.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

```
┌──────────────────────────────────┐
│          Your Application        │
├──────────────────────────────────┤
│ ASP.NET / WPF / WinForms / WCF   │
├──────────────────────────────────┤
│   Base Class Library (BCL)       │
├──────────────────────────────────┤
│   Common Language Runtime (CLR)  │
└──────────────────────────────────┘
```

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.
- `peverify` → 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
' 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:

```csharp
[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
`ildasm.exe`	Disassembles IL/metadata.
`sn.exe`	Generates strong-name key pairs.
`gacutil.exe`	Manages the GAC.
`peverify`	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
mscoree.dll	CLR shim that selects runtime version.
mscorlib.dll	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	`newobj`, `callvirt`	Instantiation/method calls.
Metadata	`ldtoken`, `castclass`	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`).	`private int _count;`
Properties	Controlled access to fields (get/set).	`public string Name { get; set; }`
Methods	Define behavior.	`public void Save() { ... }`
Constructor	Initialize new objects.	`public Car() { ... }`

---

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#)
