

UNIT 1

Introducing C# and the .NET Framework

(7 Hours) — DotNet Technology (CACS302)

This unit introduces the C# programming language and the .NET Framework that hosts it. We look at the core ideas that shape modern managed application development — object orientation, type safety, and automatic memory management — and then place C# within the wider .NET ecosystem: the Common Language Runtime (CLR), the .NET Framework itself, competing and complementary frameworks, and the standardisation effort represented by .NET Standard 2.0.

1. Object Orientation

C# was designed from the ground up as an object-oriented language. Every piece of executable code in C# lives inside a class (or a struct), and the language provides direct, first-class support for the four pillars of object orientation.

- Encapsulation – data and the methods that operate on it are bundled together inside a class, and access modifiers (private, protected, public, internal) control what is visible from outside.
- Inheritance – a class can derive from a single base class, inheriting its members and extending or specialising them.
- Polymorphism – virtual methods, method overriding, and interfaces let the same call site behave differently depending on the runtime type of the object.
- Abstraction – abstract classes and interfaces define contracts that hide implementation detail behind a well-defined member list.

Example 1.1 — A Simple Class Hierarchy

```
public abstract class Shape
{
    public abstract double Area();
}

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

Note: Unlike C++, C# supports only single inheritance for classes, but a class may implement any number of interfaces, which gives most of the flexibility of multiple inheritance without its ambiguity problems.

2. Type Safety

C# is a strongly, statically typed language. Every variable, parameter, and return value has a type that is known at compile time, and the compiler rejects operations that are not valid for that type. This is what is meant by type safety: the runtime guarantees that code can only access memory through the type it was declared as, preventing an entire class of bugs and security vulnerabilities that plague unsafe languages such as C.

- The CLR performs type-checking at load time (verification) and again at execution time for operations such as casts.
- Implicit conversions are only allowed when they cannot lose information or throw (e.g. int to long); anything else needs an explicit cast.
- Generics (*List<T>*, *Dictionary<TKey,TValue>*) give type-safe collections without boxing or unsafe casts.
- An *unsafe* keyword exists for pointer arithmetic, but code that uses it must be explicitly marked, making unsafe operations visible and auditable.

Aspect	Type-Safe Language (C#)	Type-Unsafe Language (C, raw pointers)
Memory access	Only through declared type	Any pointer can be reinterpreted
Array bounds	Checked at runtime	Not checked – can overflow
Casting	Verified; invalid casts throw	Reinterpret_cast style, no checks
Common bugs prevented	Buffer overruns, type confusion	N/A – programmer's responsibility

3. Memory Management

One of the biggest productivity gains .NET offers over languages like C++ is automatic memory management through garbage collection (GC). Developers allocate objects with *new* but never explicitly free them; the CLR's garbage collector reclaims memory once it determines an object is no longer reachable.

3.1 The Managed Heap and Generations

Objects are allocated on the managed heap. To keep collection fast, the GC divides the heap into generations:

Generation	Contents	Collection Frequency
Gen 0	Newly allocated, short-lived objects	Very frequent
Gen 1	Objects that survived a Gen 0 collection	Occasional – acts as a buffer
Gen 2	Long-lived objects (e.g. static data, caches)	Rare, more expensive
LOH	Large objects ($\geq 85,000$ bytes)	Collected with Gen 2

- The GC is generational and tracing: it starts from known roots (static fields, local variables, CPU registers) and marks everything reachable; unreachable objects are garbage.
- Compaction moves surviving objects together to avoid heap fragmentation and keep allocation fast (a simple pointer bump).
- The *IDisposable* interface and the *using* statement let developers release unmanaged resources (file handles, sockets, database connections) deterministically, since the GC only manages memory, not external resources.

Example 1.2 — Deterministic Cleanup with using

using (var reader = new StreamReader("data.txt"))

```
{
    string line = reader.ReadLine();
}
// reader.Dispose() is called automatically here,
// even if an exception is thrown inside the block.
```

Note: Garbage collection removes an entire category of bugs (dangling pointers, double frees, memory leaks from forgotten deallocation) but shifts responsibility to the developer to manage non-memory resources explicitly via Dispose.

4. Platform Support

Historically, the .NET Framework was a Windows-only technology, tightly bound to the Win32 API and the Windows operating system. Over time, Microsoft's platform story broadened considerably.

Implementation	Platforms	Notes
• .NET Framework	Windows only	The original, mature runtime; still used for legacy Windows apps
• .NET Core / .NET (5+)	Windows, Linux, macOS	Cross-platform, open-source, high performance

Implementation	Platforms	Notes
• Xamarin / .NET MAUI	iOS, Android, Windows, macOS	Mobile and cross-platform desktop UI
• .NET Compact/Micro Fw.	Embedded and mobile devices	Legacy, superseded by .NET IoT / nanoFramework

This shift towards cross-platform support is one of the main reasons the .NET Standard specification exists (covered in Section 9): it defines a common surface area that every one of these implementations can promise to support, letting a single class library run unmodified across all of them.

5. C# and the CLR

C# does not run directly on hardware. The compiler translates C# source into Common Intermediate Language (CIL), a platform-neutral, low-level instruction set packaged inside an assembly (a .dll or .exe file with a manifest describing its contents).

- At run time, the Common Language Runtime (CLR) loads the assembly and its Just-In-Time (JIT) compiler translates CIL into native machine code for the current CPU, method by method, as each method is first called.
- The CLR also supplies the services that make managed code possible: garbage collection, type verification, exception handling, thread management, and security enforcement.
- Because CIL is language-neutral, any Common Language Infrastructure (CLI) compliant language — C#, VB.NET, F#, managed C++ — compiles down to the same CIL and can interoperate seamlessly on the CLR: a C# class can inherit from a VB.NET base class without either language knowing about the other.

Example 1.3 — From Source Code to Execution

1. C# source (.cs) is compiled by csc/Roslyn → CIL + metadata (assembly)
2. CLR loads the assembly and verifies the CIL is type-safe
3. JIT compiler compiles each method to native code on first call
4. Native code executes directly on the CPU, managed by the CLR

Note: CIL is disassembled by tools such as ildasm or dotPeek, which is why decompiling .NET assemblies back to readable C#-like code is comparatively easy — the CIL retains rich type metadata.

6. CLR and the .NET Framework

The CLR is only one part — the execution engine — of the larger .NET Framework. The Framework as a whole is best understood as a layered stack.

Layer	Role
Languages (C#, VB.NET, F#, ...)	Source code written by developers
Base Class Library (BCL) / FCL	Collections, I/O, strings, networking, threading, etc.
Common Language Runtime (CLR)	JIT compilation, garbage collection, type safety, security
Operating System	Windows (traditional .NET Framework) or cross-platform host

On top of the BCL sit higher-level application frameworks such as ASP.NET (web applications and services), Windows Presentation Foundation – WPF and Windows Forms (desktop UI), ADO.NET and Entity Framework (data access), and Windows Communication Foundation – WCF (distributed services).

Note: A common exam distinction: the CLR executes code; the .NET Framework is the entire platform — CLR plus class libraries plus application frameworks — that developers actually build on.

7. Other Frameworks

Beyond Microsoft's own .NET Framework, several related and competing frameworks are worth knowing about, either because they share heritage with .NET or because they represent alternative approaches to the same managed-runtime idea.

- Mono – an open-source, cross-platform implementation of the CLI, originally created to bring .NET to Linux; it later became the foundation for Xamarin's mobile runtimes.
- .NET Core (now unified into .NET 5+) – Microsoft's own open-source, cross-platform, high-performance reimplement of the runtime and BCL, designed to run on Windows, Linux, and macOS from a single codebase.
- Java / JVM – conceptually the closest analogue: Java bytecode and the JVM parallel CIL and the CLR respectively, including JIT compilation and garbage collection, but the two platforms are not binary-compatible.
- Node.js and other language runtimes – not CLI-based at all, but frequently compared to .NET when choosing a server-side platform for new projects.

8. Framework Overview

Pulling the previous sections together, the .NET Framework can be summarised by its major version families and the application types each one enabled.

Version Family	Key Additions
.NET Framework 1.0 – 1.1	Initial CLR, BCL, ASP.NET, Windows Forms
.NET Framework 2.0	Generics, nullable types, 64-bit support
.NET Framework 3.0 – 3.5	WPF, WCF, WF, LINQ, Entity Framework
.NET Framework 4.0 – 4.8	Dynamic language runtime, async/await, TPL
.NET Core 1.0 – 3.1	Cross-platform rewrite, side-by-side deployment
.NET 5 and later	Unified single platform for all app types

Architecturally, every version keeps the same essential shape described in Section 6: a language compiles to CIL, the CLR hosts and executes that CIL, and a class library provides the functionality applications actually call.

9. .NET Standard 2.0

As the .NET ecosystem fragmented across .NET Framework, .NET Core, Xamarin, and Mono — each with a slightly different API surface — library authors faced a problem: which APIs could they safely use if they wanted their library to run everywhere? .NET Standard was created to answer that question.

- .NET Standard is a formal specification of a common set of APIs that every conforming .NET implementation must provide — it is not itself a runtime or a product you install.
- Each implementation (.NET Framework, .NET Core, Xamarin, Mono, UWP) maps to a particular .NET Standard version it supports; a higher .NET Standard version requires more APIs but runs on fewer, newer implementations.
- .NET Standard 2.0 was a major milestone: it roughly doubled the API surface compared with 1.x, adding back many commonly used BCL types, which made porting existing .NET Framework libraries to .NET Core substantially easier.
- .NET Standard 2.0 is supported by .NET Framework 4.6.1+, .NET Core 2.0+, Mono 5.4+, and Xamarin.iOS/Android, giving library authors a single target that covers almost the entire ecosystem at the time.

Concept	Description
.NET Standard	A specification (a contract) of available APIs
.NET Framework / Core / Xamarin	Concrete implementations that fulfil the contract
Class library targeting .NET Standard 2.0	Can be referenced from any implementation that supports it

Note: From .NET 5 onward, Microsoft recommends targeting .NET (5, 6, 8, ...) directly rather than .NET Standard for new code, since the unified platform replaced the need for the standard in most new

projects. .NET Standard remains important for libraries that must still support older .NET Framework consumers.

10. Applied Technologies

The concepts covered in this unit underpin a wide range of real-world technologies built on top of C# and the .NET platform:

- ASP.NET Core – building web applications, REST APIs, and real-time services (SignalR).
- Windows Presentation Foundation (WPF) and .NET MAUI – rich desktop and cross-platform client applications.
- Entity Framework Core – object-relational mapping for data access against relational databases.
- Azure Functions / ASP.NET microservices – cloud-native, serverless, and containerised deployment of .NET applications.
- Unity game engine – uses C# (via Mono/IL2CPP) as its primary scripting language for game logic.

Worked Example 1.4 — Tracing a Program's Journey

Problem: A developer writes a C# console application and runs it on both Windows and Linux. Explain what happens at each stage.

Solution:

1. The C# compiler (Roslyn) compiles the source into an assembly containing CIL and metadata – this step is identical on both OSes.
2. On Windows, the .NET runtime's CLR loads the assembly; on Linux, the .NET (Core) runtime's CLR does the same – both are CLI-compliant execution engines, just built for different operating systems.
3. Each runtime's JIT compiler converts CIL to native code for the host CPU architecture on first use.
4. The garbage collector manages the managed heap identically in principle on both platforms, since GC behaviour is part of the CLR specification, not the OS.

This illustrates exactly why type safety, JIT compilation, and a common CIL format matter: the same source code and the same compiled assembly can run unmodified on multiple operating systems.

Key Takeaways

- ✓ C# is a strongly-typed, object-oriented language supporting encapsulation, inheritance, polymorphism, and abstraction.
- ✓ Type safety means the CLR restricts access to memory through declared types, preventing an entire class of low-level bugs.
- ✓ The CLR's garbage collector automatically manages the managed heap using a generational (Gen 0/1/2 + LOH) scheme; unmanaged resources still need explicit Dispose/using.
- ✓ The .NET Framework began as Windows-only; modern .NET runs across Windows, Linux, macOS, and mobile platforms.
- ✓ C# source compiles to CIL, which the CLR loads, verifies, and JIT-compiles to native code at run time.
- ✓ The .NET Framework is a layered stack: languages, BCL/FCL, CLR, and OS.
- ✓ Mono, .NET Core, and Java/JVM are related or comparable managed-runtime frameworks.
- ✓ .NET Standard 2.0 is a specification (not a runtime) that unifies the API surface across .NET Framework, .NET Core, Xamarin, and Mono.
- ✓ C# and .NET underpin applied technologies ranging from ASP.NET Core and WPF to Entity Framework and the Unity game engine.

— *End of Unit 1* —