// Get field including non-public
 FieldInfo field = type.GetField("_internal", BindingFlags.NonPublic | BindingFlags.Instance);
```

#### 3.2 Dynamic Object Creation

```
object obj = Activator.CreateInstance(type);
```

#### 3.2 Method Invocation

```
// static method invoked without object
MethodInfo method = typeof(Math).GetMethod("Max", new[] { typeof(int), typeof(int) });
object result = method.Invoke(null, new object[] { 5, 10 }); // Static method call
```

```
// non-static methods need object (usually dynamically created) as first arg to method.Invoke()
object result = method.Invoke(obj, new object[] { ... }); // Non-Static method call
```

## 4. Performance Considerations

### 4.1 Caching Strategies

```
// Cache expensive reflection operations
private static readonly MethodInfo _toStringMethod = typeof(object)
 .GetMethod("ToString");

// Reuse cached method
string result = (string)_toStringMethod.Invoke(obj, null);
```

### 4.2 Alternatives

- **Compiled Expressions:** Faster than pure reflection

```
Func<object, string> toString = obj => obj.ToString(); // Faster alternative
```

## MSDN References

- Reflection in .NET
- Type Class
- Runtime Type Handling

## Unsafe Code and P/Invoke

## 1. unsafe Code Basics

### Concept

- The `unsafe` keyword allows pointer operations and direct memory access in C#. Requires compiling with `/unsafe` flag.

### Minimal Example

```
unsafe void PointerDemo()
{
 int value = 10;
 int* ptr = &value; // Pointer declaration
 Console.WriteLine(*ptr); // Dereference pointer
}
```

## 2. fixed Keyword

### Concept

- Pins managed objects in memory to prevent GC relocation during pointer operations.

### Minimal Example

```
unsafe void FixedDemo()
{
 int[] numbers = { 10, 20, 30 };
 fixed (int* ptr = numbers)
 {
 Console.WriteLine(ptr[1]); // Access array via pointer
 }
}
```

### 3. P/Invoke (Platform Invoke)

#### Concept

- Calls native functions from System DLLs. Requires `DllImport` attribute.

#### Minimal Example

```
using System.Runtime.InteropServices;

class NativeMethods
{
 [DllImport("user32.dll")]
 public static extern int MessageBox(IntPtr hWnd, string text, string caption, int type);
}

// Usage:
NativeMethods.MessageBox(IntPtr.Zero, "Hello", "Message", 0);
```

#### Key Notes

##### 1. Requirements:

- Enable "Allow unsafe code" in project settings
- Mark methods with `unsafe` keyword
- Use `fixed` when pointers reference managed objects

##### 2. Safety:

- These features bypass .NET safety checks
- Use only when absolutely necessary

- Validate all pointer operations

### 3. P/Invoke Tips:

- Match native types precisely (`int` vs `Int32`)
- Consider marshaling for complex types
- Use `CharSet` in `DllImport` for string handling

## MSDN References

- [Unsafe Code](#)
- [fixed Statement](#)
- [P/Invoke](#)

## String Class in C#

### 1. Introduction and Key Characteristics

- **Immutable**: Strings cannot be modified after creation (operations return new strings)
- **Reference Type**: Stored on the heap, but with special optimizations
- **Unicode Support**: UTF-16 encoded (2 bytes per character)
- **String Pool**: Runtime optimization for string literals

### 2. String Intern Pool

#### Concept

- Special memory area storing unique string literals
- Reuses identical strings to save memory
- Literals automatically interned at compile time

#### Intern Methods

```
string s1 = "hello";
string s2 = String.Intern(new StringBuilder().Append("he").Append("llo").ToString());
Console.WriteLine(ReferenceEquals(s1, s2)); // True - same reference

string s3 = String.IsInterned("hello") ?? "not interned"; // Check if interned
```

### 3. Commonly Used Methods

#### Basic Operations

```
string text = "Hello World";

// Length property
int len = text.Length; // 11

// Index access (read-only)
char first = text[0]; // 'H'

// Concatenation
string combined = String.Concat("Hello", " ", "World");
```

#### Searching

```
// Contains
bool hasWorld = text.Contains("World"); // true

// IndexOf
int index = text.IndexOf('W'); // 6
```

```
// StartsWith/EndsWith
bool starts = text.StartsWith("Hello"); // true
```

### Modification (Returns New Strings)

```
// Substring
string part = text.Substring(6, 5); // "World"

// Replace
string updated = text.Replace("World", "C#"); // "Hello C#"

// ToUpper/ToLower
string upper = text.ToUpper(); // "HELLO WORLD"

// Trim
string clean = " text ".Trim(); // "text"
```

### Splitting/Joining

```
// Split
string[] parts = "a,b,c".Split(','); // ["a", "b", "c"]

// Join
string joined = String.Join("-", parts); // "a-b-c"
```

## 4. String Comparison

```
// Culture-aware comparison
bool equal = String.Equals("hello", "HELLO", StringComparison.OrdinalIgnoreCase);

// Sorting order
int result = String.Compare("apple", "banana"); // -1
```

## 5. StringBuilder Introduction

### Purpose

- Mutable string buffer for efficient concatenation
- Avoids multiple allocations during string building

### Basic Usage

```
StringBuilder sb = new StringBuilder();
sb.Append("Hello");
sb.AppendLine(" World");
sb.AppendFormat("{0} times", 5);

string result = sb.ToString(); // "Hello World\n5 times"
```

### Key Features

- **Capacity Management:** Pre-allocate buffer size
- **Chained Methods:** `Append()`, `AppendLine()`
- **Thread Safety:** Not thread-safe by default

## 6. Performance Considerations

## 1. String Pool Benefits

- Reduces memory for duplicate literals
- Fast comparison via reference equality ([ReferenceEquals](#))

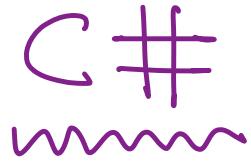
## 2. When to Use StringBuilder

- Multiple concatenations in a loop
- Building large strings incrementally
- When intermediate string results aren't needed

### MSDN References

- [String Class](#)
- [StringBuilder](#)
- [String Interning](#)

SUNBEAM INFOTECH



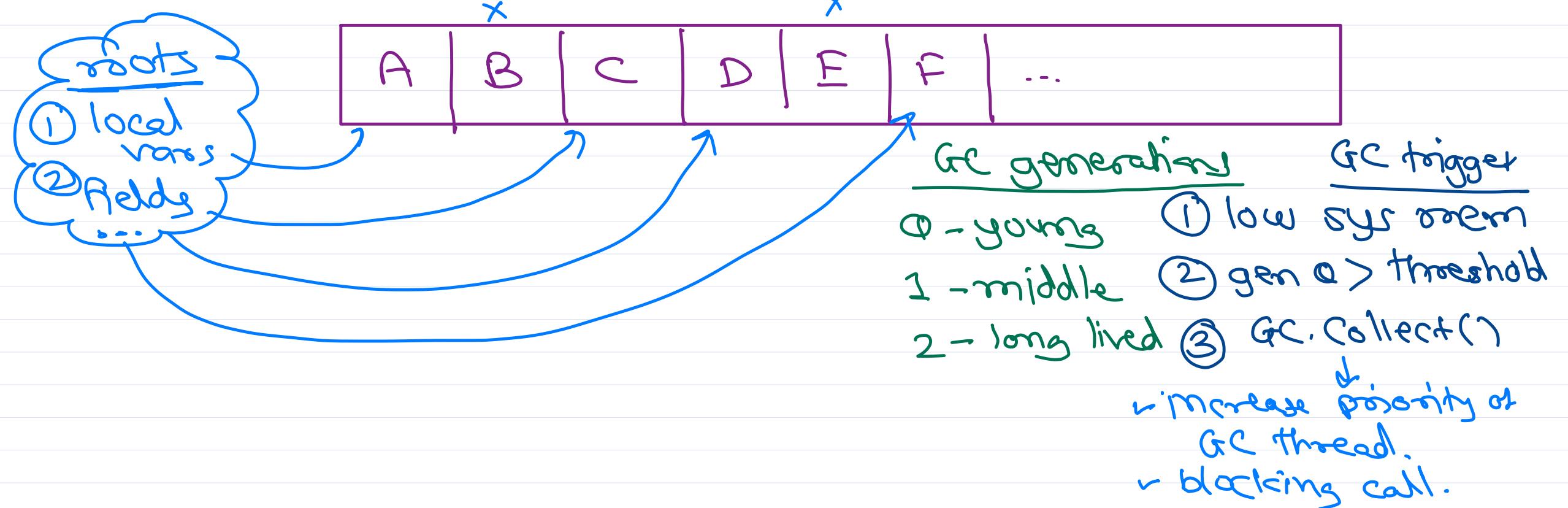
# C#.NET @ Sunbeam Infotech

***Trainer: Nilesh Ghule***



# Garbage Collection

CLR → Auto mem management  
Programmer → New obj allocation → heap. heap  
garbage collector → Auto mem release of unused objs  
mark + Finalize() + reclaim



# Metadata & Reflection

assembly

PE header

Manifest

Type  
Metadata

IL

Resources

Type object

- ① obj.GetType()
- ② typeof(type)
- ③ Type.GetType("cls");

Metadata of all types (public/non public)

For each type:

- ① name + type (class, struct, enum, interface, delegate).
- ② base class
- ③ interfaces
- ④ access specifier + flags  
(is abstract, is static, is sealed, ...)
- ⑤ Constructors
- ⑥ methods
- ⑦ properties
- ⑧ fields
- ⑨ custom attrs

↳ class  
members.

ConstructorInfo  
FieldInfo  
MethodInfo  
PropertyInfo

- member's metadata
- ① name + type
  - ② access specifier + flags (sealed, abstract, static, override, ...)
  - ③ params
    - ↳ ParameterInfo
      - ↳ name, type, flag
  - ④ return type
  - ⑤ ...





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

## Exceptions:

**Handling Keywords:** try, catch, finally  
one try may have multiple catch blocks.  
always handle specific exceptions &  
generic exception catch block at last.

**Throwing Keyword:** throw  
throw new Exception("...");

## Exception

- ArgumentException
- IOException
- InvalidOperationException
- ApplicationException
  - app specific exceptions

It is not mandatory to catch the exception. If exception occurred but not caught, it will be handled by CLR and application is terminated.

```
class InvalidIndexException:
 ApplicationException {
 private int _index;
 public InvalidIndexException(
 int index=-1){
 _index = index;
 }
 public int Index => _index;
 // ...
 }
 // may call super class
 // constructor --
 // throw --> Message.
 throw new InvalidIndexException(
 i);
```

The diagram shows a hand-drawn illustration overlaid on a screenshot of a Microsoft Visual Studio IDE. The code in the editor is:

```
24 public string Get(int index)
25 {
26 if (index < 0 || index > _top)
27 throw new InvalidIndexException(index);
28 return _arr[index];
29 }
30 public string _arr[int index]
31 {
32 get { return _arr[index]; }
33 set { _arr[index] = value; }
34 }
35
36 }
```

The illustration uses arrows to show pointer assignments:

- A pink arrow points from the variable `s[2]` to the `this` keyword in the `get` accessor.
- An orange arrow points from the `v = s[2];` assignment to the `_arr` variable in the `set` accessor.

The Solution Explorer on the right shows two projects: `01_Serialization` and `02_ExceptionIndexIterators`.

The screenshot shows a Microsoft Visual Studio interface with a C# code editor open. The code implements an iterator for prime numbers:

```
134 class Numbers : IEnumerable
135 {
136 public IEnumerator GetEnumerator()
137 {
138 yield return 1;
139 yield return 3;
140 yield return 5;
141 yield return 7;
142 yield return 11;
143 yield return 13;
144 }
145 }
146 static void Main(string[] args)
147 {
148 Numbers primes = new Numbers();
149 foreach(int num in primes)
```

Handwritten red annotations explain the use of `yield return`:

- "`yield return`" is easier way to implement iterator/enumerator in .net code.
- it can be used to traverse a collection with foreach loop.
- "`yield return`" internally creates an iterator which returns the current value but store the execution ctx.
- Each call to `MoveNext() + Current` (foreach loop internally does that) will provide access to next ele until the last elem.

Visual Studio status bar: Ln: 144 Ch: 14 SPC CRLF

## Classic OOP: Interfaces - Immutable

- No Interface versioning`

Add extra fns in existing interfaces without  
breaking old/existing applications.

- Traditionally purely abstract, interfaces gained implementation capability in C# 8.0 (.NET Core 3.0, 2019) to support API evolution without breaking changes.

New apps can take advantage of new fns.  
"default" methods

1. Backward Compatibility: Add functionality without forcing implementation  
2. Multiple Inheritance: Classes can inherit multiple default implementations  
3. Explicit Invocation: Default methods must be called through interface reference

added in interfaces (existing) without need that  
existing inheriting classes to implement them.

```
```csharp
public interface ILogger {
    void Log(string message); // Traditional abstract method
}
```



Q1. Which of the following best describes the relationship between CTS, CLR, and CLS in the .NET framework? A) CTS defines the rules for type safety across languages, enforced by the CLR, while CLS ensures only language-specific features are exposed for maximum compatibility. B) CTS is a component of CLS, which ensures all .NET languages follow CLR's runtime behavior. C) CLR is implemented in CTS and interacts with CLS to translate MSIL into unmanaged code. D) CLS is the execution engine of .NET, CTS is the memory manager, and CLR is a garbage collector designed for platform independence.

- Answer:- A

Q2. In the .NET ecosystem, which of the following statements correctly describes the difference between .exe and .dll assemblies? A) .exe files are executable programs with an entry point; .dll files are reusable libraries without single entry point. B) .dll files are standalone executables designed for user interfaces; .exe files are used only by background services. C) .dll files are the only format that can contain MSIL code in the .NET framework. D) .exe files can only be generated in unmanaged languages; .dll files are exclusive to .NET Core.

- Answer:- A

Q3. Which of the following sequences accurately represents the compilation and execution process of a C# program using the .NET Framework? A) C# Code → JVM → IL Code → CLR → Native Code → CPU B) C# Code → CLR → IL Code → Native Code → TI → OS C) C# Code → C# Compiler → IL → CLR → JIT → Native Code → CPU D) C# Code → CTS → CLS → JIT → JVM → Native

- Answer:- C

Q4. Which of the following type groupings under the Common Type System (CTS) is incorrectly paired? A) Value Type → Int32, Decimal, Char, Enum B) Reference Type → String, Object, Arrays, Interfaces C) Reference Type → Types stored on heap and accessed via address D) Value Type → Struct, Double, Boolean, class

- Answer:- D

Q5. Which of the following combinations of .NET Framework versions and their major features is correct? A) .NET Framework 2.0 introduced WPF and WCF, enabling rich UI development and service-oriented architecture. B) .NET Framework 3.5 added support for Generics, which allowed creation of type-safe collections and methods. C) .NET Framework 4.0 introduced Dynamic Language Runtime (DLR) and parallel programming support. D) .NET Framework 1.0 was the first version to support LINQ, Entity Framework, and Generics.

- Answer:- C

Q6. What will be the output of the following C# code?

```
static void Main(string[] args) {
    int X = 6, Y = 2;
    X *= X / Y;
    Console.WriteLine(X);
}
```

A) 12 B) 6 C) 18 D) Compile time error

- Answer:- C

Q7. Which of the following statements about objects in "C#" is correct? A) Everything you use in C# is an object, including Windows Forms and controls B) Objects have methods and events that allow them to perform actions C) All objects created from a class will occupy equal number of bytes in memory D) All of the mentioned

- Answer:- B

Q8. Consider the following statements about the public and internal access modifiers in C#. Which of the following options is correct? A) A public class is only accessible within the project's solution, while an internal class is accessible from other assemblies when marked with using. B) An internal class is accessible from any project that references its DLL, while a public class is restricted to the same assembly unless marked as static. C) A public class is accessible from anywhere including outside its assembly; an internal class is accessible only within the same assembly (project). D) Both public and internal classes are accessible across all DLLs, but public requires the class to be sealed and internal requires inheritance.

- Answer:- C

Q9. Which of the following best describes the working and behavior of the JIT compiler in the .NET runtime environment? A) JIT translates C# source code into Intermediate Language (IL), then into native code during installation to reduce startup overhead. B) JIT is a runtime feature of the CLR that compiles IL code to native code just before execution, ensuring efficient and platform-specific code execution. C) JIT compiles IL code to native code line-by-line at the moment of writing each line in Visual Studio, allowing live runtime previews. D) JIT directly interacts with the .NET assembly manifest to bypass IL code and execute metadata in a memory-optimized format via the CPU's execution engine.

- Answer:- B

Q10. Which of the following statements is correct about constructors in C#.NET? A) A constructor cannot be declared as private B) A constructor cannot be overloaded C) A constructor can be a static constructor D) None of the mentioned.

- Answer:- C

1. Calculate the number of bytes a structure variable "s" occupies in the memory (on 64-bit system) if it is defined as follows.

```
class trial {
    int i;
    Decimal d;
}
struct sample {
    private int x;
    private Single y;
    private trial z;
}
// in Main()
sample s = new sample();
```

- A. 24 bytes
- B. 8 bytes
- C. 16 bytes
- D. 12 bytes
- Answer: C

2. Which among the following cannot be used as a datatype for an enum in C#.NET?

1. short
2. double
3. byte
4. Object
5. bool
6. int

- A. 1,2,3,5
 - B. 2,4,5
 - C. 2,3,4,5
 - D. 1,2,4,6
 - Answer: B
-

Q3.Two namespaces n1 and n2 each contain a class named Demo. Which option correctly creates separate objects x and y from both classes:-

- A.

```
using n1;
using n2;

Demo x = new n1.Demo();
Demo y = new n2.Demo();
```

- B.

```
using n1;
using n2;

n1.Demo x = new Demo();
n2.Demo y = new Demo();
```

- C.

```
using Demo1 = n1.Demo;
using Demo2 = n2.Demo;

Demo1 x = new Demo1();
Demo2 y = new Demo2();
```

- D.

```
import n1;
import n2;

Demo x = new n1.Demo();
Demo y = new n2.Demo();
```

- Answer: C
-

4. Which among the following is a correct statement about namespace used in C#.NET?

- A. Classes must belong to a namespace, whereas structures need not.
 - B. All elements of the namespace must belong to one file
 - C. By default, namespace name is same a project name in Visual Studio.
 - D. All of the mentioned
 - Answer: C
-

5. Which class Greet() method will get called?

```
class Root {
    public new virtual void Greet() { }
}

class Middle : Root {
    public override void Greet() { }
}

class Leaf : Middle
{
}

class Program
{
    static void Main() {
        Root obj = new Leaf();
        obj.Greet();
    }
}
```

- A. Leaf
- B. Middle
- C. Root
- D. Compile-Time Error
- Answer: B

6. Which of the following statements about ref, out, and in parameters in .NET is incorrect?

- A. ref parameter provides two way communication between the caller and method.
- B. In ref as well as out parameter, method can return a value.

- C. An out argument must be assigned in the called method before returning, while a ref argument must be assigned in the calling method before passing.
- D. The ref and out keywords are interchangeable, as both allow bidirectional data flow between methods.

Answer: D

7. What will happen when OuterMethod() is called?

```
void OuterMethod() {
    int x = 10;

    static void InnerMethod() {
        Console.WriteLine(x);
    }

    InnerMethod();
}
```

- A. The code fails to compile because static local functions cannot capture local variables from the enclosing method.
- B. The code compiles and prints 10, because static local functions capture outer variables like lambdas.
- C. The code compiles, but throws a runtime error since static methods delay variable resolution until runtime.
- D. The code compiles and runs successfully by automatically promoting x to a static field in the enclosing class.
- Answer: A

8. A company, "Acme Corp," has multiple teams working on different parts of their e-commerce platform. Each team has developed classes, structs, and other code related to their specific area. Team A is responsible for product catalog management, Team B handles user accounts, and Team C manages shopping carts. How should the company best organize their code to avoid naming conflicts and ensure maintainability, particularly when these teams might eventually need to interact with each other's code?

- A. Place all classes, structs, and other code into a single global namespace called "AcmeCorp".
 - B. Create separate namespaces for each team, such as "AcmeCorp.ProductCatalog", "AcmeCorp.UserAccounts", and "AcmeCorp.ShoppingCart".
 - C. Use different access modifiers (private, public, etc.) to manage visibility between the teams' code.
 - D. Place all code into a single namespace and rely on unique class names within that namespace.
 - Answer: B
-

9. Choose the correct statement.

- A. The code inside #region will only compile Debug mode; it is skipped entirely in Release mode.
 - B. #region and #endregion define code that can be collapsed in the Visual Studio editor, but have no effect on compilation or runtime behavior.
 - C. #region automatically wraps the code inside a try-catch block to simplify debugging in the editor.
 - D. Using #region disables syntax errors inside the block until #endregion is reached, allowing partial compilation.
 - Answer: B
-

10. Why we use properties instead of simple getter and setter in c#?

- A. Properties are required when defining private fields and are only usable within the same class unless marked public
 - B. Properties improve performance by allowing the compiler to optimize field access at runtime and bypass method calls entirely.
 - C. Properties provide a familiar field-like syntax while offering access restrictions and input verification.
 - D. Properties enforce compile-time type safety in a way that getter and setter methods cannot, due to CLR-level enforcement rules.
 - Answer: C
-

1. Select the correct statement among the given statements?

- 1. C#.NET support partial implementation of interfaces.
 - 2. Properties could be declared inside an interface.
 - 3. Interface method calls are resolved at runtime using the actual object type.
 - 4. One interface can be implemented in another interface.
 - 5. Access specifiers that can be used for interface are public, protected and private.
- A. 1,2,4
 - B. 1,2,3
 - C. 2,3
 - D. 1,4,5
 - Answer: C
-

2. Choose the incorrect statement:

- A. To implement delegates necessary condition is class declaration.
 - B. A single delegate can invoke more than one method.
 - C. delegates could be shared.
 - D. delegate is a value type.
 - E. Delegates allow methods to be passed as parameters.
 - Answer: D
-

3. What will be the correct way to implement the interface in the following C# code?

```
interface person {  
    string name {  
        get;  
        set;  
    }  
}
```

```
    }  
}
```

- A.

```
class emp: person {  
    private string str;  
    public string name;  
    {  
        get { return str; }  
        set { str = value; }  
    }  
}
```

- B.

```
class emp: implements person  
{  
    private string str;  
    public string name  
    {  
        get { return str; }  
        set { str = value; }  
    }  
}
```

- C.

```
class emp: person {  
    private string str;
```

```
public string person.name
{
    get { return str; }
    set { str = value; }
}
```

- D: None of the mentioned
 - Answer: D
-

4. A class FileLogger needs to implement a contract that includes:

- A method: Log(string message)
 - A property: LogLevel { get; set; }
 - Support for multiple inheritance
 - No default implementation for members
 - Which of the following is the correct approach?
 - A. Use an abstract class with virtual methods and override them in FileLogger.
 - B. Use an interface with default method implementations and explicitly implement LogLevel.
 - C. Use an interface without default implementations and let FileLogger provide all members.
 - D. Use a static class with extension methods for Log().
 - Answer: C
-

5. What will happen when this code is executed?

```
public class Test {
    sealed int secret = 42;
```

```
public void Reveal() {
    Console.WriteLine(secret);
}

public class Program {
    public static void Main() {
        Test test = new Test();
        test.Reveal();
    }
}
```

A. outputs 42 B. Compile error: "sealed" cannot be applied to fields C. 0 D. Runtime error: sealed fields require initialization in constructor

- Answer: B
-

6. What is the correct output?

```
using System;
interface IJump {
    void Jump();
}
abstract class Animal {
    public virtual void Jump() {
        Console.Write("Animal jumps ");
    }
    public abstract void Sound();
}
class Dog : Animal, IJump {
    public override void Jump() {
        Console.Write("Dog jumps ");
    }
    public override void Sound() {
```

```
        Console.WriteLine("bark ");
    }
}
class Program {
    static void Main() {
        Animal myPet = new Dog();
        myPet.Jump();
        myPet.Sound();
    }
}
```

- A. "Animal jumps bark"
- B. "Dog jumps bark"
- C. Compile error
- D. "Animal jumps Dog jumps bark"
- Answer: B

7. Which statement about this property is true?

```
public int Id { get; init; }
```

- A. The property can be modified at any time by methods within the same class.
- B. After object initialization, attempting to change Id will succeed without errors.
- C. The init accessor restricts setting the property to object initialization only.
- D. This behaves identically to a property with a public set accessor.

- Answer: C
-

8. Which statement about this class is false?

```
public abstract class Processor {
    public abstract void Validate();
    public void Log(string message) {
        Console.WriteLine(message);
    }
    public virtual void Preprocess() {
        Console.WriteLine("Base preprocessing");
    }
}
```

- A. A derived concrete class must implement Validate() but can optionally override Preprocess()
 - B. The Log() method can be hidden in a derived class using the new keyword
 - C. The Preprocess() method must be overridden in any derived concrete class
 - D. A derived abstract class can defer implementing Validate() to further child classes
 - Answer: C
-

9. What are the limitation of generics in C#?

1. Enums cannot have generic type parameters.
 2. Lightweight dynamic methods cannot be generic.
 3. Using generics with value types always avoids boxing entirely.
 4. Generic types cannot use arithmetic operators directly.
 5. Each generic type instantiation gets its own independent static fields.
- A. 1,3,5

- B. 1,2,4
- C. 2,4,5
- D. 1,2,3
- Answer: B

1. Enums cannot have generic type parameters.

- Correct. Enum definitions in C# cannot include generic type parameters (e.g., `enum MyEnum<T> { ... }` is invalid).

2. Lightweight dynamic methods cannot be generic.

- Correct. Dynamic methods created via `System.Reflection.Emit.DynamicMethod` cannot be generic. This is a limitation of the reflection emit API.

3. Using generics with value types always avoids boxing entirely.

- Incorrect. While generics *usually* avoid boxing for value types, boxing can still occur in scenarios like interface constraints (e.g., `where T : IComparable`), converting to `object`, or using `Enum/Delegate` types.

4. Generic types cannot use arithmetic operators directly.

- Correct. Arithmetic operators (e.g., `+`, `-`, `*`) cannot be applied directly to generic type parameters without constraints (e.g., `T result = a + b;` fails). Workarounds require advanced constraints (e.g., `INumber<T>`) or dynamic methods.

5. Each generic type instantiation gets its own independent static fields.

- Incorrect. This is **not a limitation** but a deliberate feature. Each closed generic type (e.g., `MyClass<int>, MyClass<string>`) has separate static fields, which is intentional behavior.

- **Correct Options:**

- **1** (Enums cannot have generic type parameters)
- **2** (Lightweight dynamic methods cannot be generic)
- **4** (Generic types cannot use arithmetic operators directly)

10. Which keyword should be used when we need: "A field that cannot be modified after initialization"?

- A. Only const
 - B. Only readonly
 - C. Either const or readonly (depending on requirements)
 - D. sealed
 - Answer: C
-

MCQ

1. Select the correct statement:

1. When a MulticastDelegate contains multiple methods and one throws an exception, then invocation stops, and the exception propagates.
 2. You can inherit from MulticastDelegate to create custom delegate types.
 3. delegate.GetInvocationList() can be used to retrieve all individual delegates from a MulticastDelegate.
 4. inheritance hierarchy of MulticastDelegate is Object → Delegate → MulticastDelegate.
 5. If a MulticastDelegate has a return type, it returns an array of all method results.
- A. 1,3,4
 - B. 2,4
 - C. 1,4,5
 - D. 3,4,5
 - Answer: A
-

2. What will be output?

```
static void Main(string[] args) {  
    char A = 'K';  
    char B = Convert.ToChar(76); //76 is ascii of 'L'  
    A++;  
    B++;  
    Console.WriteLine(A+ " " +B);  
    Console.ReadLine();  
}
```

- A. M L

- B. U L
 - C. L M
 - D. A B
 - Answer: C
-

3. Which of the following statements best describes the difference between declaring a delegate field and declaring an event?

- A. Declaring an event simply generates a multicast delegate field with no additional restrictions on how client code can assign handlers.
 - B. Declaring a delegate field prevents external code from removing handlers, while declaring an event allows both adding and removing handlers.
 - C. Declaring a delegate field exposes the invocation list to external code, which can replace or clear it, whereas declaring an event restricts external code to using only += and -= operators.
 - D. Declaring an event automatically ensures that only one subscriber can be attached at a time, preventing multiple handlers from being invoked.
 - Answer: C
-

4. Which of the following statements are true for the following snippets?

```
//Snippet 1:  
ArrayList arr = new ArrayList();  
arr.Add(100);  
arr.Add("hello");
```

```
//Snippet 2:  
List<object> list = new List<object>();  
list.Add(100);  
list.Add("hello");
```

1. Both ArrayList and `List<object>` store their elements as object references, requiring boxing of value types.
 2. ArrayList provides compile-time type safety and prevents adding mixed types.
 3. Recommendation is using `List<T>` instead of ArrayList for new development.
 4. `List<object>` performs better than ArrayList for most operations due to its generic implementation.
 5. Using `List<object>` instead of ArrayList automatically eliminates the need for casting when retrieving elements.
 6. Both collections fully support LINQ extension methods.
- A. 1,2,3
 - B. 2,3,5
 - C. 3,4,6
 - D. 1,3,4
 - Answer: D
-

5. In C#, what is the accessibility level of a member declared as internal protected?
- A. Accessible only within the same assembly.
 - B. Accessible only within the same class and its derived classes.
 - C. Accessible within the same assembly OR in derived classes (even in another assembly).
 - D. Accessible only to derived classes within the same assembly.
 - Answer: C
-

6. You're building a simple order processing system where:
 - Orders must be processed in the exact sequence they arrive (**FIFO**).
 - The system should only keep the **100 most recent orders**.
 - Older orders beyond 100 should be **automatically discarded**.

- No multi-threading is required.
 - Which collection best fits these needs?
- A. `List<Order>`

```
List<Order> orders = new List<Order>(100);
orders.Add(newOrder);
if (orders.Count > 100) orders.RemoveAt(0);
```

- B. `Queue<Order>`

```
Queue<Order> orders = new Queue<Order>(100);
orders.Enqueue(newOrder);
if (orders.Count > 100) orders.Dequeue();
```

- C. `Dictionary<int, Order>`

```
Dictionary<int, Order> orders = new Dictionary<int, Order>();
orders.Add(orders.Count, newOrder);
if (orders.Count > 100) orders.Remove(orders.Count - 100);
```

- D. `LinkedList<Order>`

```
LinkedList<Order> orders = new LinkedList<Order>();
orders.AddLast(newOrder);
if (orders.Count > 100) orders.RemoveFirst();
```

- Answer: B
-

7. Which of the following statements is true for the given snippets?

```
//snippet 1:  
List<int> list1 = new List<int>(3);  
list1[0] = 9;
```

```
//snippet 2:  
List<int> list2 = new List<int>(3);  
list2.Add(1);  
list2.Add(2);  
list2.Add(3);  
list2[2] = 10;
```

- A. In Snippet 1, `list1[0] = 9;` will compile but throw `ArgumentOutOfRangeException` at runtime.
- B. In Snippet 1, `new List<int>(3);` creates a list with three elements initialized to `0`.
- C. In Snippet 2, `list2[2] = 10;` will succeed because `Count` is `3` after adding three elements.
- D. In Snippet 2, the capacity stays at `3` as long as no additional items are added.

- Answer: C
-

8. Which of the following statement is correct?

- A. In C#, writing `int numbers[];` is valid syntax for declaring an array variable.
- B. The `params` keyword can be applied to any parameter, even if it is not the last parameter in the method signature.
- C. When you declare an array with `int[] values = new int[3]{1,2,3};`, specifying the length inside the brackets is optional because the initializer provides it.

- D. The params keyword modifies the parameter name rather than the parameter type.
- Answer: C

9. What is the output of the following C# code?

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

class Program
{
    static void Main()
    {
        Dictionary<string, int> ages = new Dictionary<string, int>()
        {
            { "Alice", 25 },
            { "Bob", 30 }
        };

        ages["Charlie"] = 28;
        ages.Remove("Bob");

        if (ages.TryGetValue("Alice", out int age))
            Console.WriteLine(age);
        else
            Console.WriteLine("Not found");
    }
}
```

- A. 25
- B. ages["Charlie"] = 28; fails at runtime.

- C. Not found
- D. Throws `KeyNotFoundException`
- Answer: A

10. Which of these code snippets will throw an `InvalidOperationException` with the message "Collection was modified"?

- A.

```
var colors = new List<string> { "Red", "Green" };
foreach (var color in colors.ToArray()) {
    colors.Add(color + "!");
}
```

- B.

```
var queue = new Queue<int>(new[] { 1, 2, 3 });
foreach (var num in queue) {
    if (num > 1) queue.Dequeue();
}
```

- C.

```
var dict = new Dictionary<int, string> { { 1, "A" }, { 2, "B" } };
foreach (var pair in dict) {
    Console.WriteLine(pair.Value);
}
```

- D.

```
var numbers = new List<int> { 1, 2, 3 };
for (int i = 0; i < numbers.Count; i++) {
    if (numbers[i] == 2) numbers.RemoveAt(i);
}
```

- Answer: B
-

.NET MCQ

1. What happens when you compile this code?

```
var query = from e in employees  
            where e.Age > 30  
            select e.Name, e.Age;
```

- A. It works fine and returns Name and Age together.
- B. It only works if you write using System.Tuple;
- C. You can replace the comma with into to fix it.
- D. It gives an error because select must return a single object.
- Answer: D

2. You are working on a banking application that saves **Customer** objects to disk. The **Customer** class is given below. When serializing a **Customer** object, which field/property will **NOT** be saved to disk?

```
[Serializable]  
public class Customer  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
    [NonSerialized] private string _password;  
    public string TemporarySessionToken { get; set; }  
}
```

- A. Name
 - B. `_password` & `TemporarySessionToken`
 - C. `_password`
 - D. `TemporarySessionToken`
 - Answer: C
-

3. What happens when you compile this code?

```
public class ConfigLoader {
    var _config = LoadConfig(); // Line 1
    public void Init() {
        var config = _config; // Line 2
        var = "test"; // Line 3
        var timeout = 30; // Line 4
        Console.WriteLine(timeout.GetType().Name);
    }
    private Dictionary<string, string> LoadConfig() => new();
}
```

- A. Output: Int32
 - B. Compile error at Line 1: "var cannot be used here" and Line 3: "Invalid expression"
 - C. Output: Dictionary`2
 - D. Compile error at Line 1 and runtime error at Line 4
 - Answer: B
-

4. Which of the following statements about module initializers is FALSE?

- A. Module initializers execute before any other method in the assembly, including static constructors.
 - B. A single assembly can contain multiple module initializer methods.
 - C. Module initializers can be defined as instance methods in non-static classes.
 - D. If a module initializer throws an exception, the application fails immediately
 - Answer: C
-

5. With nullable reference types enabled What does the compiler do?

```
string nonNullableString = "Hello";
var inferredString = nonNullableString;
inferredString = null;
```

- A. No warnings – var infers non-nullable string.
 - B. Warning – var infers string? (nullable).
 - C. Error – var does not allow null assignments.
 - D. No warnings – var acts like dynamic.
 - Answer: A
-

6. In a WinForms app, the auto-generated MainForm.Designer.cs declares a partial method:

- partial void InitializeCustomComponents();
- A developer implements it in MainForm.cs:

```
partial void InitializeCustomComponents() {  
    // ...  
}
```

- Later, another developer adds a second implementation in MainForm.Experimental.cs. What happens?
 - A. Compile error – Partial methods allow only one implementation
 - B. Runtime error – Duplicate method execution
 - C. Silent merge – Both implementations run
 - D. Warning only – Code works with a compiler warning
 - Answer: A
-

7. Which of the following statements about extension methods is true?

- A. An extension method declared in a static class can override an instance method on the same type if it has the same signature.
 - B. Extension methods can be declared as static methods in any class, even if the class itself is not static, as long as the method is static and public.
 - C. If an extension method and an instance method have the same name and signature, the compiler will always prefer the extension method.
 - D. Extension methods must be static methods declared inside a static class, and the first parameter must be prefixed with "this" to specify the type they extend.
 - Answer: D
-

8. What will be the output of the following code snippet?

```
class Program {  
    static void Main(string[] args) {
```

```
int[] nums = {3, 1, 2, 5, 4};
var ltAvg = from n in nums
            let x = nums.Average()
            where n < x
            select n;
Console.WriteLine("The average is " + nums.Average());
}
}
```

- A. Run time error
 - B. 3
 - C. 1, 2
 - D. Compile time error
 - Answer: B
-

9. Consider this StreamWriter constructor - Which statement about the parameters is FALSE?

- new StreamWriter(string path, bool append)
 - A. path must be a full absolute path – relative paths always throw ArgumentException
 - B. When append is true, the writer preserves original content but does not guarantee atomic writes.
 - C. If the file at path is locked by another process, this will throw IOException regardless of append value.
 - D. Path can be a UNC network path (e.g., \\server\\file.txt).
 - Answer: A
-

10. Which of the following statements about the ArgumentException class is correct?

- A. ArgumentException is thrown when a method receives a null argument that is not allowed; therefore, it is the same as ArgumentNullException.
- B. ArgumentException is in the System.Collections namespace.
- C. ArgumentException indicates that at least one of the arguments passed to a method does not meet the parameter requirements.
- D. ArgumentException can only be thrown by the .NET runtime and cannot be thrown explicitly in your code.
- Answer: C