C# : Intro

C# Programming – Lecture - 1

C# is one of the most powerful, modern, and versatile programming languages in the software industry today. Whether you are building enterprise-grade applications, desktop solutions, web APIs, cloud-native microservices, IoT applications, or mobile apps using MAUI/Xamarin, C# and the .NET ecosystem provide a complete end-to-end development environment. This lecture introduces the foundational concepts required for any beginner to comfortably start writing programs in C# while understanding how the .NET ecosystem functions behind the scenes.

1. Course Fundamentals & .NET Overview

What is .NET?

.NET is a **free, open-source, cross-platform** developer platform created by Microsoft. It allows developers to build applications that run on Windows, Linux, and macOS. .NET supports multiple programming languages, but **C# is the primary and most widely used language**.

The .NET platform includes:

- **.NET Runtime (Core CLR / CLR)**
- **Base Class Library (BCL)**
- **Language Compilers**
- **Development Tools like Visual Studio**

Key Reasons to Use .NET

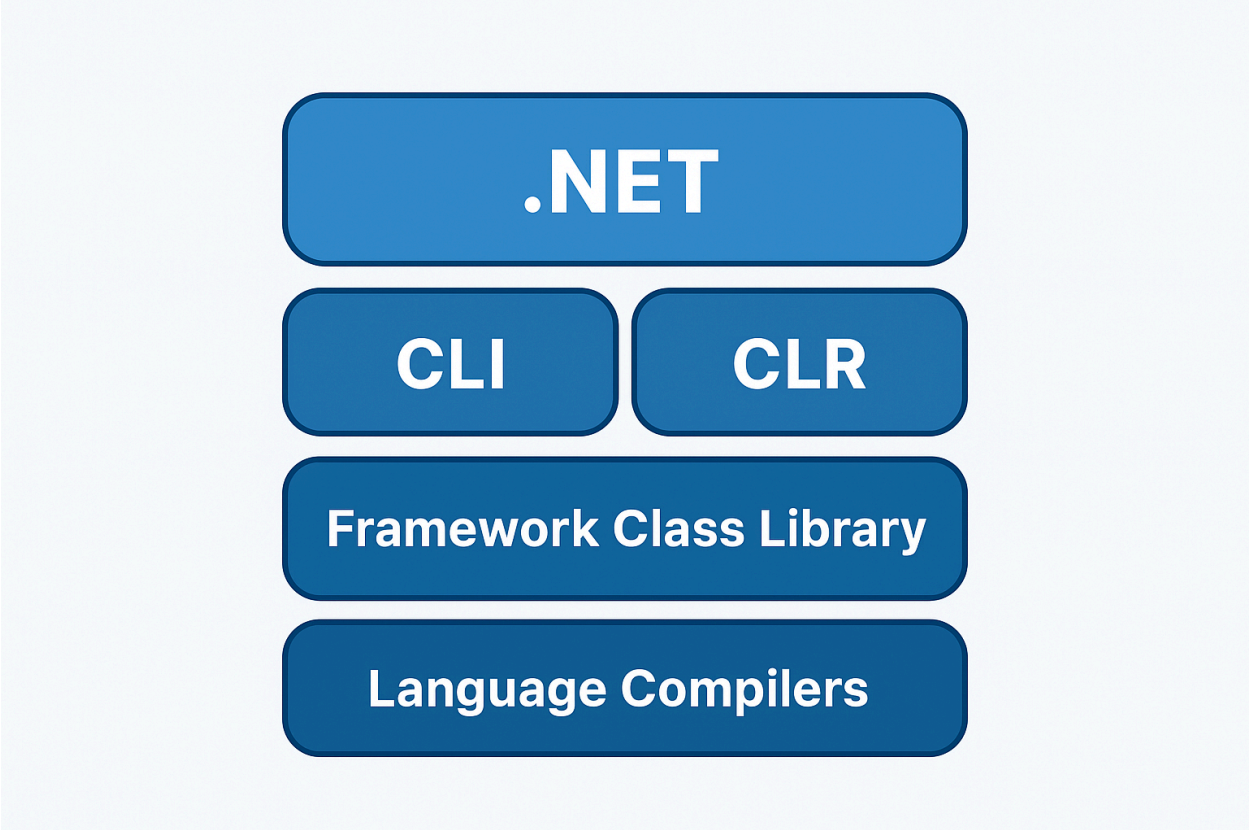
- Cross-platform execution
- Unified development model
- Vast standard library

- High performance and security
- Easy deployment
- Massive community support

.NET is a free, open-source, cross-platform development framework by Microsoft used to build:

- Desktop apps
- Web APIs
- Cloud apps
- Mobile apps (Xamarin / MAUI)
- IoT apps

It provides a unified platform for developers using C#, F#, or VB.NET.



Key Components of .NET

Component	Description
CLI (Common Language Infrastructure)	Standard by ECMA/ISO. Defines how languages run on .NET.
CLR (Common Language Runtime)	Execution engine of .NET—handles memory, garbage collection, security, and thread management.
BCL (Base Class Library)	Predefined classes like `System`, `Collections`, `IO`, etc.
Framework Architecture	Layered model: applications → runtime → libraries → OS.

CLI vs CLR (Simple Explanation)

1. CLI (Common Language Infrastructure)

What it is

CLI is a standard/specification defined by Microsoft.

Think of it as:

CLI defines:

- **How code should be compiled**
- **How languages interoperate**
- **How execution should happen**
- **Structure of Intermediate Language (IL)**
- **Common Type System (CTS)**

- **Common Language Specification (CLS)**

CLI itself does not run code.

Example

CLI says:

- **All .NET languages should compile to IL**
- **All languages should follow common data types like int, string**

Use case

- **Enables language independence**
- **Allows C#, VB.NET, and F# to work together**
- **Makes .NET cross-platform possible**

---

2. CLR (Common Language Runtime)

What it is

CLR is the actual runtime engine that executes .NET programs.

Think of it as:

CLR implements the CLI rules.

CLR does:

- **Executes IL code**
- **Converts IL to machine code (JIT compilation)**
- **Memory management (Garbage Collection)**
- **Exception handling**
- **Security and type safety**

Example

You write a C# program → compiled to IL → CLR runs it

Console.WriteLine("Hello");

CLR executes this instruction.

Use case

- **Running desktop apps**
- **Running web applications (ASP.NET)**
- **Running enterprise systems**

Key Differences (Exam Table)

Feature	CLI	CLR

Type	Specification / Standard	Runtime Environment
Role	Defines rules	Executes code
Runs code	No	Yes
Example	Blueprint	Engine
Dependency	CLR follows CLI	CLR is based on CLI

One-Line Exam Answer

- **CLI is a standard that defines how .NET languages and runtime should work.**
- **CLR is the runtime that executes .NET applications by following CLI rules.**

Easy Memory Trick

- **CLI = Rules**
- **CLR = Runner**

How .NET Works Internally

1. **C# Source Code (.cs)**
2. Compiled into **IL (Intermediate Language)** by the C# compiler
3. Stored inside **.dll or .exe**
4. At runtime, CLR converts IL → **Machine Code (JIT compilation)**
5. Program executes

This ensures platform independence and high performance.

2. Understanding CLI, CLR & .NET Framework Architecture

To understand how a C# program executes, we need to understand several core components:

Common Language Infrastructure (CLI)

A standardized execution model that ensures languages like C#, F#, and VB.NET run in a unified manner.

Common Language Runtime (CLR)

The CLR is the heart of .NET. It acts as a virtual machine that:

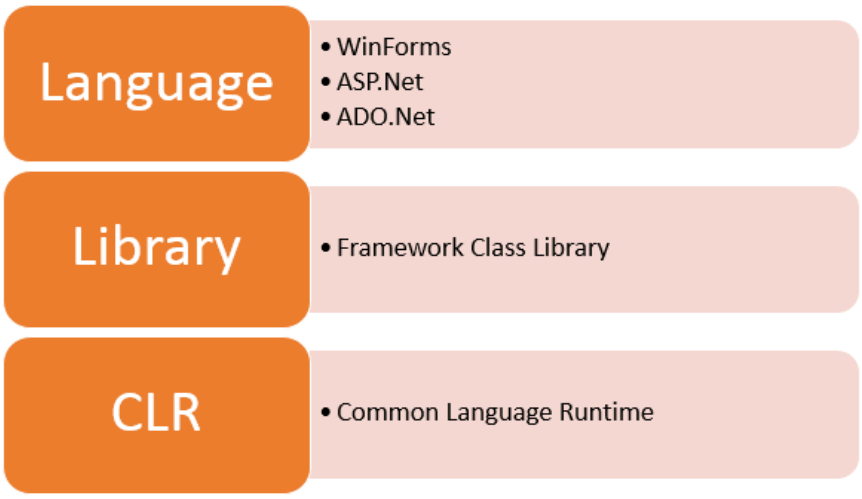
- Provides **memory management**
- Runs **garbage collection**
- Compiles IL to machine code (JIT)

- Ensures **type safety**
- Manages **exceptions**, **security**, and **threading**

Execution Process

1. You write C# code (`.cs file`).
2. Compiler converts it into **Intermediate Language (IL)**.
3. IL is stored inside **DLL/EXE assemblies**.
4. CLR's **JIT compiler** converts IL into machine code.
5. Code is executed by your CPU.

This architecture allows .NET to be powerful, secure, consistent, and portable.



.Net Framework Architecture Diagram

---

2. Introducing Visual Studio

What is Visual Studio?

Visual Studio is Microsoft's **full-featured IDE** for building C# applications. It includes:

- Intelligent code suggestions (IntelliSense)
- Powerful debugging tools
- Integrated terminal
- NuGet package manager
- Project templates

Creating Your First C# Console Project

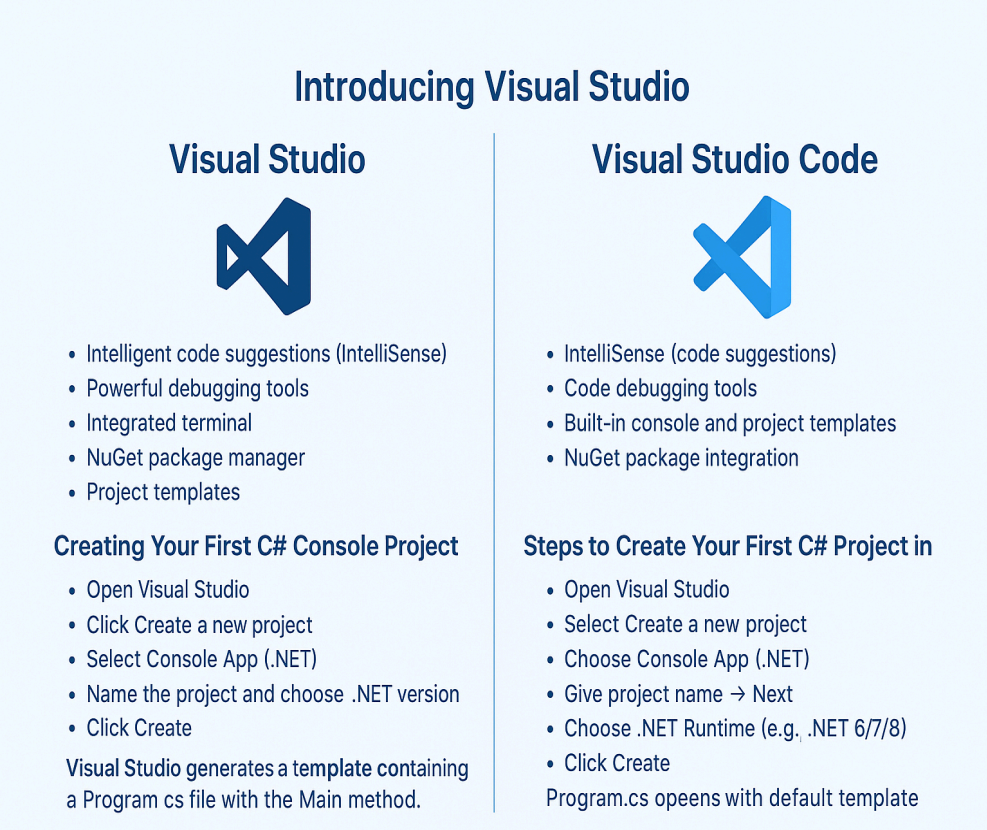
1. Open Visual Studio
2. Click **Create a new project**
3. Select **Console App (.NET)**
4. Name the project and choose .NET version
5. Click **Create**

Visual Studio generates a template containing a Program.cs file with the Main method.

Visual Studio Code (VS Code)

VS Code is a lightweight editor used widely by developers for quick programming tasks.
You must install:

- C# Dev Kit Extension
- .NET SDK



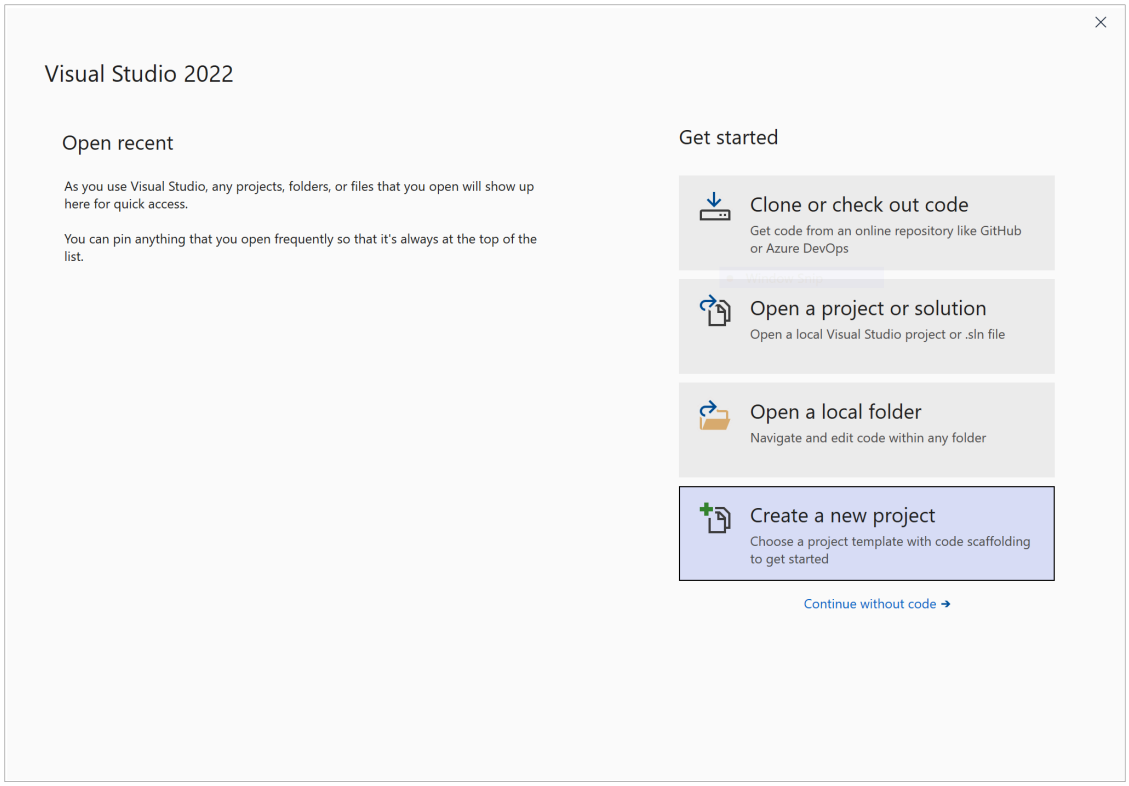
VS Code is ideal when you want a fast environment for algorithmic programming, competitive coding, or script-based tasks.

Microsoft's IDE used for developing C#/.NET applications. Features:

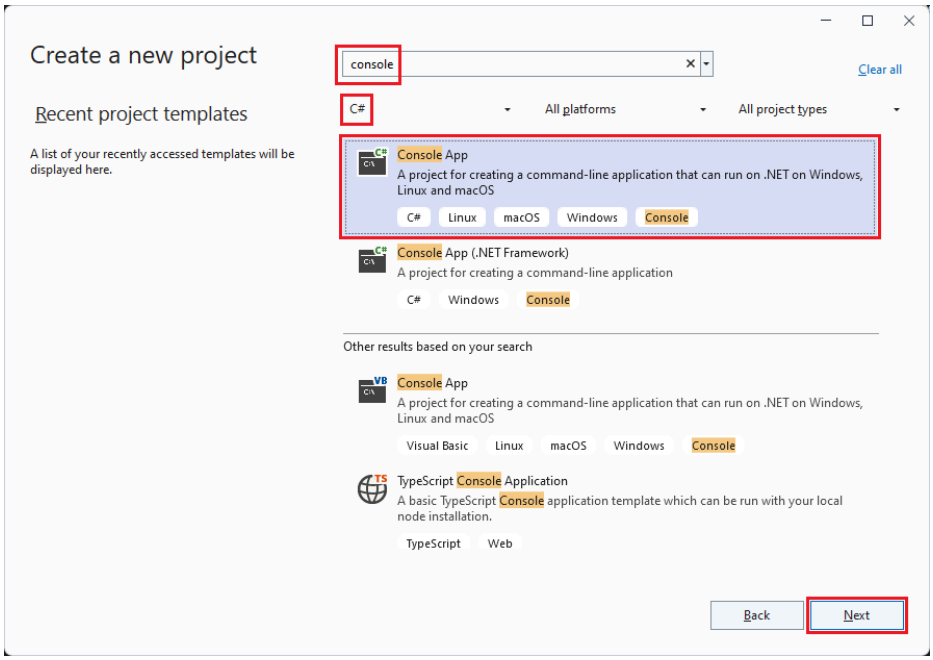
- IntelliSense (code suggestions)
- Code debugging tools
- Built-in console and project templates
- NuGet package integration

Steps to Create Your First C# Project in Visual Studio

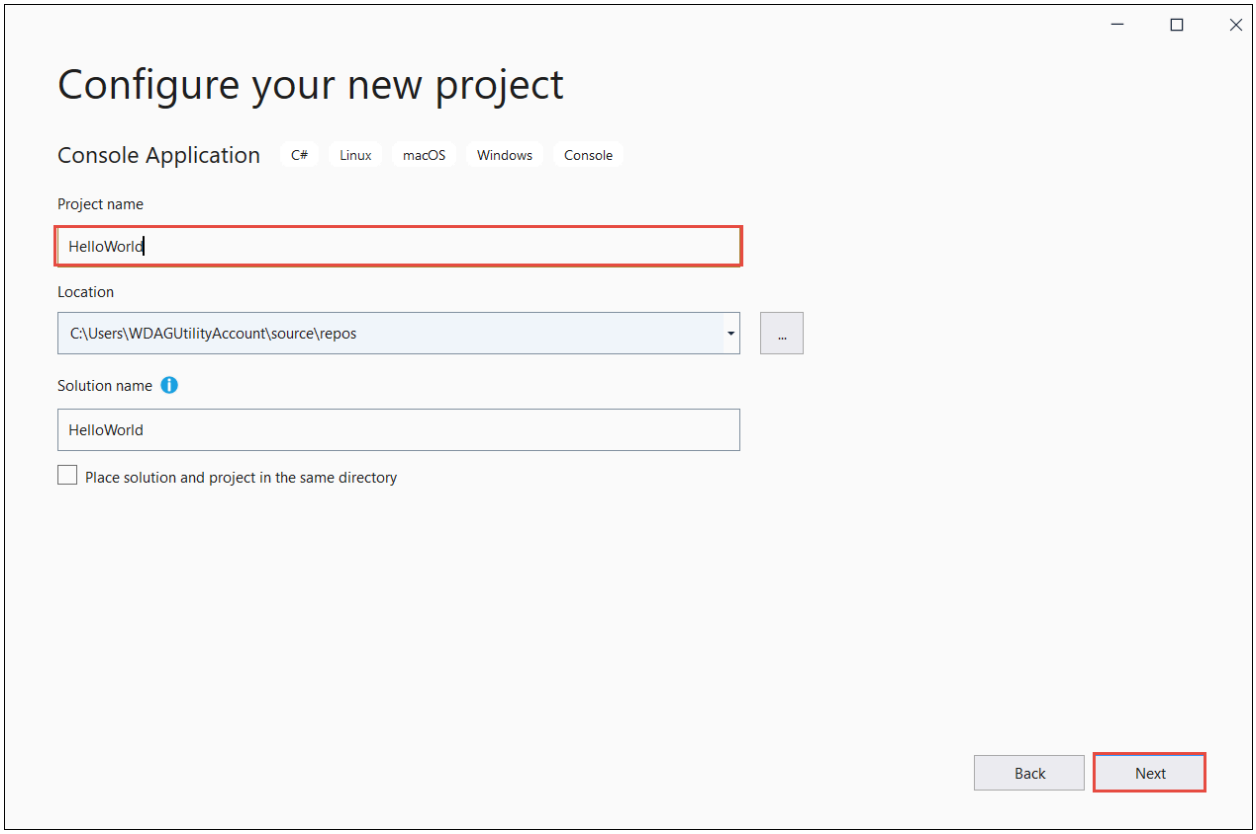
1. Open **Visual Studio**
2. Select **Create a new project**



3. Choose **Console App (.NET)**



4. Give project name → *Next*



Additional Information Dialog — Configuration Steps

In the **Additional information** dialog:

- **Select .NET 8.**
- **Select *Do not use top-level statements*.**
- **Select Create.**

The template creates a simple application that displays **"Hello, World!"** in the console window. The code is generated inside the **Program.cs** (for C#) or **Program.vb** (for Visual Basic) file.

5. Choose .NET Runtime (e.g., .NET 6/7/8)
6. Click **Create**
7. `Program.cs` opens with default template

Visual Studio Code (VS Code)

VS Code is lightweight and uses extensions:

- C# Dev Kit Extension
- .NET SDK
- Debugger for C#

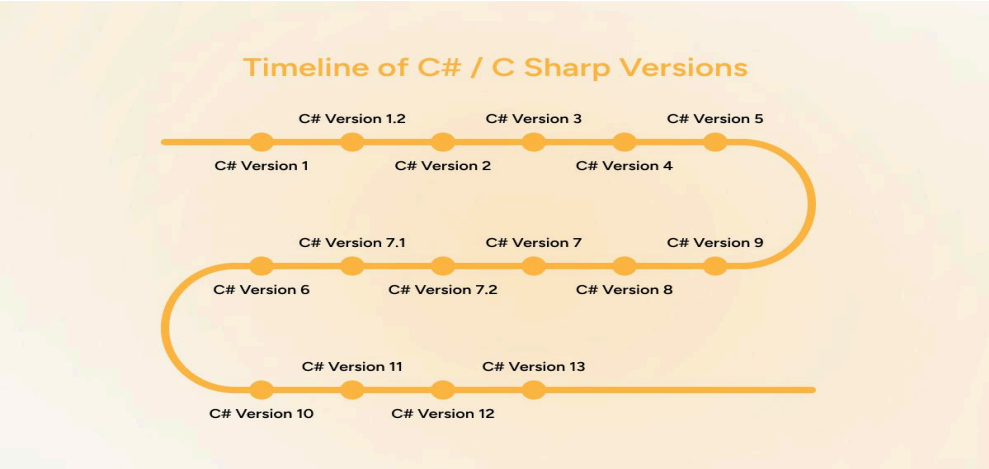
Used for logical programming tasks, algorithmic programs.

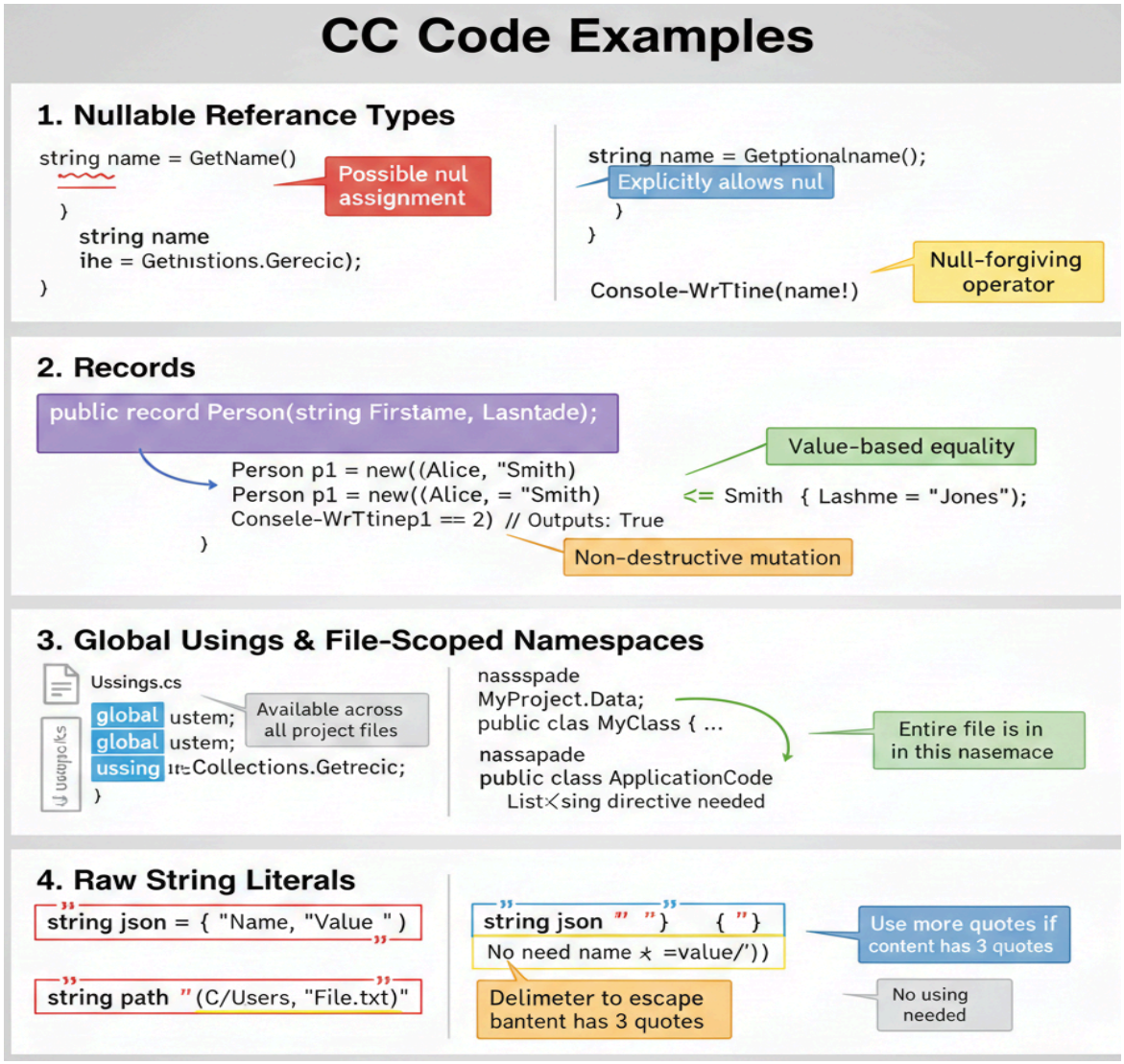
---

3. Latest Versions of C# (Above 8.0 Features)

Important Features > C# 8.0

- **C# 8.0**: Nullable reference types, async streams, switch expressions
- **C# 9.0**: Records, init-only setters
- **C# 10.0**: Global usings, file-scoped namespaces
- **C# 11.0**: Required members, raw string literals
- **C# 12.0**: Primary constructors, alias any type
- **C# 13.0+**: Enhancements in pattern matching & collection expressions





---

4. C# Language Introduction

Understanding C#

C# is a modern, object-oriented, strongly-typed language.

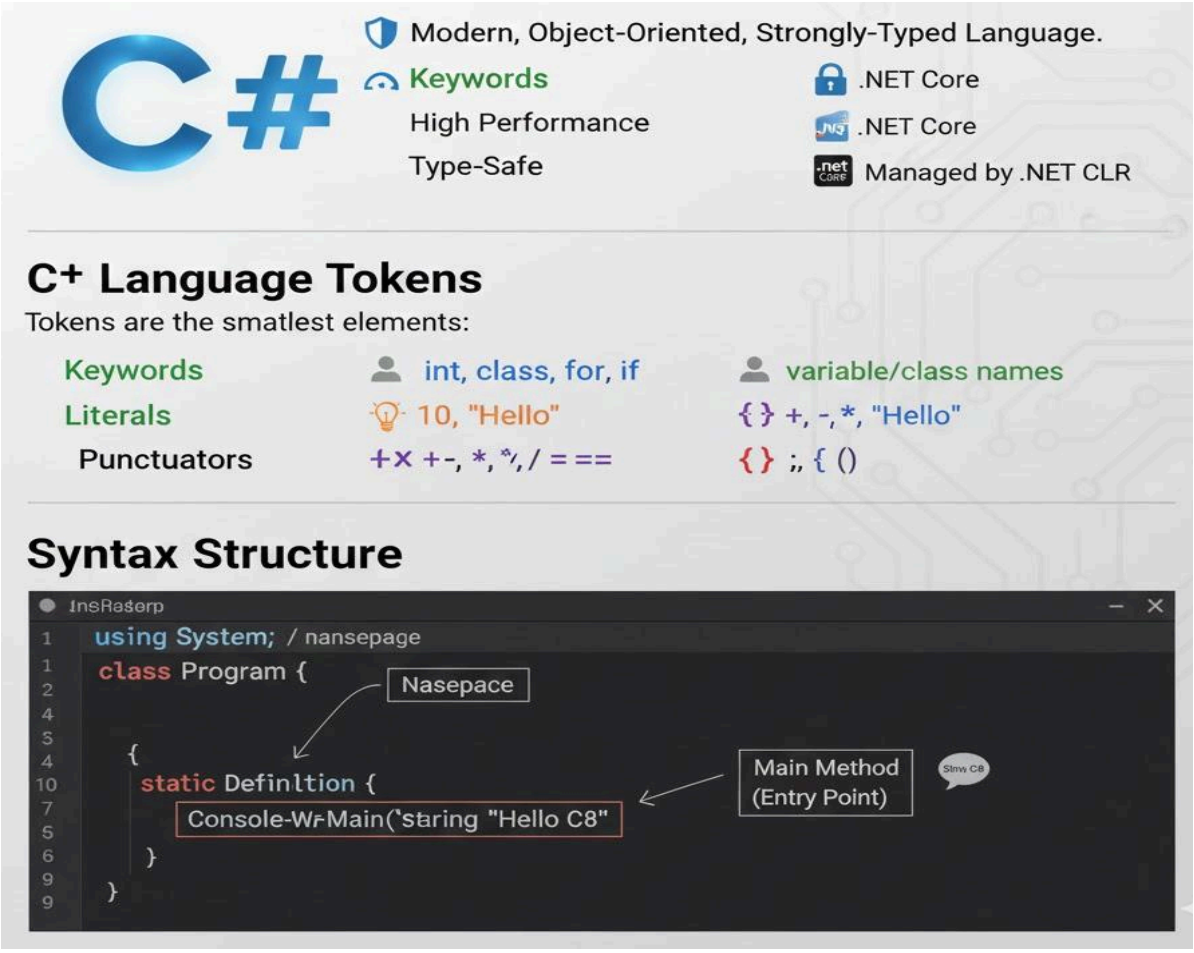
- High performance
- Type-safe
- Managed by .NET CLR

C# Language Tokens

Tokens are the smallest elements:

- **Keywords** `int`, `class`, `for`, `if`
- **Identifiers** (variable/class names)
- **Literals** `10`, `"Hello"`
- **Operators** `+`, `-`, `*`, `/`, `==`
- **Punctuators** `;`, `{}`, `()`

Syntax Structure

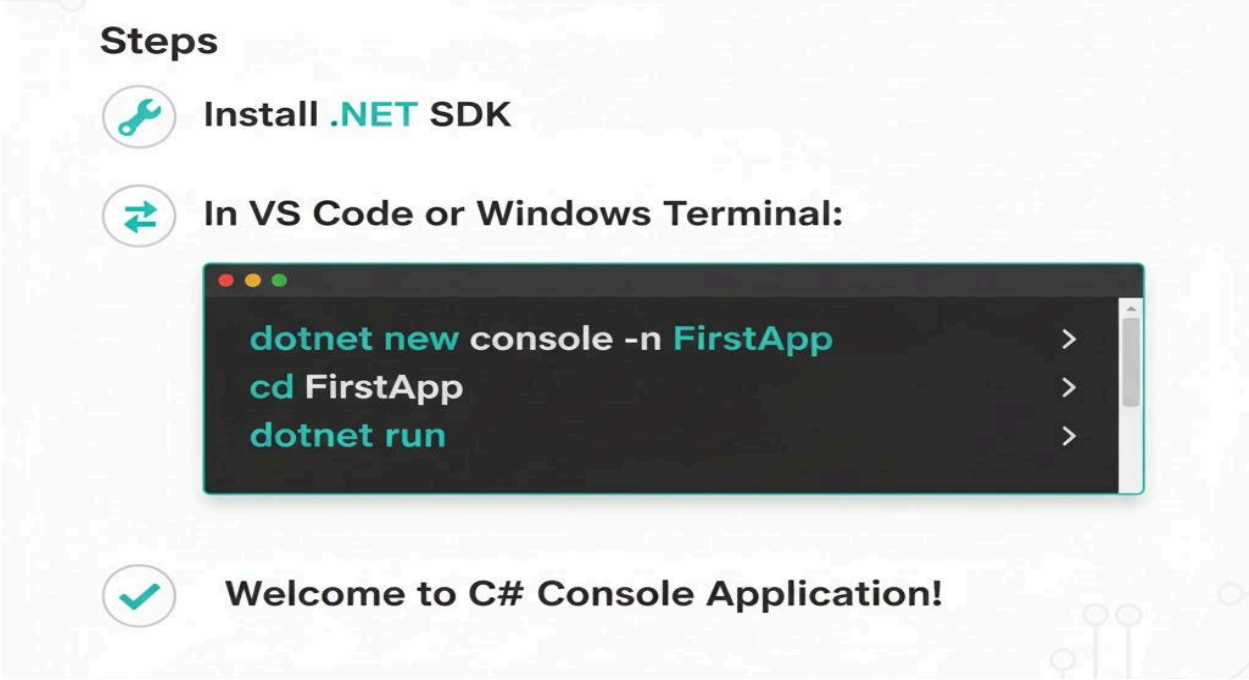


---

5. First C# Application (Console App)

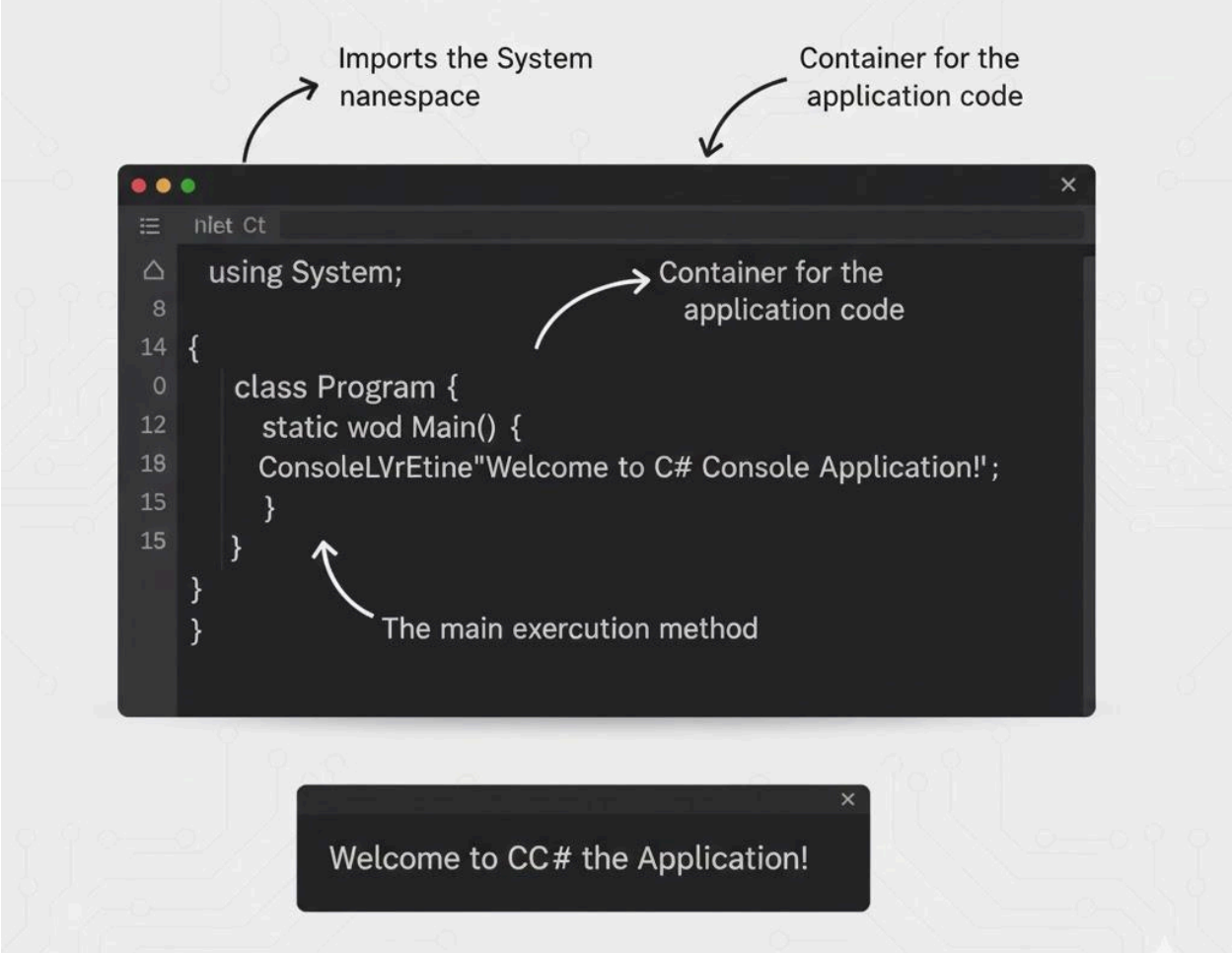
Steps

1. Install **.NET SDK**
2. In VS Code or Windows Terminal:



Example Code

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Welcome to C# Console Application!");
    }
}
```



---

6. Basic Programming Concepts

Variables

Variables store data in memory.

Variables are fundamental building blocks in any programming language. In C#, a variable is essentially a **named storage location** in the computer's memory that holds a value.

Because C# is a **strongly-typed** language, every variable must be declared with a specific data type (like int, double, or string) before it can be used. This type determines the kind of data the variable can hold and how much memory it will occupy.

The structure of a variable declaration is:

Data_Type Variable_Name = Value

Primitive Types

Type	Example	Storage Size (Typical)	Description	Key Detail
int	10	4 bytes	Used for whole numbers (integers) without any decimal component. This is the most common choice for counting.	Ranges from roughly ± 2.1 billion. When you need a larger integer, use long (8 bytes).

float	3.14f	4 bytes	Represents single-precision floating-point numbers. This type is generally faster but offers less precision.	Requires the suffix f or F (e.g., 3.14f) to tell the compiler it's a float, otherwise it defaults to double.
double	99.99	8 bytes	Represents double-precision floating-point numbers. It is the default for decimal numbers in C# and offers greater precision than float.	Suitable for most scientific and financial calculations where precision is important. It is usually the default literal (no suffix needed).
char	'A'	2 bytes	Stores a single Unicode character. It is always enclosed in single quotes.	Used for letters, digits, or symbols. The 2-byte size allows it to represent any Unicode character.
bool	true or false	1 byte	Stores a Boolean logical value. It can only be one of	Essential for all decision-making (conditional) statements like if and loops.

			two states: `true` or `false`.	
`string`	**`"Hello"`**	**Varies**	**Stores a sequence of text characters. It is technically a reference type, but is often treated as a primitive for ease of use.**	**Always enclosed in double quotes. C# strings are immutable (cannot be changed after creation; any modification creates a new string).**

💡 Note on `decimal`

While `float` and `double` are used for general-purpose fractional numbers, C# also offers the `decimal` type (16 bytes). You use `decimal` for financial and currency calculations where extremely high precision is required, as it avoids the rounding errors common in standard binary floating-point types (`float` and `double`).

Operators

Operators are symbols used to perform operations on variables and values (called operands). C# groups its operators into several categories based on the type of operation they perform.

- Arithmetic: `+ - * / %`
- Comparison: `== != < > <= >=`
- Logical: `&& || !`
- Assignment: `= += -=`

1. Arithmetic Operators

These operators are used to perform common mathematical operations.

Operator	Name	Description	Example
`+`	Addition	Adds two operands.	`10 + 5` results in `15`
`-`	Subtraction	Subtracts the right operand from the left.	`10 - 5` results in `5`
`*`	Multiplication	Multiplies two operands.	`10 * 5` results in `50`
`/`	Division	Divides the left operand by the right.	`10 / 5` results in `2`
`%`	Modulus	Returns the **remainder** of a division.	`10 % 3` results in `1`

2. Comparison (Relational) Operators

These operators compare two operands and return a **Boolean** result (`true` or `false`). They are essential for conditional logic.

Operator	Name	Description	Example

`==`	Equal to	Checks if two operands are equal.	`a == b`
`!=`	Not equal to	Checks if two operands are not equal.	`a != b`
`<`	Less than	Checks if the left operand is less than the right.	`a < b`
`>`	Greater than	Checks if the left operand is greater than the right.	`a > b`
`<=`	Less than or equal to	Checks if the left is less than or equal to the right.	`a <= b`
`>=`	Greater than or equal to	Checks if the left is greater than or equal to the right.	`a >= b`

3. Logical Operators

`These operators are used to combine multiple Boolean conditions and return a single Boolean result.`

Operator	Name	Description	Example
`&&`	Logical AND	Returns `true` if **both** operands are `true`.	`(a > 5) && (b < 10)`

**`		`**	Logical OR
!	Logical NOT	Inverts the logical state (e.g., changes true to false).	!(a == 10)

4. Assignment Operators

These operators are used to assign a value to a variable.

Operator	Name	Description	Example	Equivalent to...
=	Simple Assignment	Assigns the value on the right to the variable on the left.	a = 10	
+=	Add and Assign	Adds the right operand to the left operand and assigns the result to the left operand.	a += 5	a = a + 5
-=	Subtract and Assign	Subtracts the right operand from the left operand and assigns the result to the left operand.	a -= 5	a = a - 5

*=	Multiply and Assign	Multiplies the right operand by the left operand and assigns the result to the left operand.	a *= 5	a = a * 5

Example Code Execution

```
int a = 10, b = 20;
Console.WriteLine(a + b);
```

Output:

```
30
```

This output is the result of the + (Addition Arithmetic Operator) acting on the variables a (10) and b (20).

---

7. Simple Programs

✔ Program 1: Area of Circle

Formula: Area = πr²

```
using System;


class Program

{

    static void Main()

    {

        Console.Write("Enter r: ");
```

```
        double r = Convert.ToDouble(Console.ReadLine());


        double area = Math.PI * r * r;


        Console.WriteLine("Area of circle = " + area);
    }
}
```

This program demonstrates fundamental C# concepts: reading user input, converting data types, using built-in mathematical constants, and performing calculations.

Input Request:

Console.Write("Enter Radius: ");

1. The Console.Write method prompts the user to input a value.

Reading and Conversion:

double r = Convert.ToDouble(Console.ReadLine());

2.
    - Console.ReadLine() reads the text (string) entered by the user.
    - Since the radius can be a decimal, the input string must be converted to a numeric type. Convert.ToDouble() converts the input string into a double (a high-precision floating-point number), which is then stored in the variable r.

Calculation:

double area = Math.PI * r * r;

3.
    - This line implements the area formula: A = pi .r^2.

- Math.PI is a constant provided by the System.Math class. Using this constant ensures the most accurate value for pi.
- The multiplication operator (*) is used twice to calculate r squared (r x r). The result is stored in the double variable area.

Output:

```
Console.WriteLine($"Area of Circle = {area}");
```

4.

- The string interpolation syntax ($"") is used to embed the calculated area value directly within the output string for a clean presentation.

This program ensures that the radius is treated as a floating-point number, guaranteeing a precise calculation of the circle's area.

✔ Program 2: Feet to Centimeters

1 foot = 30.48 cm

```
using System;


class Program
{
   static void Main()
   {
      Console.Write("Enter value in feet: ");
      double feet = Convert.ToDouble(Console.ReadLine());


      double cm = feet * 30.48;
```

```
        Console.WriteLine("Centimeters = " + cm);
    }
}
```

This program demonstrates a practical unit conversion, focusing on handling floating-point numbers accurately and applying a simple multiplicative conversion factor.

1. Conversion Constant: The core of the program is the conversion factor: 1 foot = 30.48 cm. This is the constant multiplier used in the calculation.

2. Input Request:

Console.Write("Enter value in Feet: ");

This line uses `Console.Write` to display a message, asking the user for the length they wish to convert.

3. Reading and Conversion:

double feet = Convert.ToDouble(Console.ReadLine());

The user's input (a string) is read using `Console.ReadLine()`.

It is immediately converted to a `double` and stored in the variable `feet`. Since length measurements often include decimal points, `double` is the appropriate type to maintain precision.

4. Calculation:

double cm = feet * 30.48;

- The arithmetic multiplication operator (`*`) is used. The input value in `feet` is multiplied by the conversion factor (`30.48`).
- The result, which is the length in centimeters, is stored in the `double` variable `cm`.

Output:

Console.WriteLine("Centimeters = {cm}");

- The result is displayed using string interpolation (`$""`), which embeds the calculated `cm` value directly into the output line for the user.

This program effectively illustrates how C# handles simple linear conversions, relying on the `double` type for precise decimal arithmetic.

✔ Program 3: Convert Seconds to Minutes

```
using System;

class Program
{
   static void Main()
   {
      Console.Write("Enter time in seconds: ");
      int sec = Convert.ToInt32(Console.ReadLine());

      double minutes = sec / 60.0;

      Console.WriteLine("Minutes = " + minutes);
   }
}
```

This program demonstrates time unit conversion and highlights a crucial concept in C# programming: **type compatibility and floating-point division**.

Input Request:

```
Console.Write("Enter seconds: ");
```

This line prompts the user for the number of seconds to be converted.

Reading and Integer Conversion:

```
int sec = Convert.ToInt32(Console.ReadLine());
```

- The input is read as a string.
- Since time units like seconds are typically whole numbers, the input is converted to an **`int`** (integer) and stored in the variable `sec`.

Critical Calculation - Floating-Point Division:

double minutes = sec / 60.0;

- This is the most important part. To convert seconds to minutes, we divide by 60.
- **The expression uses `60.0` (a `double`) instead of just `60` (an `int`).**
- **Why this is necessary:** In C#, dividing an `int` by another `int` (e.g., `sec / 60`) performs **integer division**, which discards any remainder, resulting in an inaccurate whole number.
- By including the decimal point (`60.0`), the divisor is treated as a `double`, forcing C# to perform **floating-point division**. This ensures the result, stored in the `double` variable `minutes`, includes the fractional part and is therefore accurate (e.g., 90 seconds / 60.0 = 1.5 minutes).

Output:

Console.WriteLine("Minutes = {minutes}");

The final, accurate decimal value for the minutes is displayed to the console using string interpolation.

This program is a great illustration of how crucial it is to use the correct data types when performing division to avoid unintentional truncation.

Assignment

1: Basic Level (Finance Calculations)

1. Simple Interest Calculator
Write a C# program to calculate **Simple Interest**
Formula:

```
SI = (P * R * T) / 100
```

2. Currency Conversion (INR → USD)
User inputs INR, convert using fixed rate (₹1 = $0.012 approx).

3. EMI Calculator (Without reducing balance)
Use basic formula: EMI = Loan / Months

2: Medium Level (Insurance Basics)

1. Health Insurance Eligibility Check
Input: Age
Rules:

- Age < 18 → Not eligible
- 18–60 → Eligible
- 60+ → Senior Citizen Plan

2. Vehicle Insurance Premium Calculator
Formula example:

```
Premium = BaseAmount + (AgeFactor * VehicleAge) + GST
```

3. Credit Score Classification
Input score (300–900):

- ≥750 → Excellent
- ≥650 → Good
- ≥550 → Average
- else → Poor

Starter Code Example (Vehicle Insurance Premium):

```
double baseAmount = 3000;
double ageFactor = 200;

Console.Write("Enter vehicle age: ");
int vAge = Convert.ToInt32(Console.ReadLine());
```

```
double premium = baseAmount + (ageFactor * vAge);
double totalWithGST = premium * 1.18;

Console.WriteLine("Insurance Premium = " + totalWithGST);
```

Assignment 3: Logical Level (Finance Algorithms)

1. Generate Monthly SIP Investment Growth (N months)
SIP = fixed monthly investment
Interest = monthly rate

2. Reverse an Account Number (Algorithmic)
Without converting to string.

3. Detect Fraud Transaction (Pattern-Based)
If transaction amount > 50,000 and location changes suddenly → "Flag Fraud".

Example (Reverse Account Number):

```
int acc = 123456, rev = 0;

while(acc > 0)
{
    int digit = acc % 10;
    rev = rev * 10 + digit;
    acc /= 10;
}
Console.WriteLine("Reversed: " + rev);
```

Use Case: Insurance Premium & Claim Eligibility System

A real-world style mini-project.

Scenario

An Insurance Company needs a **console-based C# tool** to:

✔ Take customer details
✔ Calculate monthly premium based on age
✔ Determine plan type
✔ Check if user qualifies for insurance claim
✔ Show summary report

Useful for: **Life Insurance / Health Insurance Training**.

Rules (Business Logic)

Premium Rule (Age-based)

Age Range	Premium (per month)
18–30	₹1200
31–50	₹1800
51–60	₹2500
> 60	₹3000

Claim Eligibility

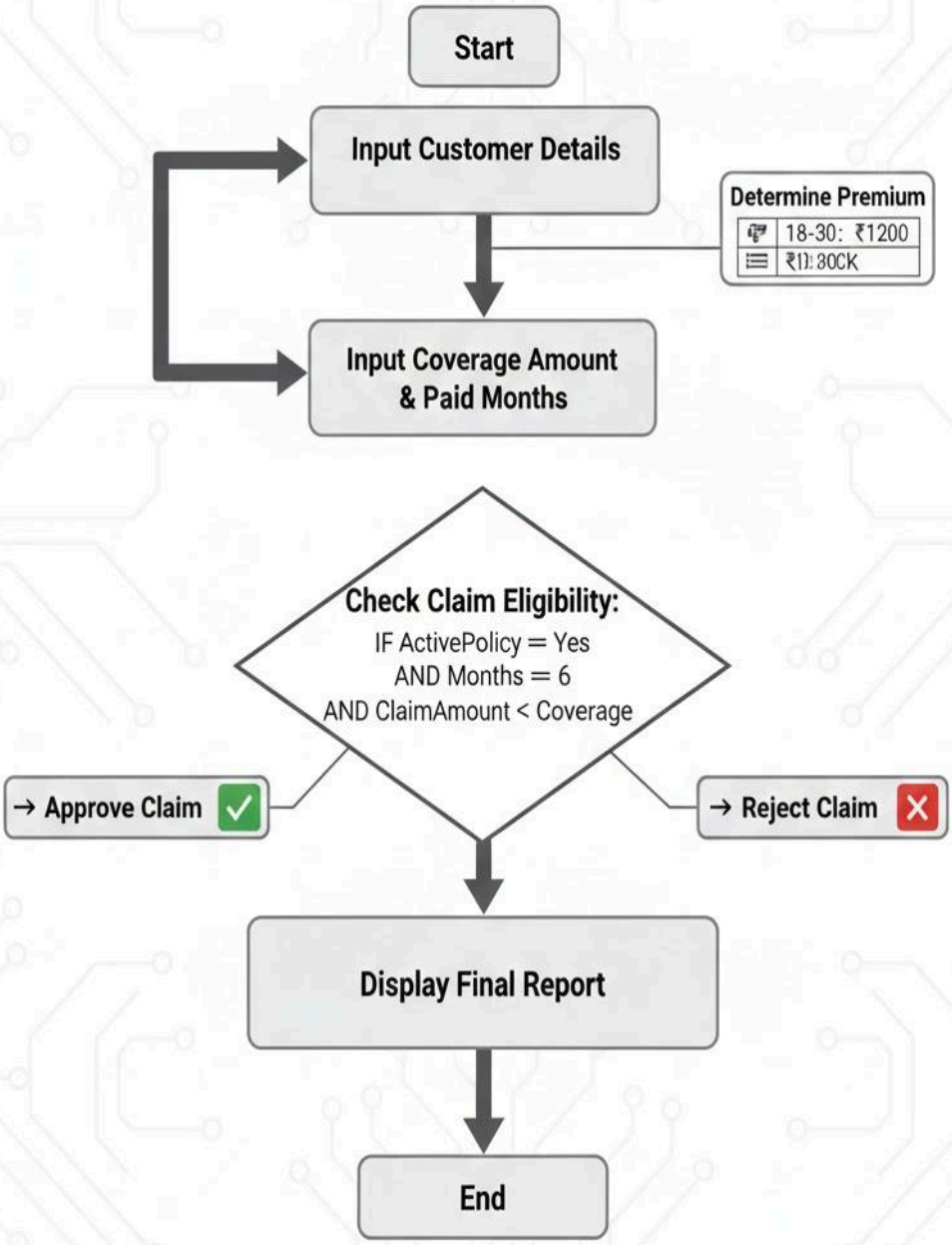
A claim is approved only if:

- Policy active = Yes
- Premium paid months ≥ 6
- Claim amount < coverage

Flow Diagram (Text Version)

```
Start
    ↓
Input Customer Details
    ↓
Input Age → Determine Premium
    ↓
Input Coverage Amount & Premium Months
    ↓
Check Claim Eligibility:
   IF ActivePolicy == Yes
   AND Months >= 6
   AND ClaimAmount < Coverage
       → Approve Claim
   ELSE → Reject Claim
    ↓
Display Final Report
    ↓
End
```

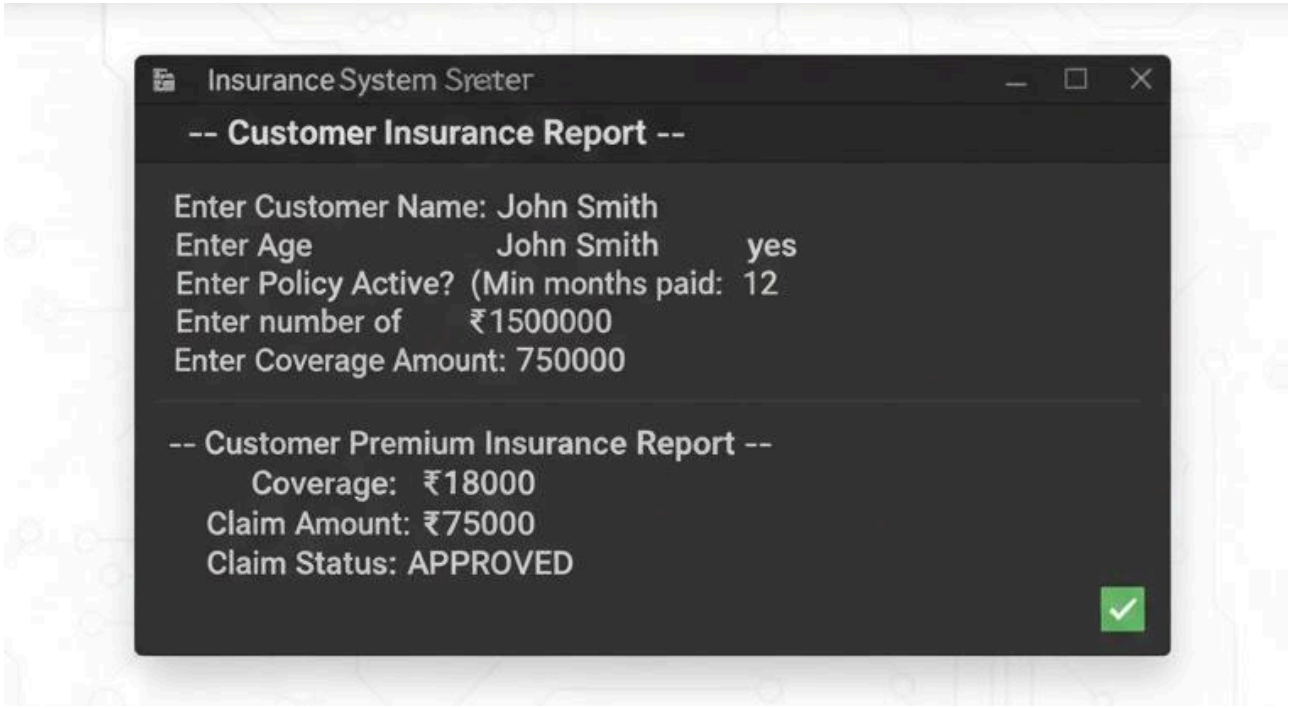
8. Insurance Premium & Claim System



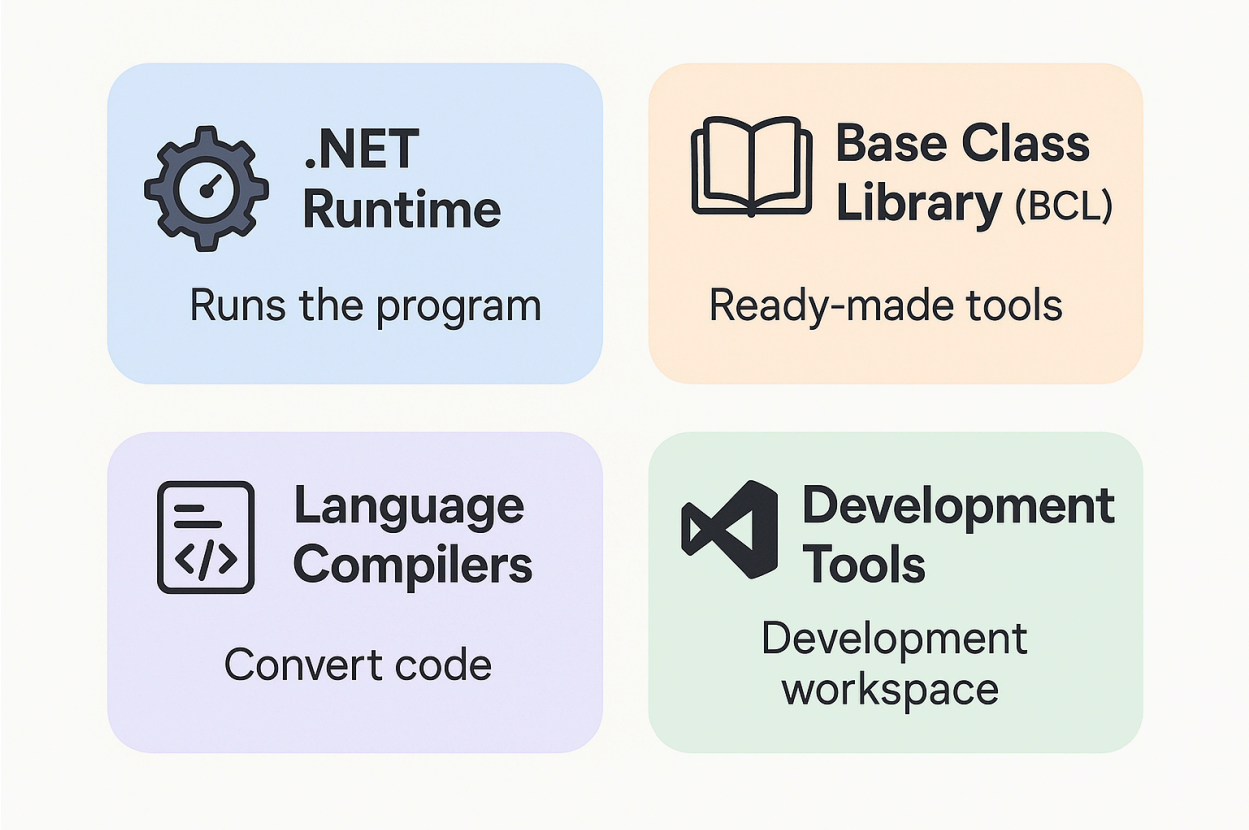
Full C# Implementation (Insurance Premium Use Case)

```
using System;

class InsuranceSystem
{
    static void Main()
    {
        Console.WriteLine("--- Insurance Premium & Claim System ---");

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

        Console.Write("Enter Age: ");
        int age = Convert.ToInt32(Console.ReadLine());

        Console.Write("Is Policy Active? (yes/no): ");
        string policy = Console.ReadLine().ToLower();

        Console.Write("Enter number of months paid: ");
        int months = Convert.ToInt32(Console.ReadLine());

        Console.Write("Enter Coverage Amount: ");
        double coverage = Convert.ToDouble(Console.ReadLine());

        Console.Write("Enter Claim Amount: ");
        double claim = Convert.ToDouble(Console.ReadLine());

        // Premium Calculation
        double premium = age switch
        {
            >= 18 and <= 30 => 1200,
            >= 31 and <= 50 => 1800,
```

```
            >= 51 and <= 60 => 2500,
            _ => 3000
        };

        // Claim Eligibility
        string claimStatus =
            (policy == "yes" && months >= 6 && claim < coverage)
            ? "APPROVED"
            : "REJECTED";

        // Report
        Console.WriteLine("\n--- Customer Insurance Report ---");
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"Age: {age}");
        Console.WriteLine($"Monthly Premium: ₹{premium}");
        Console.WriteLine($"Coverage: ₹{coverage}");
        Console.WriteLine($"Claim Amount: ₹{claim}");
        Console.WriteLine($"Claim Status: {claimStatus}");
    }
}
```



.NET runtime



---

1. .NET Runtime (CLR / Core CLR)

What it is

The .NET Runtime is the engine that runs your .NET program.

Think of it like:

What it does

- Converts your compiled code into machine code

- Manages memory (automatic garbage collection)
- Handles exceptions (errors)
- Ensures security and performance

Example

You write a C# program:

Console.WriteLine("Hello World");

The runtime:

- Executes this line
- Allocates memory for the program
- Cleans memory after program ends

Real-life use case

- Running a banking application
- Running a hospital management system
- Running a web API

Without the runtime, the program cannot run.

---

2. Base Class Library (BCL)

What it is

The BCL is a huge collection of ready-made classes and functions provided by .NET.

Think of it like:

What it contains

- File handling
- Data types
- Collections (List, Dictionary)
- Date and time
- Input/output
- Networking

Example

Instead of writing code to open a file from scratch, you use:

File.ReadAllText("data.txt");

This method comes from the BCL.

Real-life use case

- Reading employee data from files
- Sending emails
- Working with dates (salary calculation, attendance)

BCL saves time and reduces errors.

---

3. Language Compilers

What they are

Compilers convert your code into Intermediate Language (IL) which the runtime understands.

.NET supports multiple languages, such as:

- C#
- VB.NET
- F#

Each language has its own compiler.

Think of it like:

Example

You write C# code:

int x = 10;

The C# compiler:

- Converts it into IL code
- This IL code is later run by the .NET Runtime

Real-life use case

- Teams using different languages can work on the same project
- One module in C#, another in VB.NET

All work together because of the common compiler and runtime.

---

4. Development Tools (Visual Studio)

What they are

Development tools help developers write, test, debug, and deploy applications easily.

Visual Studio provides

- Code editor
- Error highlighting
- Debugging tools
- Project templates
- Build and publish options

Think of it like:

Example

Using Visual Studio, you can:

- Create a Console App
- Create a Web App
- Create a Desktop App

  with just a few clicks.

Real-life use case

- Developing enterprise software
- Debugging large applications
- Team collaboration using Git

Visual Studio increases productivity and reduces development time.

.dll files

What are .DLL files?

.DLL stands for Dynamic Link Library.

A .dll file is a collection of reusable code (functions, classes, methods) that multiple programs can use without copying the code again and again.

Think of it like:

Why are .DLL files used?

- To reuse code
- To reduce program size
- To make applications modular
- To allow easy updates

Simple Example

Suppose you write code to:

- Calculate tax
- Format dates
- Connect to a database

Instead of writing this code in every program, you put it in a TaxUtility.dll.

Now:

- Accounting App uses it
- Payroll App uses it
- Billing App uses it

All programs share the same DLL.

.DLL in .NET (Important)

In .NET:

- Your C# code is compiled into IL (Intermediate Language)
- This compiled code is stored in a .dll file or .exe file

Both .dll and .exe contain IL code.

Difference:

- .exe → runs directly
- .dll → used by other programs

Real-Life Use Cases

- System DLLs

Example: System.dll, mscorlib.dll

Used by almost every .NET application

- Application DLLs

  Business logic stored in separate DLLs

- Third-party libraries

  Example: logging, payment gateway libraries

Advantages of .DLL Files

1. Code reuse – write once, use many times
2. Easy maintenance – update DLL, all apps benefit
3. Smaller applications
4. Faster development

Disadvantages (For Exams)

- Missing DLL causes runtime error
- Version mismatch can cause issues (DLL Hell)

Simple Diagram Explanation (Text)

Application
  |
  |--- Uses ---> Utility.dll
  |--- Uses ---> Database.dll

One-Line Exam Answer

A .dll file is a reusable library containing compiled code that can be shared by multiple programs at runtime.

Difference: .DLL vs .EXE (Quick Table)

Feature	.DLL	.EXE
Runs directly	No	Yes
Reusable	Yes	Limited
Contains code	Yes	Yes
Used by other apps	Yes	No

JIT

What is JIT Compilation?

JIT stands for Just-In-Time Compilation.

JIT compilation is the process where Intermediate Language (IL) code is converted into machine code at runtime, just before execution.

Think of it as:

How JIT Works (Step by Step)

1. You write code in C#, VB.NET, etc.
2. The compiler converts it into IL code
3. The IL code is stored in a .dll or .exe
4. When the program runs, JIT compiler converts IL into machine code
5. CPU executes the machine code

Simple Flow Diagram (Text)

C# Code
  ↓
IL Code (.exe / .dll)
  ↓
JIT Compiler (at runtime)
  ↓
Machine Code
  ↓
Execution

Simple Example

C# code:

```
int a = 10;
int b = 20;
int c = a + b;
```

- This code is stored as IL
- When execution reaches this line,
- JIT compiles it into machine code
- CPU executes it

Why is JIT Used?

- Makes .NET platform independent
- Optimizes code based on current machine
- Improves performance after first execution

Real-Life Use Cases

- Desktop applications (Windows apps)
- Web applications (ASP.NET)

- Enterprise systems
- Cloud applications

Every time a .NET app runs, JIT compilation happens.

Types of JIT Compilation (Basic Awareness)

1. Normal JIT – Compiles methods when they are called
2. Pre-JIT – Compiles entire code at startup
3. Tiered JIT – Starts fast, optimizes later (used in modern .NET)

Advantages of JIT

- Platform independence
- Better runtime optimization
- Smaller executable size

Disadvantages (Exam Point)

- Slight delay at first execution
- Requires runtime environment

JIT vs Compiler (Quick Difference)

Feature	Compiler	JIT
When it works	Before execution	During execution
Output	IL code	Machine code
Speed	Fast compilation	Optimized execution

One-Line Exam Answer

JIT compilation converts Intermediate Language code into machine code at runtime, just before execution.

What is IL Code?

IL (Intermediate Language) is a low-level, CPU-independent code generated by .NET compilers.

- Also called MSIL or CIL
- Not human-friendly like C#
- Not machine code either

Think of IL as:

Simple C# Code Example

C# Code

```
int a = 10;

int b = 20;

int c = a + b;

Console.WriteLine(c);
```

How IL Code Looks (Simplified)

```
.locals init ([0] int32 a,

              [1] int32 b,

              [2] int32 c)


ldc.i4.s   10

stloc.0


ldc.i4.s   20

stloc.1


ldloc.0

ldloc.1
```

```
add

stloc.2


ldloc.2

call void [System.Console]::WriteLine(int32)
```

Meaning of IL Instructions (Very Simple)

IL Instruction	Meaning
ldc.i4.s 10	Load constant 10
stloc.0	Store in variable a
ldloc.0	Load variable a
add	Add two values
call	Call a method

IL uses a stack-based execution model.

Another Simple Example

C# Code

```
Console.WriteLine("Hello");
```

IL Code

```
ldstr "Hello"

call void [System.Console]::WriteLine(string)
```

Key Characteristics of IL Code

- Stack-based (push and pop values)
- CPU independent
- Generated by .NET compilers
- Stored inside .exe or .dll
- Converted to machine code by JIT

Where Can You See IL Code?

You can view IL code using:

- ILDASM (IL Disassembler)
- dotPeek
- ILSpy

Simple Comparison

Level	Language
High-level	C#, VB.NET
Medium-level	IL
Low-level	Machine Code

One-Line Exam Answer

IL code is an intermediate, low-level code generated by .NET compilers, which is later converted into machine code by JIT at runtime.

Intermediate Code

Why was Intermediate Code Needed?

Languages like .NET and Java use Intermediate Code (IL / Bytecode) instead of directly converting code into machine code.

Simple reason:

Problem with Direct Machine Code

If code is compiled directly to machine code:

- Machine code is hardware specific
- One CPU ≠ another CPU
- Same program must be compiled separately for:
    - Windows
    - Linux
    - Mac
    - Different processors (x86, ARM)

Example

If C# compiled directly to machine code:

Platform	Need

Windows (x64)	Separate build
Linux	Separate build
macOS	Separate build

This is time-consuming and costly.

Solution: Intermediate Code

What Intermediate Code Does

Intermediate code acts as a middle layer between:

- High-level language
- Machine-specific code

Flow

High-level code
↓
Intermediate Code (IL / Bytecode)
↓
Runtime (CLR / JVM)
↓
Machine Code

Reasons Why Intermediate Code Is Used

1. Platform Independence

Write code once → run anywhere.

- Same IL runs on Windows, Linux, macOS
- Only the runtime changes

Example:

- Same .dll runs on Windows and Linux using .NET runtime

---

2. Language Independence

Multiple languages → same intermediate code.

- C#, VB.NET, F# → IL
- Java, Kotlin → Bytecode

This allows:

- Code reuse
- Mixed-language projects

---

3. Runtime Optimization (JIT)

Intermediate code allows Just-In-Time compilation:

- Code optimized based on:

    - CPU type
    - Memory
    - Runtime behavior

Result:

- Better performance than static machine code

---

4. Security & Verification

Intermediate code can be:

- Verified for safety
- Checked for type safety
- Prevents illegal memory access

Direct machine code:

- Cannot be easily verified

---

5. Easier Maintenance & Updates

- Runtime can be upgraded
- Applications automatically benefit

- No recompilation required

Simple Analogy

Without Intermediate Code

With Intermediate Code

Comparison Table (Exam Friendly)

Aspect	Direct Machine Code	Intermediate Code
Platform independent	No	Yes
Reusability	Low	High
Optimization	Limited	High (JIT)
Security checks	No	Yes
Language support	One	Multiple

One-Line Exam Answer

Intermediate code is used to achieve platform independence, language interoperability, runtime optimization, and security, which is not possible with direct machine code.

Easy Memory Trick

- Direct machine code = fast but fixed
- Intermediate code = flexible and portable

C# Compilation with Respect to Different Hardware

Below is a hardware-focused explanation of how C# code becomes different machine code on different CPUs.

Step-by-Step C# Compilation Flow

1. C# Source Code (Hardware-Independent)

```
int a = 5;
int b = 3;
int c = a + b;
```

At this stage:

- Pure C# code
- No dependency on CPU (x86, ARM, etc.)

---

2. C# Compiler (csc) → IL Code

The C# compiler converts code into IL (Intermediate Language).

IL code (same for all hardware):

IL_0000: ldc.i4.5
IL_0001: stloc.0
IL_0002: ldc.i4.3
IL_0003: stloc.1
IL_0004: ldloc.0
IL_0005: ldloc.1
IL_0006: add
IL_0007: stloc.2

Key point:

---

3. CLR + JIT → Machine Code (Hardware-Dependent)

When the program runs:

- CLR invokes JIT (Just-In-Time) compiler
- JIT converts IL → machine code for the current CPU

Same C# Program → Different Machine Code

On x86 / x64 (Intel / AMD)

Assembly:

mov eax, 5
add eax, 3

Machine code (hex):

B8 05 00 00 00
83 C0 03

On ARM (Mobile / ARM PCs)

Assembly:

MOV R0, #5
ADD R0, R0, #3

Machine code (hex):

E3A00005
E2800003

Why This Works So Well in C#

Because of two-step compilation:

Stage	Output	Hardware Dependency
C# → IL	IL (.exe/.dll)	Independent
IL → Machine Code	Native code	CPU-specific

Key Advantages of This Model

1. Same .exe runs on:

    - Intel
    - AMD
    - ARM
2. No recompilation needed by developer
3. Runtime optimizations by JIT
4. Platform portability

Very Important Exam Statement

Visual Summary (Memory Hook)

C# Code
  ↓ (csc)
IL Code (.exe / .dll)
  ↓ (CLR + JIT)
x86 Machine Code OR ARM Machine Code

Why This Was a Big Deal (Interview Insight)

Before .NET:

- One compiler → one hardware
- Recompile needed for each CPU

With .NET:

- Write once → run anywhere CLR exists

One-Line Answer (Perfect for Viva / Exam)

Console Class

In C#, `Console` refers to a **built-in class** provided by the **.NET framework** that represents the **command-line window (console window)** where a program can interact with the user using text.

What is `Console` in C#?

`Console` is a **class** defined in the **System** namespace.

using System;

When you write:

Console.WriteLine("Hello");

you are telling the program to **display text in the console window**.

What does the console mean?

The **console** is the **black/white text-based window** where:

- Output is shown as text
- Input is taken from the keyboard

This window appears when you run a **Console Application**.

Why is it called `Console`?

Historically, computers interacted through **text terminals** (called consoles).
C# keeps this concept for learning, testing, and backend programs.

So:

- `Console` = text-based input/output screen
- Not a graphical window
- Not a web page

What does `Console.WriteLine()` do?

Console.WriteLine("Welcome");

- `Console` → the console window
- `WriteLine()` → prints text and moves to a new line

Output:

Welcome

How is `Console` used?

1. Display output

Console.WriteLine("Result is 10");

2. Take input

string name = Console.ReadLine();

3. Combine input and output

Console.WriteLine("Enter your age:");
int age = Convert.ToInt32(Console.ReadLine());

Where is `Console` used in real life?

Learning and practice

- Beginner programming
- Understanding logic and flow

Backend / system programs

- Log processing
- Data migration scripts
- Automation tools

Interviews and exams

- Most coding questions use console input/output

Why not use UI instead of Console?

Because:

- Console programs are **simple and fast**
- No UI design required
- Easy to test logic
- Used heavily in **backend development**

Below is a **complete, exam-ready list of important `Console` class functions in C#**, explained in **simple words**, with **what / why / where / how / when**, plus **real-life examples**. No icons or emojis are used.

What is the Console class

The `Console` class is a built-in class in the `System` namespace used for **text-based input and output** in C# console applications.

using System;

Why Console functions are important

Console functions are used to:

- Take user input
- Display output
- Control cursor, screen, and colors
- Pause program execution
- Show error messages

They are heavily used in:

- Learning C#
- Exams and interviews
- Backend scripts
- System utilities

Where Console functions are used

- Console Applications
- Backend programs
- Automation scripts
- Debugging and logging
- Competitive programming

Important Console Functions (Complete List)

---

1. `Console.WriteLine()`

What
Prints text and moves to the next line.

Why
Used to display messages and results.

How

```
Console.WriteLine("Hello World");
```

When
Whenever output should appear on a new line.

Example

```
Console.WriteLine("Total = " + total);
```

---

2. `Console.Write()`

What
Prints text without moving to a new line.

Why
Used when input is expected on the same line.

How

```
Console.Write("Enter name: ");
```

When
Before `ReadLine()`.

---

3. `Console.ReadLine()`

What
Reads a full line of input as a string.

Why
Used to take user input.

How

```
string name = Console.ReadLine();
```

When
Whenever input is required.

---

4. `Console.Read()`

What
Reads a single character (ASCII value).

Why
Used for low-level input.

How

int ch = Console.Read();

When
Rarely used, mostly in system programs.

---

5. `Console.ReadKey()`

What
Reads a key pressed by the user.

Why
Used to pause the program.

How

Console.ReadKey();

When
At the end of programs.

---

6. `Console.Clear()`

What
Clears the console screen.

Why
Keeps output clean.

How

Console.Clear();

When
Before showing new output.

---

7. `Console.Beep()`

What
Produces a beep sound.

Why
Alert or notification.

How

```
Console.Beep();
```

When
Warnings or alerts.

---

8. `Console.SetCursorPosition()`

What
Moves the cursor to a specific position.

Why
Used for formatted output.

How

```
Console.SetCursorPosition(10, 5);
```

When
Games or dashboards.

---

9. `Console.ForegroundColor`

What
Changes text color.

Why
Highlights output.

How

Console.ForegroundColor = ConsoleColor.Green;

When
Errors, success messages.

---

10. `Console.BackgroundColor`

What
Changes background color.

Why
Visual distinction.

How

Console.BackgroundColor = ConsoleColor.Black;

---

11. `Console.ResetColor()`

What
Resets colors to default.

Why
Avoids color leakage.

How

Console.ResetColor();

---

12. `Console.Title`

What
Sets console window title.

Why
Identifies the program.

How

```
Console.Title = "Student App";
```

---

13. `Console.WindowHeight` and `Console.WindowWidth`

What
Gets or sets window size.

Why
UI control.

How

```
Console.WindowWidth = 100;
Console.WindowHeight = 30;
```

---

14. `Console.BufferHeight` and `Console.BufferWidth`

What
Controls scroll buffer size.

Why
Prevents scrolling issues.

---

15. `Console.Error.WriteLine()`

What
Prints error messages.

Why
Separates errors from normal output.

How

```
Console.Error.WriteLine("Invalid input");
```

---

16. `Console.Out.WriteLine()`

What
Explicit standard output.

Why
Advanced output control.

---

17. `Console.In.ReadLine()`

What
Explicit standard input.

Why
Advanced input handling.

---

18. `Console.KeyAvailable`

What
Checks if a key is pressed.

Why
Non-blocking input.

How

```
if(Console.KeyAvailable)
{
   Console.ReadKey();
}
```

Real-Life Simple Example

Student marks program

```
Console.Write("Enter marks: ");
int marks = Convert.ToInt32(Console.ReadLine());

if(marks >= 40)
```

```
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("Pass");
}
else
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Fail");
}

Console.ResetColor();
Console.ReadKey();
```

Float vs Double | Char vs String

Difference between float and double

Feature	float	double
Size	4 bytes	8 bytes
Precision	About 7 digits	About 15–16 digits
Accuracy	Lower	Higher
Memory usage	Less	More
Speed	Slightly faster	Slightly slower
Suffix used in C#	Must write f (example: 3.14f)	No suffix needed (3.14)
When to use?	When memory matters, low precision is okay	When high precision is required

Quick example

```
float x = 3.1415926f;
double y = 3.14159265358979;
```

Difference between char and string

Feature	char	string
Represents	A single character	A sequence of characters (text)
Size	2 bytes	Depends on length
Notation	Single quotes 'A'	Double quotes "Apple"
Type	Value type	Reference type
Examples	'a', '7', '@'	"Hello", "C# Programming"

Quick example

```
char grade = 'A';       // only one character
string name = "Aman";   // multiple characters
```

Namespace in C#

What is a namespace in C#?

A **namespace** in C# is a **container (or group)** that is used to **organize classes, interfaces, structs, enums, and other namespaces**.

It helps keep code **well-structured**, **readable**, and **conflict-free**.

Example:

using System;

Here, `System` is a **namespace**.

What is a namespace?

A **namespace** is a **logical container** used to **group related classes and other types**.

- It does **not** contain logic
- It does **not** have methods
- It does **not** get executed
- It **cannot be instantiated**

Why do we need namespaces?

Namespaces solve three major problems:

1. Avoid name conflicts

Two classes can have the same name if they are in different namespaces.

CompanyA.Employee
CompanyB.Employee

Without namespaces, this would cause confusion.

---

2. Organize large projects

As applications grow, namespaces help **group related code logically**.

Example:

- `Finance`
- `HR`
- `Inventory`

---

3. Improve code readability and maintenance

Namespaces make it easy to:

- Understand where a class belongs
- Maintain large applications
- Share libraries safely

Where are namespaces used?

Namespaces are used:

- In **every C# application**
- In **.NET libraries**
- In **enterprise and enterprise-level projects**
- In **frameworks like ASP.NET, Entity Framework, etc.**

Example:

System
System.IO
System.Collections
System.Linq

How does a namespace work?

Syntax of a namespace

```
namespace MyApplication
{
   class Program
   {
      static void Main()
      {
         Console.WriteLine("Hello");
      }
   }
}
```

Here:

- `MyApplication` is the namespace
- `Program` is a class inside that namespace

How to access a namespace?

Method 1: Using `using` keyword

```
using System;

Console.WriteLine("Hello");
```

Method 2: Fully qualified name

```
System.Console.WriteLine("Hello");
```

Used when:

- Two namespaces contain the same class name

What is the `using` keyword?

The `using` keyword tells the compiler:

> "Classes from this namespace may be used directly."

Example:

```
using System;
using System.IO;
```

Without `using`, you must write:

```
System.Console.WriteLine("Hello");
```

Real-life simple example

Real-life analogy: Library

Think of a **library**:

- Sections → Namespaces
- Books → Classes

You go to the **Science section** to find science books.
Similarly, you go to the **System.IO namespace** to find file-related classes.

Example with multiple namespaces

```
namespace Finance
{
   class Account
   {
      public void ShowBalance() { }
   }
}
```

```
namespace HR
{
    class Account
    {
        public void ShowEmployeeDetails() { }
    }
}
```

Both classes are named `Account`, but there is **no conflict** because they are in different namespaces.

Nested namespaces

Namespaces can exist inside other namespaces.

```
namespace Company.Project.Module
{
    class Test
    {
    }
}
```

This improves structure in large projects.

Built-in namespaces in C#

Some important built-in namespaces:

- `System` – Basic classes (Console, Math, Convert)
- `System.IO` – File handling
- `System.Collections` – Data structures
- `System.Linq` – Query operations
- `System.Data` – Database related classes

When should you create your own namespace?

You should create a namespace when:

- Your project has multiple modules
- You are building a library
- You want clean, professional code
- You want to avoid naming conflicts

Key differences: Namespace vs Class

Namespace	Class
Logical container	Blueprint of object
Groups related classes	Defines behavior
Cannot be instantiated	Can be instantiated

Summary (easy to remember)

- A **namespace is a container**
- It **organizes code**
- It **prevents naming conflicts**
- It improves **readability and maintenance**
- `using` helps access namespaces easily

Tab 9

Before .NET, application development—especially on Windows—had many serious issues.

.NET was created to fix these core problems.

---

1. Language Incompatibility Problem

Before .NET

- C++, VB, Java worked independently
- Difficult to make different languages work together
- Rewriting code was common

How .NET Solved It

- Introduced Common Language Runtime (CLR)
- All languages compile into Intermediate Language (IL)
- Full cross-language interoperability

Example:

C# class can be used in VB.NET without rewriting.

---

2. Memory Management Issues

Before .NET

- Manual memory allocation (malloc, free)
- Memory leaks
- Crashes due to dangling pointers

How .NET Solved It

- Automatic Garbage Collection
- CLR tracks unused objects and frees memory safely

Result:

More stable applications, fewer crashes.

---

3. Application Crashes & Stability Problems

Before .NET

- One application crash could affect others
- DLL conflicts (“DLL Hell”)

DLL Hell is a problem in Windows where applications stop working or behave incorrectly because of missing, incompatible, or overwritten DLL (Dynamic Link Library) files.

- Multiple applications share the same DLL
- One application installs or updates a DLL
- That DLL replaces an older version
- Other applications that depend on the old version break

This situation is called DLL Hell.

How .NET Solved It

- Application isolation
- Versioned assemblies
- Side-by-side execution

Result:

No more DLL Hell.

---

4. Security Weaknesses

Before .NET

- No controlled execution
- Unsafe memory access
- Easy to inject malicious code

How .NET Solved It

- Code Access Security
- Type safety
- Managed execution environment

Result:

Safer applications.

---

5. Platform Dependency & Tight OS Coupling

Before .NET

- Applications tightly bound to OS internals
- Hard to upgrade or maintain

How .NET Solved It

- CLR acts as a middle layer
- Hardware and OS abstraction

Result:

Easier upgrades and portability (especially with .NET Core later).

---

6. Slow Development & Code Duplication

Before .NET

- Developers wrote everything from scratch
- No unified standard libraries

How .NET Solved It

- Base Class Library (BCL)
- Thousands of ready-to-use classes

Result:

Faster development, cleaner code.

---

7. Poor Exception Handling

Before .NET

- Unstructured error handling
- Hard to trace runtime issues

How .NET Solved It

- Structured exception handling (try-catch-finally)
- Unified error model across languages

---

8. Deployment & Versioning Problems

Before .NET

- Complex installations
- Registry issues
- Version conflicts

How .NET Solved It

- Assembly-based deployment
- Private assemblies
- Easy updates

---

9. Scalability & Enterprise Application Issues

Before .NET

- Hard to build large, secure, scalable apps

How .NET Solved It

- ASP.NET for web apps
- Built-in threading, networking, database access

Result:

Enterprise-ready applications.

Quick Exam Summary Table

Problem Before .NET	How .NET Solved It

Language mismatch	Common IL + CLR
Manual memory	Garbage Collection
DLL Hell	Versioned assemblies
Weak security	Managed code + CAS
Slow development	BCL
Unstable apps	CLR control
Deployment issues	Side-by-side execution

One-Line Exam Answer

Tab 10

A memory leak happens when a program uses memory but does not release it even after it is no longer needed.

Over time, this wastes memory and slows down or crashes the system.

What Actually Goes Wrong

When a program runs:

1. It requests memory from RAM
2. Uses it to store data
3. Forgets to free it

That unused memory:

- Cannot be reused
- Keeps accumulating
- Reduces available RAM

Tab 11

Below are important terminal (command-line) commands specifically for C# (.NET), written simple, practical, and exam-ready.

These are the commands you actually use while creating, building, and running C# console applications.

---

1. Check C# / .NET Installation

dotnet --version

Shows installed .NET SDK version.

---

2. Create a New C# Console Application

dotnet new console

With project name:

dotnet new console -n MyFirstCSharpApp

---

3. Move into Project Folder

cd MyFirstCSharpApp

---

4. Run C# Program

dotnet run

Runs Program.cs.

---

5. Build C# Project (Compile Only)

dotnet build

Creates .exe without running.

---

6. Restore Project Dependencies

dotnet restore

---

7. Clean Build Files

dotnet clean

Removes bin and obj folders.

---

8. Create Solution File (Optional but Important)

dotnet new sln -n MySolution

Add project to solution:

dotnet sln add MyFirstCSharpApp.csproj

---

9. Add New C# Class File (Terminal Way)

dotnet new class -n AreaOfCircle

Creates:

AreaOfCircle.cs

---

10. Run Executable Directly

After build:

cd bin\Debug\net8.0
MyFirstCSharpApp.exe

---

11. List Installed SDKs

```
dotnet --list-sdks
```

---

12. Common File & Folder Commands (Windows)

```
dir          // list files
cd ..        // go back
mkdir Test   // create folder
cls          // clear screen
```

Typical C# Workflow Using Terminal

```
dotnet new console -n DemoApp
cd DemoApp
dotnet run
```

Exam-Ready One-Paragraph Answer

Must-Remember Commands (Viva / Exam)

```
dotnet new console
dotnet run
dotnet build
dotnet restore
dotnet clean
```

If you want next:

- C# compiler commands (csc.exe)
- Difference: dotnet CLI vs Visual Studio
- How terminal runs Program.cs internally
- Errors & fixes in C# terminal execution

Tell me what you want to learn next.