

1: What's the Difference Between Compiled and Interpreted Languages?

❖ **Compiled Languages**

- The source code is **translated entirely** into **machine code** by a **compiler** before the program runs.
- This machine code is what the computer actually executes.
- Once compiled, you don't need the source code or compiler to run the program.

➤ **Pros:**

- Fast execution (already translated into machine language).
- Better performance and optimization.

➤ **Cons:**

- Slower to develop and debug (because you must compile after every change).
- Platform-dependent unless designed for cross-platform use.

➤ **Examples:** C, C++, Go, Rust

❖ **Interpreted Languages**

- The source code is **read and executed line by line** by an **interpreter** at runtime.
- No separate machine code file is generated in advance.

➤ **Pros:**

- Easy to test and debug (no need to compile).
- Cross-platform (as long as interpreter exists for the system).

➤ **Cons:**

- Slower execution (interpreting every time).
- Depends on the interpreter being present.

➤ **Examples:** Python, JavaScript, Ruby

❖ So... What About C# ?

C# is **neither purely compiled nor purely interpreted** — it's **in between**, using a **hybrid model**:

- **C# is a compiled language, but not directly into machine code.**

Here's what happens:

1. You write C# code.
2. The **C# compiler (csc)** compiles it into **Intermediate Language (IL)** code.
3. That IL is stored in an assembly (like a .exe or .dll).
4. When you run the program, the **.NET Runtime (CLR)** uses a **JIT (Just-In-Time) compiler** to translate the IL into **machine code** at runtime.

So:

- **Faster than interpreted** languages.
- **Cross-platform** with .NET Core/.NET 5+.
- Still needs .NET Runtime to execute.



Summary Table:

Feature	Compiled	Interpreted	C#
Translation time	Before running	During running	Before (to IL), then JIT
Execution speed	Fast	Slower	Medium to Fast
Needs runtime env.?	No	Yes	Yes (CLR/.NET Runtime)
Platform dependency	Often yes	No	No (with .NET Core/.NET)

2: Comparison: Implicit vs Explicit vs Convert vs Parse in C#

Feature	Implicit Casting	Explicit Casting	Convert Class	Parse Method
Definition	Automatic type conversion	Manual type conversion using casting operator	Uses .NET Convert class to change one type to another	Converts strings to numeric types (or DateTime, etc.)
Syntax	double x = intVal;	int x = (int)doubleVal;	int x = Convert.ToInt32(strVal);	int x = int.Parse(strVal);
Safe?	Yes (no data loss)	Risky (possible data loss or exceptions)	Handles nulls, safer than Parse	Risky if input is null or invalid
From → To	Smaller type → larger type	Larger type → smaller type	Any supported type → Another supported type	String → Numeric/Date/Bool
Throws exception?	No	Yes (if invalid cast)	Yes (if conversion fails)	Yes (FormatException or NullReferenceException)
Use case	When conversion is always safe	When you know the value can be cast safely	When converting across types (even null strings)	When converting a valid string to a value type
Namespace	None (built-in language feature)	None (built-in language feature)	System.Convert	Each value type (e.g., int.Parse, double.Parse)

➤ **Summary:**

- **Use Implicit** when the conversion is guaranteed to be safe.
 - **Use Explicit** when you need to force a conversion (be careful of data loss).
 - **Use Convert** when working with objects or strings that may be null or need more flexible conversion.
 - **Use Parse** when converting a well-formatted string (fast but strict).
-

Bonus

3: What Does It Mean That C# Is Managed Code?

When we say that **C# is managed code**, we mean that the **execution and memory management** of C# programs is handled by the **.NET runtime environment** — specifically, the **CLR (Common Language Runtime)**.

❖ **Definition:**

Managed code is any code that runs **under the control of the CLR**, which provides automatic services like:

- Memory management
- Garbage collection
- Type safety
- Exception handling
- Security
- Cross-language integration






C# code is **compiled into IL (Intermediate Language)**, and when it runs, the CLR manages how that IL is translated into machine code and executed.

❖ **In contrast: Unmanaged code**

Unmanaged code runs **directly on the OS**, without CLR. It means:

- You are responsible for memory allocation and release (e.g., using malloc and free in C/C++).
- No garbage collection or type safety.
- Examples: C, C++, Assembly.

❖ Benefits of Managed Code in C#:

Feature	Description
 Garbage Collection	Automatically frees unused memory
 Type Safety	Ensures variables hold only valid data
 Exception Handling	Unified and structured error management
 Code Access Security	Restricts code from performing harmful actions
 Cross-language Support	Supports interop with other .NET languages like VB.NET, F#

❖ Summary:

Saying **C# is managed code** means it **runs inside the .NET CLR**, and **the runtime takes care of low-level tasks** like memory management, type checking, and security — making development easier and safer.

4: What Does It Mean That a struct Is Like a class (before)?

In C#, both struct and class are **user-defined types**, meaning you can define your own custom data structures using either. On the surface, they seem similar because:

- Both can have **fields, properties, methods**, and **constructors**.
- Both can implement **interfaces**.
- Both are declared using similar syntax.

➤ So in that sense:

A struct **looks like a class** at first glance or “**before**” you go deeper into their behavior and usage differences.

But there are **important differences** between them:

❖ **struct vs class in C#**

Feature	struct	class
Type Category	Value type	Reference type
Stored in	Stack (usually)	Heap (managed by CLR)
Memory Allocation	Inline, more efficient for small data	Allocated in heap, garbage-collected
Inheritance	Cannot inherit from another struct/class	Can inherit from another class
Default Constructor	Not allowed (unless using new() with fields manually set)	Allowed and customizable
Nullability	Cannot be null (unless Nullable<T>)	Can be null
Copy Behavior	Copied by value (creates new copy)	Copied by reference (same instance)

❖ **Summary:**

Saying "struct is like class before" means:

Structs look like classes in terms of syntax and structure, but behave differently behind the scenes due to being value types — making them suitable for lightweight, short-lived data.