Is c sharp static or dynamic

C# Static vs. Dynamic

In C#, both static and dynamic types serve different purposes and have distinct use cases. Below is a detailed comparison:

1. Static in C#

A static type is determined at compile time, meaning the compiler knows the type and can enforce type safety.

Key Features of Static:

✅ Type checking happens at compile time (avoids runtime errors).

✅ Performance is better because there is no late binding.

✅ Works with classes, methods, properties, and variables.

✅ Static members belong to the class itself, not instances.

Example of Static Class & Method

public static class MathHelper

{

public static int Add(int a, int b)

{

return a + b;

}

}

// Usage

int sum = MathHelper.Add(5, 10);

Console.WriteLine(sum); // Output: 15

Static classes cannot be instantiated.

Static methods can be accessed without creating an object.

2. Dynamic in C#

A dynamic type is resolved at runtime, meaning the compiler does not check type safety during compilation.

Key Features of Dynamic:

✅ Type checking happens at runtime.

✅ Useful when working with reflection, COM objects, or external libraries (e.g., JSON/XML).

✅ Can change its type dynamically.

✅ Allows calling methods and properties that may not exist at compile time.

Example of Dynamic Variable

dynamic data = 10; // Initially an integer

Console.WriteLine(data); // Output: 10

data = "Hello"; // Now it's a string

Console.WriteLine(data); // Output: Hello

data = true; // Now it's a boolean

Console.WriteLine(data); // Output: True

The compiler does not check type safety.

The actual type is determined at runtime, which can lead to errors if a method/property does not exist.

3. Static vs. Dynamic Comparison Table

Feature Static Dynamic

Type Checking Compile-time Runtime

Performance Faster (type safety enforced) Slower (late binding)

Flexibility Less flexible (fixed types) More flexible (changes type at runtime)

Error Detection Errors caught at compile time Errors only appear at runtime

Usage Scenario General programming, performance-critical applications Reflection, dynamic objects, working with unknown data sources

4. When to Use What?

✅ Use Static When:

You need better performance.

You want compile-time type safety.

You're working with well-defined types.

✅ Use Dynamic When:

You are dealing with reflection (e.g., accessing private members of a class dynamically).

You are working with JSON, XML, or COM objects.

You need flexibility where type changes at runtime.

5. Example: Static vs. Dynamic in Real-World Use

Static Example (Compile-time safety)

string text = "Hello";

int length = text.Length; // Compiler knows "text" is a string and has "Length"

Dynamic Example (Runtime flexibility)

dynamic text = "Hello";

int length = text.Length; // Works fine

text = 123;

length = text.Length; // Runtime Error: 'int' does not have 'Length' property

The static approach prevents errors early, whereas dynamic can introduce runtime errors.

Final Thoughts

Prefer static types for performance and safety.

Use dynamic types when working with unknown or flexible data structures.

Mix static and dynamic where necessary, but avoid excessive use of dynamic to prevent runtime errors.

Javascript is dynamic

Common Language Runtime (CLR) in C#

The Common Language Runtime (CLR) is the execution environment of the .NET framework that manages and runs .NET applications. It is responsible for converting C# (or other .NET languages) code into machine code that can be executed by the operating system.

Key Responsibilities of CLR

Compilation & Execution

Converts C# code → Intermediate Language (IL) → Machine Code using Just-In-Time (JIT) compilation.

Memory Management

Automatically manages memory allocation and deallocation using Garbage Collection (GC).

Type Safety

Ensures that objects are used correctly according to their types (e.g., preventing illegal type conversions).

Exception Handling

Provides a robust exception-handling mechanism to catch and handle runtime errors.

Security Management

Uses Code Access Security (CAS) and Role-Based Security (RBS) to prevent unauthorized code execution.

Multithreading Support

Provides built-in support for multi-threading and parallel execution.

Interoperability

Allows calling native code (C/C++) and other .NET languages.

CLR Execution Process (Step by Step)

C# Code Compilation

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

Metadata & IL Storage

The IL code, along with metadata, is stored in an assembly (.exe or .dll file).

JIT Compilation

When executed, CLR's Just-In-Time (JIT) compiler converts IL to machine code for the target system.

Execution & Garbage Collection

The machine code is executed, and CLR manages memory, exceptions, and security.

Diagram: CLR Execution Flow

C# Code → Compiler → Intermediate Language (IL) → CLR (JIT Compiler) → Machine Code → Execution

Example: CLR in Action

using System;

class Program

{

static void Main()

{

Console.WriteLine("Hello, CLR!"); // CLR executes this

}

}

What Happens Internally?

The C# compiler (csc.exe) compiles the code into IL (Program.exe).

The CLR loads Program.exe and JIT compiles it into machine code.

The CPU executes the instructions and prints "Hello, CLR!".

JIT Compilation in CLR

CLR uses Just-In-Time (JIT) compilation, meaning the IL is compiled into machine code at runtime, not before.

Types of JIT in CLR

Normal JIT – Converts IL to machine code as needed and stores it in memory.

Econo JIT – Optimizes for memory, compiling code on the fly (used in low-memory scenarios).

Pre-JIT (Ahead of Time, AOT) – Compiles the entire IL to native code before execution.

CLR vs. JVM (Java Virtual Machine)

Feature CLR (C#) JVM (Java)

Language Supports multiple languages (C#, VB.NET, F#) Primarily for Java

Compilation IL + JIT Compilation Bytecode + JIT Compilation

Security Code Access Security (CAS) Security Manager

Memory Garbage Collection (GC) Garbage Collection (GC)

Conclusion

CLR is the heart of the .NET runtime, managing memory, security, and execution.

It compiles, executes, and optimizes .NET applications using JIT.

It ensures cross-language compatibility in the .NET ecosystem.

As of January 14, 2025, the latest release of .NET Core is **.NET 9.0.1**, with the latest SDK version being **9.0.102**.

[versionsof.net](https://versionsof.net/?utm_source=chatgpt.com)

For long-term support (LTS), **.NET 8.0** is the current LTS version, with the latest release being **8.0.12**.

[versionsof.net](https://versionsof.net/?utm_source=chatgpt.com)

You can download these versions from the official .NET download page.

[dotnet.microsoft.com](https://dotnet.microsoft.com/en-us/download/dotnet?utm_source=chatgpt.com)

For detailed information on the new features and improvements in .NET 9, you can refer to the official Microsoft documentation.

[learn.microsoft.com](https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-9/overview?utm_source=chatgpt.com)

As of January 2025, the latest stable version of C# is C# 13, released in conjunction with .NET 9.0.

learn.microsoft.com

Key Features in C# 13:

Params Collections: Enhanced flexibility in parameter handling, allowing for more concise and readable code.

New Lock Object: Introduction of a new lock type and semantics to improve synchronization mechanisms.

New Escape Sequence (\e): Addition of the \e escape sequence for better string handling.

Method Group Natural Type: Improvements in method group conversions, enhancing type inference and reducing the need for explicit delegates.

For a comprehensive overview of these features and more, you can refer to the official Microsoft documentation.

learn.microsoft.com

To start using C# 13, ensure you have the latest .NET 9 SDK and Visual Studio 2022 installed. You can download them from the official .NET downloads page.

dotnet.microsoft.com

For a detailed history of C# versions and their features, you can visit Microsoft's version history page.

learn.microsoft.com

In C#, memory is primarily divided into two types:

1️⃣ Stack Memory

Stores value types (e.g., int, double, char, bool, struct)

Stores method execution flow (local variables, method parameters)

Memory is allocated and deallocated automatically (Last-In, First-Out - LIFO)

Fast and efficient

Example of Stack Memory:

void Function()

{

int x = 10; // Stored in stack

int y = 20; // Stored in stack

}

Once the function completes execution, x and y are automatically removed from memory.

2️⃣ Heap Memory

Stores reference types (e.g., class, array, string, object, interface, delegate)

Dynamic memory allocation (data stays in memory until garbage collection)

Slower than stack but more flexible

Requires Garbage Collector (GC) to free unused memory

Example of Heap Memory:

class Person

{

public string Name;

}

void Function()

{

Person p = new Person(); // Object stored in heap, reference stored in stack

p.Name = "Alice"; // Name is stored in heap

}

Even after the method completes, the Person object remains in heap memory until the Garbage Collector (GC) removes it.

Stack vs Heap - Quick Comparison

Feature Stack Memory Heap Memory

Storage Type Value types, method calls Reference types, dynamic data

Allocation Static (fixed size) Dynamic (variable size)

Speed Fast (LIFO access) Slower (GC manages it)

Scope Limited to method execution Exists until garbage collection

Example int x = 5; (stored in stack) Person p = new Person(); (object in heap, reference in stack)

C# primarily runs on the .NET Common Language Runtime (CLR), which handles memory management, security, and exception handling. Based on this, code can be classified into Managed Code and Unmanaged Code.

1️⃣ Managed Code

Code that runs under the control of CLR

Memory allocation & deallocation handled automatically by Garbage Collector (GC)

Safe from memory leaks and pointer-related errors

Written in .NET-supported languages (C#, VB.NET, F#)

Example of Managed Code in C#:

class Program

{

static void Main()

{

string name = "Alice"; // Memory managed by CLR

Console.WriteLine(name); // Safe execution

}

}

✅ CLR manages memory, prevents crashes, and ensures security.

2️⃣ Unmanaged Code

Directly interacts with system memory & hardware

No CLR intervention → Requires manual memory management

Written in languages like C, C++

Risk of memory leaks, buffer overflow, and pointer errors

Example of Unmanaged Code in C#:

C# can use unmanaged code via P/Invoke (Platform Invocation Services) or unsafe mode.

Calling Unmanaged Code using P/Invoke

using System;

using System.Runtime.InteropServices;

class Program

{

[DllImport("user32.dll")] // Import unmanaged Windows API

public static extern int MessageBox(IntPtr hWnd, string text, string caption, int type);

static void Main()

{

MessageBox(IntPtr.Zero, "Hello from unmanaged code!", "P/Invoke Example", 0);

}

}

📌 This calls a Windows API function (MessageBox), which is unmanaged.

Using unsafe Code in C# (Unmanaged Memory Access)

unsafe class Program

{

static void Main()

{

int num = 10;

int\* ptr = &num; // Pointer to unmanaged memory

Console.WriteLine($"Value: {\*ptr} at Address: {(long)ptr}");

}

}

📌 unsafe keyword allows pointer manipulation (not managed by CLR).

🔴 Requires enabling "Allow unsafe code" in project settings.

🔥 Key Differences:

Feature Managed Code Unmanaged Code

Runs On CLR (.NET runtime) Directly on OS

Memory Management Automatic (Garbage Collector) Manual (malloc/free)

Language Support C#, VB.NET, F# C, C++

Performance Slightly slower (GC overhead) Faster (direct hardware access)

Safety Secure, prevents memory leaks Risk of crashes, buffer overflows

Interoperability Can call unmanaged code via P/Invoke Can’t directly use managed code

When to Use Unmanaged Code in C#?

Calling low-level OS APIs (e.g., Windows API, Linux syscalls)

Interfacing with legacy C/C++ libraries

High-performance applications (e.g., Game development with Unity)

Embedded systems & hardware interactions

Would you like help with writing a wrapper for unmanaged C++ code in C#? 🚀

ChatGPT can make mistakes. Check important info.

Global Assembly Cache (GAC) in .NET

📌 What is GAC?

The Global Assembly Cache (GAC) is a centralized repository in Windows where shared .NET assemblies (DLL files) are stored. It allows multiple .NET applications to reuse common libraries without keeping multiple copies.

1️⃣ Why Use GAC?

Reusability → Allows multiple applications to share the same assembly.

Version Control → Supports side-by-side execution of different assembly versions.

Security → Ensures only trusted assemblies are installed.

Performance → Reduces redundancy by storing only one copy globally.

2️⃣ Where is GAC Located?

📁 GAC is usually found at:

👉 C:\Windows\Microsoft.NET\assembly (For .NET Framework)

👉 C:\Windows\assembly (For older versions)

For .NET Core & .NET 5+, GAC is not used because it supports application-local DLLs via NuGet packages.

3️⃣ How to Install an Assembly in GAC?

✅ Using Gacutil.exe (for .NET Framework assemblies)

Run this in Developer Command Prompt:

gacutil -i MyLibrary.dll

✅ Using PowerShell (for Windows)

[System.Reflection.Assembly]::Load("MyLibrary")

✅ Using Windows Explorer (Drag & Drop)

Navigate to C:\Windows\assembly

Drag & drop your DLL file

4️⃣ How to List All Assemblies in GAC?

gacutil -l

or

📌 Open C:\Windows\assembly in File Explorer.

5️⃣ How to Remove an Assembly from GAC?

gacutil -u MyLibrary

6️⃣ How to Use GAC Assembly in C#?

Add a reference to the GAC assembly in your C# project:

using MyLibraryNamespace; // Namespace from GAC DLL

class Program

{

static void Main()

{

MyLibraryClass obj = new MyLibraryClass();

obj.Method();

}

}

7️⃣ Key Points

Feature Global Assembly Cache (GAC)

Purpose Stores shared .NET assemblies

Location C:\Windows\Microsoft.NET\assembly

Used By .NET Framework (Not in .NET Core/5+)

Installation gacutil -i MyLibrary.dll

Removal gacutil -u MyLibrary

Benefits Versioning, security, reusability

📌 GAC is useful for shared assemblies but is not used in .NET Core or .NET 5+ since they rely on NuGet package management.

Would you like an example of creating and installing a strong-named assembly in GAC? 🚀

throw vs throw ex in C#

Both throw and throw ex are used to handle exceptions, but they behave differently in preserving the stack trace.

1️⃣ throw (Recommended)

✅ Preserves the original stack trace

✅ Useful for rethrowing exceptions without losing details

Example:

try

{

int result = 10 / 0; // Throws DivideByZeroException

}

catch (Exception ex)

{

Console.WriteLine("Logging exception...");

throw; // Rethrows the original exception

}

🔹 The stack trace remains unchanged, pointing to the exact line where the exception occurred.

2️⃣ throw ex (Not Recommended)

❌ Resets the stack trace

❌ Loses original exception details

Example:

try

{

int result = 10 / 0;

}

catch (Exception ex)

{

Console.WriteLine("Logging exception...");

throw ex; // BAD practice: Resets stack trace

}

🔹 Here, the stack trace resets to the throw ex; line, making it harder to debug.

3️⃣ Key Differences

Feature throw throw ex

Preserves Stack Trace? ✅ Yes ❌ No

Best for Debugging? ✅ Yes ❌ No

Use Case Rethrowing the same exception Modifying exception details (not recommended)

Stack Trace Points To Original error location Catch block (hides actual error location)

4️⃣ When to Use?

✔ Always use throw; when rethrowing an exception.

✔ Use throw ex; only when modifying the exception before throwing.

Valid Case for throw ex (Adding Custom Message)

try

{

int result = 10 / 0;

}

catch (Exception ex)

{

throw new Exception("Custom message: Division error", ex); // Keeps original exception inside

}

📌 ex is wrapped inside the new exception, preserving details while adding context.

Would you like an example with custom exception classes? 🚀

SQL Server Management Studio (SSMS)

📌 What is SSMS?

SQL Server Management Studio (SSMS) is a graphical tool developed by Microsoft to manage, configure, and interact with SQL Server databases. It provides an integrated environment for writing SQL queries, managing databases, and performing administrative tasks.

1️⃣ Key Features of SSMS

✔ Database Management → Create, modify, delete databases

✔ Query Execution → Write and run SQL queries (SELECT, INSERT, UPDATE, etc.)

✔ Backup & Restore → Manage database backups and restore operations

✔ Security Management → Control user access and permissions

✔ Table & Index Management → Create and optimize tables and indexes

✔ Performance Monitoring → View execution plans, analyze query performance

✔ Job Scheduling → Automate tasks using SQL Server Agent

2️⃣ How to Download & Install SSMS?

📌 Latest Version: Download from Microsoft

Installation Steps:

Download SSMS from the official Microsoft link

Run the installer (SSMS-Setup.exe)

Click Install (default settings work fine)

Once installed, launch SSMS

3️⃣ How to Connect to a Database in SSMS?

Open SQL Server Management Studio

In the Connect to Server window:

Server Type: Database Engine

Server Name: localhost (or the actual SQL Server name)

Authentication: Choose Windows Authentication or SQL Server Authentication

Click Connect

4️⃣ Writing & Running Queries in SSMS

Click New Query

Write an SQL command, e.g.:

SELECT \* FROM Users;

Click Execute (or press F5)

5️⃣ Common SQL Commands in SSMS

Operation SQL Command

Create Database CREATE DATABASE MyDatabase;

Select Data SELECT \* FROM Users;

Insert Data INSERT INTO Users (Name, Email) VALUES ('John', 'john@email.com');

Update Data UPDATE Users SET Email = 'john123@email.com' WHERE Name = 'John';

Delete Data DELETE FROM Users WHERE Name = 'John';

Backup Database BACKUP DATABASE MyDatabase TO DISK = 'C:\backup.bak';

6️⃣ Who Uses SSMS?

👨‍💻 Developers → Write and optimize queries

🛠️ DBAs (Database Administrators) → Manage security, backups, and performance

📊 Data Analysts → Query and analyze data

7️⃣ SSMS vs. Azure Data Studio

Feature SSMS Azure Data Studio

For SQL Server? ✅ Yes ✅ Yes

GUI-based? ✅ Yes ✅ Yes

Best for DB Admin? ✅ Yes ❌ No

Lightweight? ❌ No ✅ Yes

Cross-platform? ❌ No (Windows only) ✅ Yes (Windows, Linux, Mac)

📌 SSMS is best for full database management, while Azure Data Studio is lightweight and better for cloud-based analytics.

5️⃣ First() vs FirstOrDefault() in C#

Both First() and FirstOrDefault() are used to retrieve the first element from a collection. The difference is in how they handle empty collections.

✅ First()

Returns the first element of a collection.

Throws an exception if the collection is empty.

🔹 Example:

List<int> numbers = new List<int> { 10, 20, 30 };

int firstNumber = numbers.First(); // ✅ Returns 10

List<int> emptyList = new List<int>();

int firstEmpty = emptyList.First(); // ❌ Throws InvalidOperationException

✅ FirstOrDefault()

Returns the first element of a collection.

Returns default value (null or 0) if the collection is empty.

Does not throw an exception.

🔹 Example:

List<int> emptyList = new List<int>();

int firstOrDefault = emptyList.FirstOrDefault(); // ✅ Returns 0 (default for int)

✅ When to use?

✔ Use First() when you're sure the collection has at least one element.

✔ Use FirstOrDefault() to avoid exceptions when the collection might be empty.

6️⃣ Single() vs SingleOrDefault() in C#

Both Single() and SingleOrDefault() are used when you expect only one element in the collection.

✅ Single()

Returns the only element from a collection.

Throws an exception if:

The collection is empty.

The collection contains more than one element.

🔹 Example:

List<int> numbers = new List<int> { 10 };

int singleNumber = numbers.Single(); // ✅ Returns 10

List<int> emptyList = new List<int>();

int singleEmpty = emptyList.Single(); // ❌ Throws InvalidOperationException

List<int> multipleNumbers = new List<int> { 10, 20 };

int singleMultiple = multipleNumbers.Single(); // ❌ Throws InvalidOperationException

✅ SingleOrDefault()

Returns the only element if exactly one exists.

Returns default value (null or 0) if the collection is empty.

Throws an exception if the collection contains more than one element.

🔹 Example:

List<int> numbers = new List<int> { 10 };

int singleOrDefault = numbers.SingleOrDefault(); // ✅ Returns 10

List<int> emptyList = new List<int>();

int singleEmpty = emptyList.SingleOrDefault(); // ✅ Returns 0 (default for int)

List<int> multipleNumbers = new List<int> { 10, 20 };

int singleMultiple = multipleNumbers.SingleOrDefault(); // ❌ Throws InvalidOperationException

✅ When to use?

✔ Use Single() when you expect exactly one element and want to enforce it.

✔ Use SingleOrDefault() when zero or one element is expected, but not more.

🎯 Key Differences Summary

Method Returns Throws Exception? (Empty) Throws Exception? (Multiple Items)

First() First item ✅ Yes ❌ No

FirstOrDefault() First item or default(T) ❌ No ❌ No

Single() Single item ✅ Yes ✅ Yes

SingleOrDefault() Single item or default(T) ❌ No ✅ Yes

Would you like a real-world example using LINQ and Entity Framework? 🚀

7️⃣ Access Modifiers in C#

Access modifiers in C# control the visibility and accessibility of classes, methods, and variables.

✅ Types of Access Modifiers

Modifier Access Level Usage

public Accessible from anywhere Used for global access

private Accessible only inside the class Used to restrict access

protected Accessible inside the class and its derived (child) classes Used for inheritance

internal Accessible within the same assembly (project) Used for modularization

protected internal Accessible within the same assembly and derived classes Combines protected + internal

private protected Accessible only inside the class and its derived classes within the same assembly More restrictive than protected

🔹 1️⃣ public Modifier (No Restriction)

The member can be accessed from anywhere (inside and outside the class).

🔹 Example:

class Person

{

public string Name = "John"; // Public variable

}

class Program

{

static void Main()

{

Person p = new Person();

Console.WriteLine(p.Name); // ✅ Accessible

}

}

🔹 2️⃣ private Modifier (Restricted to the Same Class)

The member is only accessible inside the same class.

🔹 Example:

class Person

{

private string SSN = "123-45-6789"; // Private variable

void DisplaySSN()

{

Console.WriteLine(SSN); // ✅ Accessible inside the class

}

}

class Program

{

static void Main()

{

Person p = new Person();

// Console.WriteLine(p.SSN); ❌ Not accessible (Compilation Error)

}

}

🔹 3️⃣ protected Modifier (Accessible in Derived Classes)

The member is accessible only inside the same class and child classes (inheritance).

🔹 Example:

class Parent

{

protected int Age = 50; // Protected variable

}

class Child : Parent

{

void Display()

{

Console.WriteLine(Age); // ✅ Accessible in derived class

}

}

class Program

{

static void Main()

{

Child c = new Child();

// Console.WriteLine(c.Age); ❌ Not accessible from outside

}

}

🔹 4️⃣ internal Modifier (Same Assembly Access)

The member is accessible within the same project/assembly but not outside.

🔹 Example:

internal class InternalClass

{

public void Show()

{

Console.WriteLine("Hello from Internal Class!");

}

}

class Program

{

static void Main()

{

InternalClass obj = new InternalClass();

obj.Show(); // ✅ Accessible in the same project

}

}

If another project tries to access InternalClass, it will cause an error.

🔹 5️⃣ protected internal Modifier (Protected + Internal)

Accessible in the same assembly OR in derived classes (even if in different assemblies).

🔹 Example:

public class BaseClass

{

protected internal void Display()

{

Console.WriteLine("Protected Internal Method");

}

}

class Derived : BaseClass

{

void Show()

{

Display(); // ✅ Accessible in derived class

}

}

class Program

{

static void Main()

{

BaseClass obj = new BaseClass();

obj.Display(); // ✅ Accessible within the same assembly

}

}

🔹 6️⃣ private protected Modifier (Most Restrictive)

Accessible only within the same class and derived classes in the same assembly.

More restrictive than protected internal.

🔹 Example:

public class Base

{

private protected void Show()

{

Console.WriteLine("Private Protected Method");

}

}

public class Derived : Base

{

void Display()

{

Show(); // ✅ Accessible inside the derived class in the same assembly

}

}

class Program

{

static void Main()

{

// Base obj = new Base();

// obj.Show(); ❌ Not accessible from outside

}

}

🎯 Summary Table

Modifier Inside Class Derived Class Same Assembly Outside Assembly

public ✅ ✅ ✅ ✅

private ✅ ❌ ❌ ❌

protected ✅ ✅ ❌ ❌

internal ✅ ✅ ✅ ❌

protected internal ✅ ✅ ✅ ✅ (only in derived)

private protected ✅ ✅ ❌ ❌

🚀 When to Use Which Modifier?

✔ Use public → When you want global access.

✔ Use private → When restricting access inside the same class.

✔ Use protected → When allowing access only in child classes.

✔ Use internal → When keeping access within the same project.

✔ Use protected internal → When allowing access in the same project and child classes outside.

✔ Use private protected → When you want the strictest access control.

Would you like an example project demonstrating access modifiers? 🚀

8️⃣ Difference Between Overloading and Overriding in C#

🔹 Method Overloading vs Method Overriding

Feature Method Overloading Method Overriding

Definition Defining multiple methods with the same name but different parameters in the same class. Redefining a method from a parent class in a derived class with the same signature.

Where It Happens Same class Different classes (inheritance required)

Parameter Differences Must have different number, type, or order of parameters. Parameters must remain the same.

Return Type Can be different. Must be the same or covariant (compatible with base type).

Access Modifier Can have any access modifier (public, private, etc.). Cannot override private methods.

Keyword Used No keyword needed. Uses override in derived class and virtual in base class.

Performance Impact Resolved at compile-time (early binding). Resolved at runtime (late binding).

Polymorphism Type Compile-time (static) polymorphism. Runtime (dynamic) polymorphism.

🔹 Example of Method Overloading (Same Class)

class MathOperations

{

public int Add(int a, int b) // First method

{

return a + b;

}

public double Add(double a, double b) // Second method (different type)

{

return a + b;

}

public int Add(int a, int b, int c) // Third method (different number of parameters)

{

return a + b + c;

}

}

class Program

{

static void Main()

{

MathOperations obj = new MathOperations();

Console.WriteLine(obj.Add(2, 3)); // Calls first method

Console.WriteLine(obj.Add(2.5, 3.5)); // Calls second method

Console.WriteLine(obj.Add(2, 3, 4)); // Calls third method

}

}

✅ Same method name (Add) but different parameter lists.

🔹 Example of Method Overriding (Inheritance)

class Parent

{

public virtual void Show() // Base method marked as 'virtual'

{

Console.WriteLine("Hello from Parent class!");

}

}

class Child : Parent

{

public override void Show() // Overriding the base method

{

Console.WriteLine("Hello from Child class!");

}

}

class Program

{

static void Main()

{

Parent obj1 = new Parent();

obj1.Show(); // Calls Parent class method

Child obj2 = new Child();

obj2.Show(); // Calls Child class method (overridden version)

Parent obj3 = new Child();

obj3.Show(); // Calls Child class method (runtime polymorphism)

}

}

✅ The Child class redefines (overrides) the Show() method from the Parent class.

🔹 Key Takeaways

1️⃣ Method Overloading → Same class, different parameter list, compile-time binding.

2️⃣ Method Overriding → Inheritance required, same method signature, runtime binding.

Call by Value vs Call by Reference in C#

Feature Call by Value Call by Reference

Definition A copy of the value is passed to the method. A reference (memory address) of the variable is passed.

Effect on Original Value Changes inside the method do not affect the original variable. Changes inside the method affect the original variable.

Data Type Used Used for value types (int, double, struct, etc.). Used for reference types (class, object, array, etc.).

Keyword Used No special keyword required. Uses ref or out keyword.

Memory Used Stored in the stack. Stored in the heap (for reference types).

Performance Faster as it works with copies. Slightly slower due to working with references.

🔹 Example of Call by Value

using System;

class Program

{

static void ModifyValue(int number)

{

number = 100; // Changing local copy

}

static void Main()

{

int num = 50;

Console.WriteLine("Before: " + num); // Output: 50

ModifyValue(num);

Console.WriteLine("After: " + num); // Output: 50 (unchanged)

}

}

✅ The original value remains unchanged because a copy is passed.

🔹 Example of Call by Reference (ref Keyword)

using System;

class Program

{

static void ModifyReference(ref int number)

{

number = 100; // Changes the original variable

}

static void Main()

{

int num = 50;

Console.WriteLine("Before: " + num); // Output: 50

ModifyReference(ref num);

Console.WriteLine("After: " + num); // Output: 100 (changed)

}

}

✅ The ref keyword passes the actual variable, so modifications affect the original value.

🔹 Example of out Keyword (Similar to ref, but No Initial Value Required)

using System;

class Program

{

static void AssignValue(out int number)

{

number = 200; // Must assign a value

}

static void Main()

{

int num; // No need to initialize

AssignValue(out num);

Console.WriteLine("Value: " + num); // Output: 200

}

}

✅ The out keyword ensures the method assigns a value before returning.

🔹 Key Takeaways

1️⃣ Call by Value → Copies the value, no effect on the original variable.

2️⃣ Call by Reference (ref) → Passes a reference, changes affect the original variable.

3️⃣ out Keyword → Similar to ref, but must be assigned inside the method.

10.types of constructor and its uses

Types of Constructors in C# and Their Uses

A constructor is a special method in a class that is automatically called when an object is created. In C#, constructors are mainly used for initializing objects.

🔹 Types of Constructors in C#

Constructor Type Description Example

1. Default Constructor Initializes an object with default values. ✅ Used when no explicit values are required.

2. Parameterized Constructor Accepts parameters to assign specific values at object creation. ✅ Used to initialize objects with custom values.

3. Copy Constructor Creates a new object by copying values from another object. ✅ Used for cloning objects.

4. Static Constructor Called once for a class, before any object is created. ✅ Used for initializing static data members.

5. Private Constructor Prevents object instantiation outside the class. ✅ Used in Singleton Design Patterns.

6. Constructor Overloading Multiple constructors with different parameters. ✅ Provides multiple ways to initialize objects.

🔹 1. Default Constructor (No Parameters)

If no constructor is defined, C# provides a default one.

class Person

{

public string Name;

// Default constructor

public Person()

{

Name = "Unknown";

}

public void Display()

{

Console.WriteLine("Name: " + Name);

}

}

class Program

{

static void Main()

{

Person p = new Person(); // Calls the default constructor

p.Display(); // Output: Name: Unknown

}

}

🔹 2. Parameterized Constructor

Accepts parameters to initialize values at object creation.

class Person

{

public string Name;

// Parameterized constructor

public Person(string name)

{

Name = name;

}

public void Display()

{

Console.WriteLine("Name: " + Name);

}

}

class Program

{

static void Main()

{

Person p = new Person("John"); // Passing "John" as parameter

p.Display(); // Output: Name: John

}

}

🔹 3. Copy Constructor

Creates a new object with the same values as an existing object.

class Person

{

public string Name;

// Parameterized constructor

public Person(string name)

{

Name = name;

}

// Copy constructor

public Person(Person obj)

{

Name = obj.Name;

}

public void Display()

{

Console.WriteLine("Name: " + Name);

}

}

class Program

{

static void Main()

{

Person p1 = new Person("Alice");

Person p2 = new Person(p1); // Copying object p1 into p2

p1.Display(); // Output: Name: Alice

p2.Display(); // Output: Name: Alice

}

}

🔹 4. Static Constructor

Called only once before any object is created.

Used for static data initialization.

class Example

{

static int Count;

// Static constructor

static Example()

{

Count = 10;

Console.WriteLine("Static Constructor Called");

}

public static void ShowCount()

{

Console.WriteLine("Count: " + Count);

}

}

class Program

{

static void Main()

{

Example.ShowCount(); // Output: Static Constructor Called, Count: 10

}

}

✅ The static constructor runs once before any method is executed.

🔹 5. Private Constructor

Prevents object creation outside the class.

Used in Singleton Design Patterns.

class Singleton

{

private static Singleton instance;

// Private constructor

private Singleton()

{

Console.WriteLine("Singleton Instance Created");

}

public static Singleton GetInstance()

{

if (instance == null)

{

instance = new Singleton();

}

return instance;

}

}

class Program

{

static void Main()

{

Singleton obj1 = Singleton.GetInstance();

Singleton obj2 = Singleton.GetInstance();

// Output: Singleton Instance Created (only once)

}

}

✅ Ensures only one instance of the class exists.

🔹 6. Constructor Overloading

Multiple constructors with different parameters.

class Person

{

public string Name;

public int Age;

// Default constructor

public Person()

{

Name = "Unknown";

Age = 0;

}

// Parameterized constructor

public Person(string name)

{

Name = name;

}

// Another parameterized constructor

public Person(string name, int age)

{

Name = name;

Age = age;

}

public void Display()

{

Console.WriteLine($"Name: {Name}, Age: {Age}");

}

}

class Program

{

static void Main()

{

Person p1 = new Person();

Person p2 = new Person("Alice");

Person p3 = new Person("Bob", 25);

p1.Display(); // Output: Name: Unknown, Age: 0

p2.Display(); // Output: Name: Alice, Age: 0

p3.Display(); // Output: Name: Bob, Age: 25

}

}

✅ Allows multiple ways to create an object.

🔹 Key Takeaways

1️⃣ Default Constructor → No parameters, assigns default values.

2️⃣ Parameterized Constructor → Accepts values to initialize fields.

3️⃣ Copy Constructor → Creates a new object by copying another object.

4️⃣ Static Constructor → Runs only once, initializes static members.

5️⃣ Private Constructor → Prevents external instantiation (used in Singleton Pattern).

6️⃣ Constructor Overloading → Multiple constructors with different parameters.

11. what are collections and types of collection

🔹 Collections in C# and Their Types

🔹 What are Collections?

A collection in C# is a group of related objects stored together in memory. Collections help manage and manipulate groups of objects efficiently.

🔹 Types of Collections in C#

C# provides different types of collections categorized into two major types:

Type Description Example

1. Non-Generic Collections Uses object type, leading to boxing/unboxing overhead. ArrayList, Hashtable

2. Generic Collections Strongly typed collections that improve performance and type safety. List<T>, Dictionary<K,V>

3. Concurrent Collections Thread-safe collections for multi-threaded applications. ConcurrentBag<T>, ConcurrentQueue<T>

🔹 1. Non-Generic Collections (From System.Collections)

🔹 These collections store elements as object type, leading to typecasting issues.

✅ ArrayList (Resizable Array)

Stores different data types in the same list.

using System;

using System.Collections;

class Program

{

static void Main()

{

ArrayList list = new ArrayList();

list.Add(10);

list.Add("Hello");

list.Add(20.5);

foreach (var item in list)

{

Console.WriteLine(item);

}

}

}

🚀 Drawback: Requires type casting when retrieving elements.

✅ Hashtable (Key-Value Pair)

Stores key-value pairs.

using System;

using System.Collections;

class Program

{

static void Main()

{

Hashtable ht = new Hashtable();

ht.Add(1, "Apple");

ht.Add(2, "Banana");

Console.WriteLine(ht[1]); // Output: Apple

}

}

🚀 Drawback: Uses object type, leading to boxing/unboxing overhead.

🔹 2. Generic Collections (From System.Collections.Generic)

🔹 Generic collections are strongly typed, which improves performance and avoids typecasting.

✅ List<T> (Dynamic Array)

Fast, flexible, and type-safe.

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

List<int> numbers = new List<int> {1, 2, 3, 4};

numbers.Add(5);

foreach (var num in numbers)

{

Console.WriteLine(num);

}

}

}

🚀 Advantage: No typecasting needed.

✅ Dictionary<TKey, TValue> (Key-Value Collection)

Stores unique keys with associated values.

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

Dictionary<int, string> students = new Dictionary<int, string>();

students.Add(1, "Alice");

students.Add(2, "Bob");

Console.WriteLine(students[1]); // Output: Alice

}

}

🚀 Advantage: Fast lookups with unique keys.

✅ Queue<T> (FIFO - First In, First Out)

Elements are processed in the order they are added.

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

Queue<string> queue = new Queue<string>();

queue.Enqueue("A");

queue.Enqueue("B");

Console.WriteLine(queue.Dequeue()); // Output: A

}

}

🚀 Use Case: Task Scheduling (e.g., Print Jobs).

✅ Stack<T> (LIFO - Last In, First Out)

Last element added is the first to be removed.

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

Stack<int> stack = new Stack<int>();

stack.Push(1);

stack.Push(2);

Console.WriteLine(stack.Pop()); // Output: 2

}

}

🚀 Use Case: Undo functionality (e.g., Back button in browsers).

🔹 3. Concurrent Collections (From System.Collections.Concurrent)

🔹 Thread-safe collections for multi-threaded applications.

✅ ConcurrentBag<T> (Unordered Collection)

Stores items without any specific order, optimized for concurrency.

using System;

using System.Collections.Concurrent;

class Program

{

static void Main()

{

ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(1);

bag.Add(2);

foreach (var item in bag)

{

Console.WriteLine(item);

}

}

}

🚀 Use Case: Log buffering in multi-threaded applications.

🔹 Key Differences Between Collections

Collection Type Ordered? Allows Duplicates? Key-Value Pair?

ArrayList ✅ Yes ✅ Yes ❌ No

List<T> ✅ Yes ✅ Yes ❌ No

Hashtable ❌ No ✅ Yes ✅ Yes

Dictionary<K,V> ❌ No ✅ Yes ✅ Yes

Queue<T> ✅ FIFO ✅ Yes ❌ No

Stack<T> ✅ LIFO ✅ Yes ❌ No

🔹 When to Use Which Collection?

List<T> → If you need a dynamic array with fast lookup.

Dictionary<K, V> → If you need key-value mapping.

Queue<T> → If first-in-first-out order is required.

Stack<T> → If last-in-first-out order is needed.

ConcurrentBag<T> → If thread safety is required.

🔹 Summary

✅ Non-Generic Collections (ArrayList, Hashtable) → Allow mixed data types but require boxing/unboxing.

✅ Generic Collections (List<T>, Dictionary<K, V>) → Strongly typed, faster, and type-safe.

✅ Concurrent Collections (ConcurrentBag<T>, ConcurrentQueue<T>) → Thread-safe for multi-threaded apps.

Advantages of Object-Oriented Programming (OOP) in C#

Object-Oriented Programming (OOP) is a programming paradigm that revolves around objects and classes. It enhances code organization, reusability, and scalability.

🔹 Key Advantages of OOP

1️⃣ Code Reusability (Encapsulation & Inheritance)

OOP allows code reuse through inheritance, reducing redundancy.

Encapsulation ensures data hiding, improving security.

✅ Example:

class Animal

{

public void Eat() { Console.WriteLine("Eating..."); }

}

class Dog : Animal

{

public void Bark() { Console.WriteLine("Barking..."); }

}

class Program

{

static void Main()

{

Dog myDog = new Dog();

myDog.Eat(); // Reusing Eat() method from Animal class

myDog.Bark();

}

}

🚀 Benefit: Reduces duplication, making code maintainable.

2️⃣ Modularity (Encapsulation)

Encapsulation ensures that data is private and can only be accessed via methods.

Data is protected from unintended modifications.

✅ Example:

class BankAccount

{

private double balance = 1000;

public double GetBalance() => balance; // Read-Only Access

}

🚀 Benefit: Improved security by preventing direct data modification.

3️⃣ Flexibility & Scalability (Polymorphism)

Polymorphism allows a method to have multiple implementations, making the system flexible.

✅ Example:

class Shape

{

public virtual void Draw() { Console.WriteLine("Drawing Shape..."); }

}

class Circle : Shape

{

public override void Draw() { Console.WriteLine("Drawing Circle..."); }

}

class Program

{

static void Main()

{

Shape shape = new Circle();

shape.Draw(); // Calls Circle's Draw() method

}

}

🚀 Benefit: Code adaptability without modifying existing code.

4️⃣ Improved Maintenance & Debugging

Divides complex programs into smaller, manageable classes.

Easier to fix bugs and enhance features.

✅ Example:

Instead of one large function handling everything, OOP breaks functionality into multiple classes.

5️⃣ Code Abstraction (Hides Complexity)

Users don’t need to know the internal workings of a class.

✅ Example:

abstract class Vehicle

{

public abstract void Start();

}

class Car : Vehicle

{

public override void Start() { Console.WriteLine("Car Starting..."); }

}

🚀 Benefit: Hides unnecessary implementation details.

6️⃣ Security (Encapsulation & Access Modifiers)

Encapsulation hides sensitive data from unauthorized access.

✅ Example:

class User

{

private string password = "secure123";

public bool Authenticate(string inputPassword) => inputPassword == password;

}

🚀 Benefit: Prevents unauthorized access to critical data.

7️⃣ Parallel Development (Team Collaboration)

Multiple developers can work on different objects simultaneously without conflict.

✅ Example:

Frontend developers can work on the User Interface, while backend developers handle Business Logic in separate classes.

🔹 Summary of OOP Advantages

Feature Benefit

Code Reusability Reduces redundancy through inheritance

Encapsulation Protects data from unauthorized modification

Polymorphism Allows flexibility & scalability

Abstraction Hides complexity & simplifies usage

Security Prevents unauthorized access

Maintainability Easier debugging & modification

Parallel Development Teams can work on different modules independently

Exception handling in c#

Exception Handling in C#

Exception handling in C# ensures that programs gracefully handle runtime errors without crashing.

🔹 What is an Exception?

An exception is an unexpected event at runtime that disrupts normal execution.

🔸 Examples: Division by zero, accessing a null object, invalid array index.

✅ Example without Exception Handling (Program Crashes)

int a = 10, b = 0;

int result = a / b; // ❌ This will cause "DivideByZeroException"

Console.WriteLine(result);

💥 Error: System.DivideByZeroException: Attempted to divide by zero.

🔹 Exception Handling Mechanisms

C# provides try-catch-finally-throw blocks to handle exceptions.

1️⃣ Try-Catch Block (Basic Handling)

✅ Example: Handling DivideByZeroException

try

{

int a = 10, b = 0;

int result = a / b; // ❌ Exception occurs here

Console.WriteLine(result);

}

catch (DivideByZeroException ex)

{

Console.WriteLine($"Error: {ex.Message}"); // 🛑 Exception Handled

}

🔹 Output: Error: Attempted to divide by zero.

2️⃣ Multiple Catch Blocks

Each catch handles a specific exception type.

✅ Example: Handling Multiple Exceptions

try

{

int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers[5]); // ❌ IndexOutOfRangeException

}

catch (DivideByZeroException ex)

{

Console.WriteLine("Cannot divide by zero.");

}

catch (IndexOutOfRangeException ex)

{

Console.WriteLine("Index out of bounds.");

}

catch (Exception ex) // Generic exception (should be last)

{

Console.WriteLine($"General Error: {ex.Message}");

}

🔹 Output: Index out of bounds.

3️⃣ Finally Block (Always Executes)

The finally block always executes, even if an exception occurs.

✅ Example: Ensuring Resource Cleanup

try

{

Console.WriteLine("Opening file...");

throw new Exception("File not found."); // Simulated error

}

catch (Exception ex)

{

Console.WriteLine($"Error: {ex.Message}");

}

finally

{

Console.WriteLine("Closing file..."); // ✅ Always executes

}

🔹 Output:

Opening file...

Error: File not found.

Closing file...

4️⃣ Throwing Custom Exceptions

Use throw to manually trigger exceptions.

✅ Example: Throwing a Custom Exception

static void ValidateAge(int age)

{

if (age < 18)

throw new ArgumentException("Age must be 18 or above."); // ❌ Custom exception

Console.WriteLine("Valid age.");

}

try

{

ValidateAge(15);

}

catch (Exception ex)

{

Console.WriteLine($"Validation Error: {ex.Message}");

}

🔹 Output: Validation Error: Age must be 18 or above.

5️⃣ Custom Exception Class

You can create custom exceptions for specific errors.

✅ Example: Creating a Custom Exception

class InvalidAgeException : Exception

{

public InvalidAgeException(string message) : base(message) { }

}

class Program

{

static void CheckAge(int age)

{

if (age < 18)

throw new InvalidAgeException("Age must be 18 or older.");

Console.WriteLine("Valid Age");

}

static void Main()

{

try

{

CheckAge(16);

}

catch (InvalidAgeException ex)

{

Console.WriteLine($"Custom Exception: {ex.Message}");

}

}

}

🔹 Output: Custom Exception: Age must be 18 or older.

🔹 Difference: throw vs throw ex

Feature throw throw ex

Stack Trace Maintains original error location Resets stack trace

Usage Used for rethrowing exceptions Used for custom exception handling

✅ Example of Difference

try

{

throw new Exception("Something went wrong.");

}

catch (Exception ex)

{

throw; // ✅ Preserves original error location

// throw ex; // ❌ Resets stack trace (not recommended)

}

🔹 Best Practices for Exception Handling

✅ Do’s

✔️ Use specific exceptions (e.g., FileNotFoundException, DivideByZeroException).

✔️ Log exceptions instead of printing them (use log4net, Serilog).

✔️ Use finally for clean-up operations (e.g., closing database connections).

✔️ Use throw (not throw ex) to preserve stack trace.

❌ Don’ts

🚫 Don’t use empty catch blocks (hides errors).

🚫 Avoid catching generic Exception unless necessary.

🚫 Don’t use exceptions for control flow (use if-else instead).

🔹 Summary

Concept Description

try-catch Catches exceptions and prevents crashes

finally Always executes, used for cleanup

throw Raises an exception manually

Custom Exception Defines user-defined errors

throw vs throw ex throw preserves stack trace; throw ex resets it

🔹 out and ref Parameters in C#

In C#, out and ref parameters allow methods to return multiple values and modify variables outside the method.

1️⃣ ref (Call by Reference)

Used to pass a variable by reference (modifications reflect in the caller).

The variable must be initialized before passing.

✅ Example: Using ref

using System;

class Program

{

static void DoubleValue(ref int num)

{

num \*= 2; // Modify original variable

}

static void Main()

{

int value = 10;

Console.WriteLine($"Before: {value}");

DoubleValue(ref value); // Pass by reference

Console.WriteLine($"After: {value}"); // ✅ Modified value: 20

}

}

🔹 Output:

Before: 10

After: 20

2️⃣ out (Output Parameter)

Used when a method returns multiple values.

The variable does not need to be initialized before passing.

The method must assign a value before returning.

✅ Example: Using out

using System;

class Program

{

static void GetValues(out int a, out int b)

{

a = 5; // Must assign values

b = 10;

}

static void Main()

{

int x, y; // No need to initialize

GetValues(out x, out y); // Get multiple values

Console.WriteLine($"x: {x}, y: {y}"); // ✅ Output: x: 5, y: 10

}

}

🔹 Output:

x: 5, y: 10

3️⃣ Difference Between out and ref

Feature ref out

Initialization Before Passing ✅ Required ❌ Not Required

Method Must Assign Value ❌ No ✅ Yes

Purpose Modify existing value Return multiple values

✅ Example Comparing ref and out

class Program

{

static void UseRef(ref int num) { num += 10; }

static void UseOut(out int num) { num = 20; } // Must assign

static void Main()

{

int a = 5; // ✅ Required for ref

UseRef(ref a);

Console.WriteLine($"Ref: {a}"); // Output: Ref: 15

int b; // ❌ No need to initialize for out

UseOut(out b);

Console.WriteLine($"Out: {b}"); // Output: Out: 20

}

}

4️⃣ When to Use ref and out

🔹 Use ref when modifying an existing variable.

🔹 Use out when returning multiple values from a method.

Would you like examples for real-world scenarios (e.g., calculations, database methods)? 🚀

Both break and continue are loop control statements in C#. They are used to control the flow of loops like for, while, and do-while.

1️⃣ break Statement

Used to exit a loop immediately.

Stops further iterations.

Mostly used in switch statements and loops.

✅ Example: Using break in a Loop

using System;

class Program

{

static void Main()

{

for (int i = 1; i <= 5; i++)

{

if (i == 3)

{

Console.WriteLine("Breaking loop at i = " + i);

break; // Exits the loop when i == 3

}

Console.WriteLine("i: " + i);

}

}

}

🔹 Output:

i: 1

i: 2

Breaking loop at i = 3

✅ Use Case: When you need to exit a loop based on a condition (e.g., stop searching after finding an item).

2️⃣ continue Statement

Skips the current iteration and moves to the next iteration.

Does not exit the loop.

✅ Example: Using continue

using System;

class Program

{

static void Main()

{

for (int i = 1; i <= 5; i++)

{

if (i == 3)

{

Console.WriteLine("Skipping i = " + i);

continue; // Skips current iteration when i == 3

}

Console.WriteLine("i: " + i);

}

}

}

🔹 Output:

i: 1

i: 2

Skipping i = 3

i: 4

i: 5

✅ Use Case: When you want to skip a specific iteration but continue looping.

3️⃣ Difference Between break and continue

Feature break continue

Effect Exits the loop completely Skips current iteration and moves to next

Used in Loops (for, while, do-while), switch Loops (for, while, do-while)

Example if (condition) break; if (condition) continue;

✅ Example Comparing break and continue

using System;

class Program

{

static void Main()

{

for (int i = 1; i <= 5; i++)

{

if (i == 3)

{

Console.WriteLine("Continue skips i = " + i);

continue; // Skips 3

}

if (i == 4)

{

Console.WriteLine("Break stops at i = " + i);

break; // Exits loop at 4

}

Console.WriteLine("i: " + i);

}

}

}

🔹 Output:

i: 1

i: 2

Continue skips i = 3

Break stops at i = 4

Difference Between Abstract Class and Interface in C#

Both abstract classes and interfaces are used to achieve abstraction in C#. However, they have key differences in how they work and when to use them.

1️⃣ Abstract Class

Can have both abstract (without implementation) and non-abstract methods (with implementation).

Can have constructors.

Can have fields (variables).

Supports access modifiers (public, protected, etc.).

Can provide default behavior (methods with implementations).

A class can inherit only one abstract class (single inheritance).

✅ Example: Abstract Class

using System;

abstract class Animal // Abstract class

{

public string Name { get; set; }

public void Display() // Concrete method (with implementation)

{

Console.WriteLine("This is an animal.");

}

public abstract void MakeSound(); // Abstract method (without implementation)

}

class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Dog barks!");

}

}

class Program

{

static void Main()

{

Dog dog = new Dog();

dog.Display(); // Calls method from abstract class

dog.MakeSound(); // Calls overridden method from Dog class

}

}

🔹 Output:

This is an animal.

Dog barks!

2️⃣ Interface

Cannot have fields (variables) or constructors.

All methods are abstract by default (no implementation).

All members are public by default (no access modifiers allowed).

Supports multiple inheritance (a class can implement multiple interfaces).

Used for defining a contract that multiple classes must follow.

✅ Example: Interface

using System;

interface IAnimal // Interface

{

void MakeSound(); // Abstract method (no implementation)

}

class Dog : IAnimal

{

public void MakeSound()

{

Console.WriteLine("Dog barks!");

}

}

class Program

{

static void Main()

{

Dog dog = new Dog();

dog.MakeSound(); // Calls method from interface

}

}

🔹 Output:

Dog barks!

3️⃣ Key Differences Between Abstract Class and Interface

Feature Abstract Class Interface

Methods Can have both abstract (without body) and concrete (with body) methods Only abstract methods (except default interface methods in C# 8+)

Fields Can have fields (variables) ❌ Cannot have fields

Constructors ✅ Can have constructors ❌ No constructors

Access Modifiers Supports public, private, protected, etc. All members are public by default

Multiple Inheritance ❌ Only single inheritance ✅ Supports multiple inheritance

Default Implementation ✅ Can provide method implementations ❌ No implementation (except in C# 8+)

4️⃣ When to Use What?

✅ Use Abstract Class When:

You need to share code among multiple related classes.

You want to partially implement methods (some with and some without implementation).

You need constructors, fields, or access modifiers.

✅ Use Interface When:

You need to enforce a contract (a set of methods) for multiple unrelated classes.

You require multiple inheritance.

You want to ensure loose coupling between classes.

17.how to prevent class from being inherited

How to Prevent a Class from Being Inherited in C#?

In C#, you can prevent a class from being inherited by marking it as sealed.

1️⃣ Using sealed Keyword

A sealed class cannot be inherited by other classes.

Useful when you want to restrict inheritance and ensure that no other class can derive from it.

✅ Example: Sealed Class

sealed class MyClass

{

public void Display()

{

Console.WriteLine("This is a sealed class.");

}

}

// ❌ This will cause a compilation error

// class DerivedClass : MyClass { }

class Program

{

static void Main()

{

MyClass obj = new MyClass();

obj.Display();

}

}

🔹 Output:

This is a sealed class.

🚨 If you try to inherit from MyClass, you will get a compilation error:

'DerivedClass': cannot derive from sealed type 'MyClass'

2️⃣ Preventing Method Overriding Using sealed with override

If you don’t want to seal the entire class but only want to prevent overriding a method, you can use sealed with an override method.

✅ Example: Sealing a Method

class BaseClass

{

public virtual void Show()

{

Console.WriteLine("Base class method.");

}

}

class DerivedClass : BaseClass

{

public sealed override void Show()

{

Console.WriteLine("Sealed method in derived class.");

}

}

// ❌ Further subclassing and overriding Show() is not allowed

// class SubDerivedClass : DerivedClass

// {

// public override void Show() { } // ❌ Compilation error

// }

class Program

{

static void Main()

{

DerivedClass obj = new DerivedClass();

obj.Show();

}

}

🔹 Output:

Sealed method in derived class.

🚨 If you try to override Show() in SubDerivedClass, you will get a compilation error:

'SubDerivedClass.Show()': cannot override inherited member because it is sealed.

3️⃣ When to Use sealed?

✅ Use sealed class when:

You don’t want other classes to inherit from your class (e.g., System.String is a sealed class).

Your class is designed as a final implementation (not meant for extension).

✅ Use sealed method when:

You want to allow inheritance but prevent method overriding.

🔹 Polymorphism in C# and Its Types

What is Polymorphism?

Polymorphism means "many forms." In C#, it allows methods or operators to behave differently based on the object that calls them.

Polymorphism helps in: ✅ Code reusability

✅ Extensibility

✅ Better maintainability

1️⃣ Types of Polymorphism in C#

C# supports two types of polymorphism:

Type Description Example

Compile-time Polymorphism (Static Polymorphism) Method call is resolved at compile-time Method Overloading & Operator Overloading

Run-time Polymorphism (Dynamic Polymorphism) Method call is resolved at runtime using method overriding Method Overriding (using virtual, override)

2️⃣ Compile-Time Polymorphism (Method Overloading & Operator Overloading)

✅ Method Overloading

Same method name but different parameters (number or type).

The method is selected at compile-time.

Example: Method Overloading

using System;

class MathOperations

{

public int Add(int a, int b)

{

return a + b;

}

public double Add(double a, double b)

{

return a + b;

}

public int Add(int a, int b, int c)

{

return a + b + c;

}

}

class Program

{

static void Main()

{

MathOperations obj = new MathOperations();

Console.WriteLine(obj.Add(2, 3)); // Calls Add(int, int)

Console.WriteLine(obj.Add(2.5, 3.5)); // Calls Add(double, double)

Console.WriteLine(obj.Add(1, 2, 3)); // Calls Add(int, int, int)

}

}

🔹 Output:

5

6

6

✅ Operator Overloading

Overloading operators like +, -, \*, etc., to work with user-defined types.

Example: Operator Overloading

using System;

class Complex

{

public int Real { get; set; }

public int Imaginary { get; set; }

public Complex(int real, int imaginary)

{

Real = real;

Imaginary = imaginary;

}

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

}

public void Display()

{

Console.WriteLine($"{Real} + {Imaginary}i");

}

}

class Program

{

static void Main()

{

Complex c1 = new Complex(2, 3);

Complex c2 = new Complex(4, 5);

Complex result = c1 + c2; // Operator Overloading

result.Display();

}

}

🔹 Output:

6 + 8i

3️⃣ Run-Time Polymorphism (Method Overriding)

✅ Method Overriding

Allows a derived class to provide a specific implementation of a method that is already defined in its base class.

Uses virtual in the base class and override in the derived class.

Method is selected at runtime based on the object type.

Example: Method Overriding

using System;

class Animal

{

public virtual void MakeSound() // Virtual method

{

Console.WriteLine("Animal makes a sound");

}

}

class Dog : Animal

{

public override void MakeSound() // Overriding method

{

Console.WriteLine("Dog barks");

}

}

class Program

{

static void Main()

{

Animal myAnimal = new Dog(); // Upcasting

myAnimal.MakeSound(); // Calls Dog's MakeSound() at runtime

}

}

🔹 Output:

Dog barks

4️⃣ Key Differences Between Method Overloading & Method Overriding

Feature Method Overloading Method Overriding

Definition Same method name, different parameters Same method name, same parameters

Binding Compile-time (early binding) Runtime (late binding)

Keyword Used No special keyword virtual (base class), override (derived class)

Inheritance Not required Required

Example Multiple Add() methods with different parameters MakeSound() redefined in the derived class

5️⃣ When to Use Polymorphism?

✅ Use method overloading when:

You need multiple methods that perform similar operations but with different parameters.

✅ Use method overriding when:

You want to customize behavior in derived classes while keeping a common interface.

🔹 Summary

Polymorphism allows one interface to have multiple implementations.

Compile-time polymorphism: Achieved using method overloading and operator overloading.

Run-time polymorphism: Achieved using method overriding (virtual, override

. Purpose of this Keyword in C#

The this keyword in C# is used to refer to the current instance of a class. It is useful when differentiating between class fields and method parameters, chaining constructors, and passing the current object as a parameter.

1️⃣ Common Uses of this in C#

✅ 1. Referring to Instance Variables

When a class field and a method parameter have the same name, this helps to differentiate them.

using System;

class Person

{

private string name;

public Person(string name)

{

this.name = name; // 'this.name' refers to the class field, 'name' refers to the parameter

}

public void Display()

{

Console.WriteLine($"Name: {this.name}");

}

}

class Program

{

static void Main()

{

Person p = new Person("John");

p.Display();

}

}

🔹 Output:

Name: John

✅ 2. Constructor Chaining (Calling Another Constructor in the Same Class)

this is used to call another constructor within the same class.

using System;

class Employee

{

private string name;

private int age;

// First constructor

public Employee(string name)

{

this.name = name;

}

// Second constructor calling the first constructor using 'this'

public Employee(string name, int age) : this(name)

{

this.age = age;

}

public void Display()

{

Console.WriteLine($"Name: {name}, Age: {age}");

}

}

class Program

{

static void Main()

{

Employee emp = new Employee("Alice", 25);

emp.Display();

}

}

🔹 Output:

Name: Alice, Age: 25

✅ 3. Passing the Current Instance (this) as a Parameter

You can pass the current instance of a class using this to another method or object.

using System;

class Car

{

public string Model { get; set; }

public Car(string model)

{

this.Model = model;

}

public void Show(Car car)

{

Console.WriteLine($"Car Model: {car.Model}");

}

public void Display()

{

Show(this); // Passing the current instance

}

}

class Program

{

static void Main()

{

Car myCar = new Car("Tesla Model S");

myCar.Display();

}

}

🔹 Output:

Car Model: Tesla Model S

✅ 4. Using this in Extension Methods

In extension methods, this is used to refer to the instance that the method extends.

using System;

static class StringExtensions

{

public static void Print(this string str)

{

Console.WriteLine(str);

}

}

class Program

{

static void Main()

{

"Hello, World!".Print(); // Calls the extension method

}

}

🔹 Output:

Hello, World!

2️⃣ Key Points About this Keyword

✅ Used to refer to the current instance of a class.

✅ Helps resolve naming conflicts between fields and parameters.

✅ Supports constructor chaining.

✅ Allows passing the current object as a parameter.

✅ Used in extension methods.

🔹 Directives in C#

Directives in C# are instructions to the compiler that control the compilation process. They are mainly used for code organization, conditional compilation, aliasing namespaces, and including external resources.

1️⃣ Types of Directives in C#

There are several types of directives in C#:

✅ 1. Using Directive

It allows you to import a namespace to avoid writing fully qualified names.

using System; // Allows direct use of Console class

class Program

{

static void Main()

{

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

}

}

🔹 Without using directive, you would have to write:

System.Console.WriteLine("Hello, World!");

✅ 2. Conditional Compilation Directives

Used to compile certain sections of code based on conditions.

🔹 #define and #undef

Used to define or undefine a symbol.

#define DEBUG // Define a symbol

using System;

class Program

{

static void Main()

{

#if DEBUG

Console.WriteLine("Debug mode is ON");

#else

Console.WriteLine("Debug mode is OFF");

#endif

}

}

🔹 Output:

Debug mode is ON

🔹 #if, #elif, #else, #endif

Used to include/exclude code during compilation.

#define TEST

using System;

class Program

{

static void Main()

{

#if TEST

Console.WriteLine("TEST mode active");

#else

Console.WriteLine("Normal mode");

#endif

}

}

✅ 3. Region and Endregion Directives

Used to group sections of code for better readability in Visual Studio.

using System;

class Program

{

#region User Methods

public void Login()

{

Console.WriteLine("Login Successful");

}

public void Register()

{

Console.WriteLine("User Registered");

}

#endregion

static void Main()

{

Program p = new Program();

p.Login();

p.Register();

}

}

🔹 Advantage: Collapsing and expanding large sections in the IDE.

✅ 4. Pragma Directive

Controls compiler warnings and optimizations.

🔹 #pragma warning

Suppresses specific compiler warnings.

#pragma warning disable CS0168 // Disables unused variable warning

using System;

class Program

{

static void Main()

{

int x; // No warning shown

}

}

🔹 #pragma checksum

Used in ASP.NET to track source file integrity.

✅ 5. Line Directive

Changes the line number and filename reported by the compiler (useful for debugging).

using System;

class Program

{

static void Main()

{

#line 100 "CustomFile.cs" // Sets the line number to 100 and filename to CustomFile.cs

Console.WriteLine("This line is reported as line 100 in CustomFile.cs");

}

}

✅ 6. Error and Warning Directives

Used to force compiler errors or warnings.

#error This is a forced compile error

#warning This is a compile-time warning

2️⃣ Summary

Directive Purpose

using Imports a namespace

#define, #undef Defines or undefines a symbol

#if, #elif, #else, #endif Enables conditional compilation

#region, #endregion Organizes code into collapsible sections

#pragma warning Suppresses compiler warnings

#pragma checksum Used for source file tracking

#line Changes reported line numbers and filenames

#error Forces a compilation error

#warning Triggers a compilation warning