**What is .NET?**

**.NET** is a **software development framework** created by Microsoft. It provides tools, libraries, and programming languages to build different types of applications like **web apps, desktop apps, mobile apps, games, and cloud services**.

It supports multiple languages, including **C#, VB.NET, and F#**, and allows developers to create cross-platform applications that run on **Windows, macOS, and Linux**.

**Features of .NET:**

1. **Cross-Platform** 🖥️➡️💻➡️📱
   * Works on **Windows, macOS, and Linux** using .NET Core (now merged into .NET).
2. **Multiple Language Support** 🛠️
   * Supports **C#, VB.NET, F#**, and many other languages.
3. **High Performance** ⚡
   * Optimized for **speed** and **efficiency**, especially in .NET Core and newer versions.
4. **Open-Source** 🌍
   * Most of .NET is open-source, meaning developers can contribute and improve it.
5. **Rich Library Support** 📚
   * Comes with a huge collection of pre-built **libraries** for faster development.
6. **Security** 🔒
   * Provides built-in **authentication, encryption, and secure coding** practices.
7. **Automatic Memory Management** 🧠
   * Uses **Garbage Collection (GC)** to free unused memory automatically.
8. **Support for Cloud & IoT** ☁️📡
   * Works well with **Azure, AWS, and IoT (Internet of Things)** applications.
9. **Microservices & Containerization** 🚀
   * Supports modern **Docker and Kubernetes** deployments.
10. **Compatibility with Legacy Systems** 🔄

* Can run old applications built on **.NET Framework** with some modifications.

**Differentiate between .net framework and .net core**

simple comparison between **.NET Framework** and **.NET Core**:

| **Feature** | **.NET Framework** | **.NET Core** |
| --- | --- | --- |
| **Platform** | Works only on Windows | Works on Windows, macOS, and Linux |
| **Performance** | Slower compared to .NET Core | Faster and more optimized |
| **Application Type** | Best for desktop and enterprise apps | Best for cloud, web, and modern applications |
| **Updates** | Gets fewer updates | Actively developed with frequent updates |
| **Future Support** | No longer actively developed (replaced by .NET 5+) | The future of .NET (now just called **.NET**) |

* **Use .NET Framework** if you are working with older Windows applications.
* **Use .NET Core (or newer .NET versions)** for modern, cross-platform, and high-performance applications.

**Compilation and Execution Flow of a .NET Application**

In a .NET application, the compilation and execution process involves multiple stages to ensure efficient execution on different platforms. The process follows these steps:

**Step-by-Step Flow**

1. **Writing Code (Source Code)**
   * Developers write code using languages like **C#, VB.NET, or F#**.
   * The source code is stored in files with extensions like .cs (C#) or .vb (VB.NET).
2. **Compilation (Source Code → Intermediate Code)**
   * The **C# Compiler (CSC.exe)** or respective compiler translates the source code into **Intermediate Language (IL)** (formerly known as MSIL - Microsoft Intermediate Language).
   * The compiled output is stored in **DLL (Dynamic Link Library) or EXE (Executable) files**.
3. **Common Language Runtime (CLR) Execution**
   * The **CLR** loads the IL code and converts it into **native machine code** using **Just-In-Time (JIT) Compilation**.
   * CLR also provides **garbage collection, security, and exception handling**.
4. **Execution on the Operating System**
   * The native machine code is executed by the OS, producing the final output.

**Diagram: .NET Compilation & Execution Flow**

+----------------+ +---------------------+ +--------------------+ +-------------------+

| Source Code | ----> | .NET Compiler (CSC)| ----> | Intermediate Code | ----> | CLR (JIT Compiler)|

| (.cs, .vb) | | (C#, VB.NET, F#) | | (IL/MSIL) | | Converts IL to |

| | | | | (.exe / .dll) | | Native Machine Code|

+----------------+ +---------------------+ +--------------------+ +-------------------+

↓

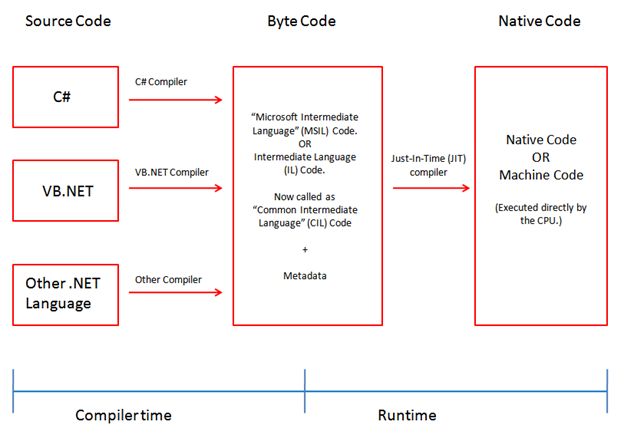
+--------------------+

| OS Execution |

| (Windows, Linux, |

| macOS) |

+--------------------+



**Key Components in the Process**

* **Compiler (CSC.exe)**: Converts **source code** into **Intermediate Language (IL)**.
* **Intermediate Language (IL/MSIL)**: Platform-independent code that is not yet machine-executable.
* **Common Language Runtime (CLR)**: Converts **IL into native code** using JIT (Just-In-Time) compilation.
* **JIT Compiler**: Translates IL into **optimized machine code** just before execution.
* **Operating System**: Finally executes the application using machine-level instructions.

**what are the project type in .net ?**

**Types of Projects in .NET**

.NET supports various project types for different kinds of applications. Below are the most common ones:

**1. Console Application 🖥️**

* A command-line application that runs in the terminal or command prompt.
* Example: Simple calculators, automation scripts.
* File Extension: .exe

**2. Windows Forms (WinForms) Application 🏠**

* A graphical desktop application with a user-friendly UI.
* Uses drag-and-drop components like buttons, text boxes, and forms.
* Example: Notepad-like applications.
* File Extension: .exe

**3. WPF (Windows Presentation Foundation) Application 🎨**

* A desktop application with advanced UI capabilities using **XAML**.
* Supports animations, styles, and media elements.
* Example: Media players, dashboards.
* File Extension: .exe

**4. ASP.NET Web Application 🌐**

* A web-based application that runs in browsers.
* Uses **ASP.NET MVC, Web API, and Blazor** for front-end and back-end development.
* Example: E-commerce websites, blogs, admin panels.
* File Extension: .dll (runs on a web server)

**5. ASP.NET Web API 🔄**

* A backend service that provides **RESTful APIs**.
* Used for communication between client apps (mobile, web) and databases.
* Example: API for mobile apps, weather services.
* File Extension: .dll

**6. Blazor Application ⚛️**

* A modern web application framework using **C# and Razor** instead of JavaScript.
* Can run **on the server (Blazor Server)** or **in the browser (Blazor WebAssembly)**.
* Example: Interactive web apps, dashboards.
* File Extension: .dll

**7. Class Library 📚**

* A reusable **.dll** file containing business logic, functions, and utilities.
* Used by multiple applications to share common code.
* Example: A **math library** used in multiple projects.
* File Extension: .dll

**8. .NET MAUI (Multi-platform App UI) 📱💻**

* A cross-platform framework for **mobile and desktop** apps.
* Successor to **Xamarin.Forms**.
* Example: Mobile banking apps, cross-platform utilities.
* File Extension: .exe / .apk / .ipa

**9. Xamarin App (Deprecated, replaced by .NET MAUI) 📲**

* Used to build **iOS and Android** mobile apps with **C#**.
* Example: Social media apps, fitness trackers.
* File Extension: .apk / .ipa

**10. Worker Service ⚙️**

* A background process that runs continuously without UI.
* Example: Email schedulers, data processing services.
* File Extension: .exe

**11. Azure Cloud Services ☁️**

* Applications designed to run on **Microsoft Azure** for cloud computing.
* Example: Serverless functions, cloud storage management.
* File Extension: .dll

**12. Test Project (Unit Testing) 🧪**

* Used for writing and running automated tests for applications.
* Example: **xUnit, NUnit, MSTest** projects for testing logic.
* File Extension: .dll

**What is C#?**

C# (C-Sharp) is a modern, object-oriented programming language developed by Microsoft as part of the .NET framework. It is widely used for desktop, web, mobile, cloud, and game development.

C# is designed to be simple, powerful, and flexible, making it one of the most popular programming languages today.

Features of C#

1. **Object-Oriented Programming (OOP) 🏛️**
   * Supports OOP concepts like classes, inheritance, polymorphism, encapsulation, and abstraction.
   * Example: You can create objects and reuse code efficiently.
2. **Type Safety 🔒**
   * Prevents errors by enforcing strong type checking at compile-time.
   * Example: Prevents unintended type conversions.
3. **Garbage Collection 🗑️**
   * C# has automatic memory management, so you don't have to manually free memory.
   * Helps avoid memory leaks and improves performance.
4. **Cross-Platform Support 🌍**
   * Works on Windows, Linux, and macOS with .NET Core/.NET 5+.
5. **Rich Library Support 📚**
   * C# provides a vast set of pre-built libraries (Framework Class Library - FCL) for file handling, networking, database access, and more.
6. **Multithreading & Asynchronous Programming ⚡**
   * Supports parallel processing using async/await and Task-based programming for better performance.
   * Example: Makes web applications faster by handling multiple requests simultaneously.
7. **Interoperability 🔄**
   * Can interact with other languages (C++, JavaScript, etc.) and older technologies like COM (Component Object Model).
8. **LINQ (Language Integrated Query) 🔍**
   * Provides easy database queries directly in C# without using SQL.
   * Example:

var numbers = new List<int> {1, 2, 3, 4, 5};

var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();

1. **Exception Handling ⚠️**
   * Helps catch and manage errors using try, catch, and finally blocks.
   * Example:

try { int x = 10 / 0; }

catch (Exception ex) { Console.WriteLine(ex.Message); }

1. **Security Features 🔐**

* Supports encryption, authentication, and secure coding practices.
* Example: Used in banking and enterprise applications.

1. **Platform Independence with .NET Core/.NET 🌐**

* C# applications can now run on any OS (Windows, Linux, macOS) with .NET.

1. **Used in Game Development (Unity Engine) 🎮**

* C# is the main language for Unity, one of the most popular game development engines.

**5. Difference Between Value Type and Reference Type in C#**

In C#, **data types** are classified into **Value Types** and **Reference Types** based on how they store data in memory.

**1. Value Type 🖥️**

* Stores **data directly in memory (stack)**.
* **Faster** access since it holds the actual value.
* Each variable gets **its own copy** of the data.
* **Examples**: int, float, char, bool, struct, enum

**Example of Value Type:**

int a = 10;

int b = a; // b gets a copy of 'a', not a reference.

b = 20; // Changing 'b' does not affect 'a'

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

Console.WriteLine(b); // Output: 20

🔹 **Explanation**: b gets a separate copy of a, so changes in b do not affect a.

**2. Reference Type 🔗**

* Stores **a reference (address) in memory (heap)**.
* Variables **point to the same memory location**.
* Changes in one variable affect the other.
* **Examples**: class, string, array, object, delegate

**Example of Reference Type:**

class Person {

public string Name;

}

Person p1 = new Person();

p1.Name = "Alice";

Person p2 = p1; // p2 now refers to the same object as p1.

p2.Name = "Bob"; // Changing 'p2' affects 'p1'

Console.WriteLine(p1.Name); // Output: Bob

Console.WriteLine(p2.Name); // Output: Bob

🔹 **Explanation**: p1 and p2 point to the same object in memory. Modifying p2.Name also changes p1.Name.

**Key Differences Between Value Type and Reference Type**

| **Feature** | **Value Type** | **Reference Type** |
| --- | --- | --- |
| **Memory Location** | Stored in **stack** | Stored in **heap** (reference in stack) |
| **Stores** | Actual value | Memory address (reference) |
| **Copy Behavior** | Creates a **new copy** | Copies the **reference (same object)** |
| **Speed** | Faster (stack is quicker) | Slower (heap requires garbage collection) |
| **Examples** | int, double, bool, char, struct | class, array, string, object, delegate |

**6. Difference Between for and foreach Loop in C#**

In C#, both for and foreach loops are used for iteration, but they have key differences in usage and behavior.

**1. for Loop 🔄**

* Used when the **number of iterations is known**.
* Allows **modification** of elements (if applicable).
* Provides **index-based** access to elements.
* Can be used for **both arrays and lists**.

**Example: Using for Loop**

int[] numbers = { 10, 20, 30, 40 };

for (int i = 0; i < numbers.Length; i++) {

Console.WriteLine(numbers[i]); // Accessing elements using index

}

📌 **Use Case:**

* Best when **indexing is needed** (e.g., modifying elements, accessing specific positions).
* Example: **Processing an array** while keeping track of index positions.

**2. foreach Loop 🔁**

* Used when **iterating through a collection without modifying it**.
* Automatically handles iteration **without an index**.
* **Read-only**: Cannot modify elements directly.

**Example: Using foreach Loop**

int[] numbers = { 10, 20, 30, 40 };

foreach (int num in numbers) {

Console.WriteLine(num); // No need for an index

}

📌 **Use Case:**

* Best when you **only need to read** elements (e.g., **displaying a list of items**).
* Example: **Iterating over collections** like List<T>, Dictionary<TKey, TValue>, Array.

**Key Differences Between for and foreach**

| **Feature** | **for Loop** | **foreach Loop** |
| --- | --- | --- |
| **Use Case** | When index-based access is required | When iterating without modification |
| **Indexing** | Provides **index access (i)** | No index, iterates over elements directly |
| **Modification** | Allows modifying collection elements | **Read-only**, cannot modify elements directly |
| **Performance** | Faster for **arrays** (direct access) | Slower for **large collections** due to extra iteration handling |
| **Best for** | **Arrays, Lists (when modifying elements)** | **Collections (when reading elements)** |

**When to Use Which?**

✅ **Use for when:**

* You need **index-based access**.
* You want to **modify elements** inside the loop.
* You work with **fixed-size collections** (like arrays).

✅ **Use foreach when:**

* You only need to **read elements** from a collection.
* You are working with **collections like List, Dictionary, or IEnumerable**.
* You want **cleaner, more readable code**.

**Example Showing the Difference**

int[] numbers = { 1, 2, 3, 4, 5 };

// Using 'for' to modify elements

for (int i = 0; i < numbers.Length; i++) {

numbers[i] \*= 2; // Modifying elements

Console.WriteLine(numbers[i]); // 2, 4, 6, 8, 10

}

// Using 'foreach' (Read-only, will not modify)

foreach (int num in numbers) {

Console.WriteLine(num); // Outputs modified values

}

**Class and Object in C#**

In C#, **a class is a blueprint** for creating objects, and **an object is an instance** of a class.

**1. What is a Class? 🏛️**

* A **class** is a template that defines **properties (variables)** and **methods (functions)**.
* It **does not store data**; instead, it is used to create objects.
* Declared using the class keyword.

**Example of a Class**

class Car // Class definition

{

// Properties (Variables)

public string Brand;

public int Speed;

// Method (Function)

public void ShowDetails() {

Console.WriteLine($"Car Brand: {Brand}, Speed: {Speed} km/h");

}

}

**2. What is an Object? 🚗**

* **An object is an instance of a class** that stores actual data.
* Created using the new keyword.
* Each object has **its own copy of the class properties**.

**Example of Creating Objects**

class Program

{

static void Main()

{

// Creating an object of the Car class

Car car1 = new Car();

car1.Brand = "Toyota"; // Assigning values

car1.Speed = 120;

Car car2 = new Car();

car2.Brand = "Honda";

car2.Speed = 100;

// Calling method

car1.ShowDetails(); // Output: Car Brand: Toyota, Speed: 120 km/h

car2.ShowDetails(); // Output: Car Brand: Honda, Speed: 100 km/h

}

}

📌 **Explanation:**

* **Car is a class** (template).
* **car1 and car2 are objects** that store individual data.
* ShowDetails() method displays object data.

**Key Differences Between Class and Object**

| **Feature** | **Class** | **Object** |
| --- | --- | --- |
| **Definition** | A blueprint for objects | An instance of a class |
| **Memory Usage** | No memory allocated | Memory allocated in heap |
| **Stores Data?** | No | Yes |
| **Example** | Car | car1, car2 |

**Constructor and Destructor in C#**

In C#, **constructors** and **destructors** are special methods used for object **initialization** and **cleanup**.

**1. Constructor 🚀**

A **constructor** is a special method that is automatically called **when an object is created**.

**Features of Constructor**

✔ **Same name as the class**  
✔ **No return type (not even void)**  
✔ **Automatically invoked when the object is created**  
✔ Used to **initialize** class properties

**Types of Constructors in C#**

1. **Default Constructor** (No parameters)
2. **Parameterized Constructor** (Takes arguments)
3. **Copy Constructor** (Copies values from another object)
4. **Static Constructor** (Runs once for the class)

**Example: Constructor in C#**

class Car

{

public string Brand;

public int Speed;

// Constructor (same name as class)

public Car(string brand, int speed)

{

Brand = brand;

Speed = speed;

Console.WriteLine("Car object created!");

}

public void ShowDetails()

{

Console.WriteLine($"Brand: {Brand}, Speed: {Speed} km/h");

}

}

class Program

{

static void Main()

{

// Creating objects (Constructor automatically called)

Car car1 = new Car("Toyota", 120);

car1.ShowDetails();

Car car2 = new Car("Honda", 100);

car2.ShowDetails();

}

}

**Output**

*Car object created!*

*Brand: Toyota, Speed: 120 km/h*

*Car object created!*

*Brand: Honda, Speed: 100 km/h*

📌 **Use Case:**

* Used for **automatic initialization** of objects.
* Example: Setting **default values** when creating an object.

**2. Destructor 🗑️**

A **destructor** is a special method that is called **when an object is destroyed** to free resources.

**Features of Destructor**

✔ **Same name as the class, prefixed with ~**  
✔ **No parameters and no return type**  
✔ **Automatically called before garbage collection**  
✔ Used for **cleaning up memory/resources**

**Example: Destructor in C#**

class Car

{

public string Brand;

// Constructor

public Car(string brand)

{

Brand = brand;

Console.WriteLine($"{Brand} car created!");

}

// Destructor

~Car()

{

Console.WriteLine($"{Brand} car is destroyed!");

}

}

class Program

{

static void Main()

{

Car car1 = new Car("BMW");

} // Destructor will be called when the object goes out of scope

**Output (When object is destroyed)**

*BMW car created!*

*BMW car is destroyed!*

📌 **Use Case:**

* Used for **resource cleanup**, like **closing database connections, freeing memory, or disposing of files**.

**Key Differences Between Constructor and Destructor**

| **Feature** | **Constructor 🚀** | **Destructor 🗑️** |
| --- | --- | --- |
| **Purpose** | Initializes an object | Cleans up resources before object destruction |
| **Name** | Same as class | Same as class, prefixed with ~ |
| **Called When?** | When an object is created | When an object is garbage collected |
| **Parameters?** | Can have parameters | No parameters |
| **Explicit Call?** | Called automatically when an object is created | Called automatically before garbage collection |

**Four Pillars of Object-Oriented Programming (OOP) in C#**

OOP in C# is based on **four main pillars**:

1. **Encapsulation** 🏠
   * **Hides data** and restricts direct access to it.
   * **Uses access modifiers** like private, protected, and public.
   * Example:

class Person {

private string name; // Private variable

public void SetName(string newName) { name = newName; }

public string GetName() { return name; }

}

* + 🔹 **Use Case**: Protecting sensitive data in applications (e.g., user passwords).

1. **Abstraction** 🎭
   * **Hides implementation details** and shows only necessary functionality.
   * Achieved using **abstract classes or interfaces**.
   * Example:

abstract class Shape {

public abstract void Draw(); // Only method signature, no implementation

}

class Circle : Shape {

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

}

* + 🔹 **Use Case**: Hiding database operations behind a class interface.

1. **Inheritance** 👪
   * Allows **a class (child) to inherit properties and methods** from another class (parent).
   * **Reusability**: Avoids code duplication.
   * Uses : (colon) syntax in C#.
2. **Polymorphism** 🎭
   * **One method, multiple behaviors** (method overloading & method overriding).
   * **Compile-time polymorphism**: Method overloading.
   * **Runtime polymorphism**: Method overriding.
   * Example:

class Animal {

public virtual void Speak() { Console.WriteLine("Animal speaks"); }

}

class Dog : Animal {

public override void Speak() { Console.WriteLine("Dog barks"); }

}

* + 🔹 **Use Case**: Creating flexible APIs where different classes can share a common interface.

**Inheritance in C# (With Code Example)**

Inheritance allows one class to **reuse the properties and methods** of another class.

**Example of Single Inheritance**

// Parent class (Base class)

class Animal {

public string Name;

public void Display() {

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

}

}

// Child class (Derived class)

class Dog : Animal {

public void Bark() {

Console.WriteLine(Name + " is barking");

}

}

class Program {

static void Main() {

Dog myDog = new Dog();

myDog.Name = "Buddy"; // Inheriting property from Animal

myDog.Display(); // Calling inherited method

myDog.Bark(); // Calling Dog's own method

}

}

**Output**

*Animal Name: Buddy*

*Buddy is barking*

📌 **Explanation**

* Dog **inherits** properties and methods from Animal.
* myDog.Name is **inherited** from Animal.
* myDog.Display() calls a **base class method**.
* myDog.Bark() is **unique to Dog**.

**Types of Inheritance in C#**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Single Inheritance** | One child class inherits from one parent class. | Dog inherits from Animal |
| **Multilevel Inheritance** | A class inherits from another child class. | Puppy → Dog → Animal |
| **Hierarchical Inheritance** | Multiple child classes inherit from the same parent. | Dog & Cat inherit from Animal |
| **Multiple Inheritance (Not supported directly in C#)** | A class inherits from multiple classes (achieved via interfaces). | Car inherits from Vehicle & Engine (via interfaces) |

**Method Overloading vs. Method Overriding in C#**

Both **method overloading** and **method overriding** are essential concepts in C# **polymorphism**, but they differ in how they allow the reuse and modification of methods.

**1. Method Overloading 🛠️**

* **Definition**: Method overloading allows a **class to have multiple methods with the same name** but different **parameter lists** (number or type of parameters).
* **Occurs at compile-time** (Compile-time polymorphism).
* The method signature must be **different** in the parameters, but the method name remains the same.
* **Does not change the method in the base class**, instead, the method is **redefined within the same class**.

**Example of Method Overloading:**

class Calculator

{

// Method to add two integers

public int Add(int a, int b)

{

return a + b;

}

// Overloaded method to add three integers

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

{

return a + b + c;

}

// Overloaded method to add two floating-point numbers

public float Add(float a, float b)

{

return a + b;

}

}

class Program

{

static void Main()

{

Calculator calc = new Calculator();

// Calling overloaded methods

Console.WriteLine(calc.Add(10, 20)); // Calls Add(int, int)

Console.WriteLine(calc.Add(10, 20, 30)); // Calls Add(int, int, int)

Console.WriteLine(calc.Add(10.5f, 20.5f)); // Calls Add(float, float)

}

}

**Output:**

*30*

*60*

*31*

**Explanation**:

* The Add method is **overloaded** to accept **different parameter types** (int, int, int) and (float, float).
* The compiler **chooses the correct method** based on the arguments passed.

**2. Method Overriding 🔄**

* **Definition**: Method overriding allows a **child class to provide a specific implementation** of a method that is already defined in its **base class**.
* **Occurs at runtime** (Runtime polymorphism).
* The method must be **marked as virtual** in the base class and **override** in the derived class.
* Used to modify or extend the behavior of a method in the derived class while keeping the same method signature.

**Example of Method Overriding:**

class Animal

{

// Base class method

public virtual void Speak()

{

Console.WriteLine("Animal speaks");

}

}

class Dog : Animal

{

// Overriding base class method

public override void Speak()

{

Console.WriteLine("Dog barks");

}

}

class Program

{

static void Main()

{

Animal myAnimal = new Animal();

myAnimal.Speak(); // Output: Animal speaks

Dog myDog = new Dog();

myDog.Speak(); // Output: Dog barks

Animal animalDog = new Dog();

animalDog.Speak(); // Output: Dog barks (polymorphism)

}

}

**Output:**

*Animal speaks*

*Dog barks*

*Dog barks*

**Explanation**:

* The Speak method is **overridden** in the Dog class, changing its behavior.
* Even though the animalDog variable is of type Animal, it calls the Dog's overridden version of Speak (this is **runtime polymorphism**).

**Key Differences Between Method Overloading and Method Overriding**

| **Feature** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| **Definition** | Multiple methods with the same name but different parameters. | A derived class provides its own implementation of a method defined in the base class. |
| **Occurs at** | **Compile-time** (Static Polymorphism) | **Runtime** (Dynamic Polymorphism) |
| **Method Signature** | Must differ in the **number** or **type** of parameters. | Method signature must be **the same** as the base class method. |
| **Inheritance** | No inheritance involved; occurs within the same class. | Requires **inheritance**; involves a base class and a derived class. |
| **Keywords Used** | No keywords; just different method signatures. | **virtual** in the base class, **override** in the derived class. |
| **Purpose** | To define multiple methods with the same name for different tasks. | To change the behavior of a method in the derived class. |

**Abstract Class vs. Interface in C#**

Both **abstract classes** and **interfaces** are used to achieve **abstraction** in C#, but they serve different purposes and have different features. Let's explore the key differences and their use cases.

**1. Abstract Class**

* **Definition**: An **abstract class** is a class that **cannot be instantiated directly**. It can have both **abstract methods** (without implementation) and **non-abstract methods** (with implementation).
* **Purpose**: Used when you want to provide a common base for derived classes, allowing both shared code and forcing the derived classes to implement certain methods.

**Key Features of Abstract Class**

* Can have **both abstract and non-abstract methods**.
* Can have **fields**, **properties**, and **constructors**.
* Can provide **default implementation** for methods.
* A class can **inherit** from **only one abstract class** (C# supports single inheritance).
* Can have **access modifiers** (like private, protected, public).

**2. Interface**

* **Definition**: An **interface** defines a contract of methods that a class must implement. It only contains method declarations, properties, events, or indexers, but **no implementation**.
* **Purpose**: Used when you want to enforce a contract that multiple classes can implement, allowing them to share the same set of method signatures but with different implementations.

**Key Features of Interface**

* Can only have **method signatures** (no method body).
* Can have **properties**, **events**, and **indexers**, but **no implementation**.
* A class can implement **multiple interfaces** (C# supports multiple interface inheritance).
* All methods in an interface are **implicitly public**.
* Cannot have **fields**, **constructors**, or **destructors**.

**Key Differences Between Abstract Class and Interface**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Purpose** | Used to provide a base class with some default behavior and force derived classes to implement specific methods. | Used to define a contract or set of methods that multiple classes must implement. |
| **Methods** | Can have both **abstract methods** (without implementation) and **non-abstract methods** (with implementation). | Can only contain **method declarations** (no method body). |
| **Fields** | Can have fields (variables). | Cannot have fields. |
| **Constructors** | Can have constructors to initialize data. | Cannot have constructors. |
| **Multiple Inheritance** | Can only **inherit** from **one** abstract class. | A class can **implement multiple interfaces**. |
| **Access Modifiers** | Can have **access modifiers** (like private, protected, etc.). | All members are **implicitly public**. |
| **Default Implementation** | Can provide **default method implementations**. | Cannot provide any implementation, only method signatures. |

**Example of Abstract Class**

// Abstract class

abstract class Animal

{

public string Name { get; set; }

// Abstract method (must be implemented by derived class)

public abstract void Speak();

// Non-abstract method (can be used by derived class)

public void Eat()

{

Console.WriteLine("Animal is eating.");

}

}

class Dog : Animal

{

public override void Speak()

{

Console.WriteLine("Dog barks");

}

}

class Program

{

static void Main()

{

Dog dog = new Dog();

dog.Name = "Buddy";

Console.WriteLine(dog.Name);

dog.Speak(); // Calls overridden method

dog.Eat(); // Calls base class method

}

}

**Output:**

*Buddy*

*Dog barks*

*Animal is eating.*

📌 **Use Case for Abstract Class**:

* When you want to provide **common functionality** (like the Eat() method) and force derived classes to implement certain methods (like Speak() in the Dog class).

**Example of Interface**

// Interface

interface IAnimal

{

void Speak();

void Eat();

}

class Dog : IAnimal

{

public void Speak()

{

Console.WriteLine("Dog barks");

}

public void Eat()

{

Console.WriteLine("Dog is eating");

}

}

class Cat : IAnimal

{

public void Speak()

{

Console.WriteLine("Cat meows");

}

public void Eat()

{

Console.WriteLine("Cat is eating");

}

}

class Program

{

static void Main()

{

IAnimal dog = new Dog();

dog.Speak(); // Dog barks

dog.Eat(); // Dog is eating

IAnimal cat = new Cat();

cat.Speak(); // Cat meows

cat.Eat(); // Cat is eating

}

}

**Output:**

*Dog barks*

*Dog is eating*

*Cat meows*

*Cat is eating*

📌 **Use Case for Interface**:

* When you need to ensure that **multiple unrelated classes** implement the same set of methods, for example, IAnimal can be implemented by both Dog and Cat, even though they don't share a common parent class.

**When to Use Abstract Class vs. Interface**

| **Use Case** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Common functionality** | Use when you want to provide common functionality (methods with implementation) in the base class. | Use when no functionality is needed in the base (only method signatures). |
| **Multiple inheritances** | Use when a class will only need to inherit from one base class. | Use when a class needs to implement **multiple contracts**. |
| **Default behavior** | Use when you want to provide a **default implementation** for some methods. | Interfaces cannot provide default implementations (until C# 8, where default interface methods were introduced, but this is still limited). |
| **Design contract** | Use when defining a **base class** with shared behavior. | Use when you need to enforce a **contract** across unrelated classes. |

**Abstraction in C#**

**Abstraction** is one of the four pillars of Object-Oriented Programming (OOP). It allows you to **hide the implementation details** and only expose the necessary functionality to the user. This helps in reducing complexity and increasing efficiency.

In C#, **abstraction** is typically implemented using:

1. **Abstract Classes**
2. **Interfaces**

Let's dive into both methods with examples:

**1. Abstraction Using Abstract Class**

An **abstract class** is a class that cannot be instantiated. It may contain both **abstract methods** (without implementation) and **concrete methods** (with implementation). The derived classes are required to implement the abstract methods.

**Example: Abstraction with Abstract Class**

// Abstract class (base class)

abstract class Shape

{

// Abstract method (must be implemented by derived class)

public abstract void Draw();

// Concrete method (common functionality for all derived classes)

public void Display()

{

Console.WriteLine("Displaying shape.");

}

}

// Derived class: Circle

class Circle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a circle");

}

}

// Derived class: Rectangle

class Rectangle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a rectangle");

}

}

class Program

{

static void Main()

{

// Cannot instantiate an abstract class directly

// Shape shape = new Shape(); // This will give an error

Shape shape1 = new Circle(); // Create object of derived class

shape1.Draw(); // Calls Circle's Draw method

shape1.Display(); // Calls the common Display method from base class

Shape shape2 = new Rectangle(); // Create object of derived class

shape2.Draw(); // Calls Rectangle's Draw method

shape2.Display(); // Calls the common Display method from base class

}

}

**Output:**

*Drawing a circle*

*Displaying shape.*

*Drawing a rectangle*

*Displaying shape.*

**Explanation:**

* **Abstract class Shape** defines the **abstract method Draw()** that must be implemented by any class that inherits Shape.
* The method Display() is common for all shapes and is implemented in the abstract class.
* **Circle and Rectangle** are derived classes that **provide their own implementation** of the Draw() method.
* In this way, abstraction allows the **user to interact with the Shape class**, but the exact **details of how each shape is drawn are hidden**.

**2. Abstraction Using Interface**

An **interface** defines a contract that the implementing classes must follow. It **cannot provide implementation** but can declare methods, properties, and events that must be implemented by the classes.

**Example: Abstraction with Interface**

// Interface (contract)

interface IDrawable

{

void Draw(); // Method signature without implementation

}

// Class implementing the interface

class Circle : IDrawable

{

public void Draw()

{

Console.WriteLine("Drawing a circle");

}

}

// Class implementing the interface

class Rectangle : IDrawable

{

public void Draw()

{

Console.WriteLine("Drawing a rectangle");

}

}

class Program

{

static void Main()

{

IDrawable shape1 = new Circle(); // Create object of class Circle

shape1.Draw(); // Calls Circle's Draw method

IDrawable shape2 = new Rectangle();// Create object of class Rectangle

shape2.Draw(); // Calls Rectangle's Draw method

}

}

**Output:**

*Drawing a circle*

*Drawing a rectangle*

**Explanation:**

* The **IDrawable interface** defines the contract that any class implementing it must provide an implementation for the Draw() method.
* **Circle** and **Rectangle** implement the **IDrawable interface** and provide their own behavior for Draw().
* This allows different classes to share the same **method signature** but implement it differently.
* In this way, abstraction ensures that you **only see the contract** (the Draw() method) and not how the drawing is done.

**Comparison of Abstract Class and Interface for Abstraction**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Purpose** | To provide a common base with shared functionality and enforce certain method implementations. | To define a contract that must be implemented by multiple classes. |
| **Implementation** | Can have both **abstract methods** and **concrete methods** (methods with implementation). | Can only contain **method declarations** (no implementation). |
| **Multiple Inheritance** | A class can **inherit** from only **one abstract class**. | A class can **implement multiple interfaces**. |
| **Fields and Constructors** | Can have **fields** and **constructors**. | Cannot have **fields** or **constructors**. |
| **Use Case** | Use when you need shared functionality and enforce method implementation. | Use when you want to define a common set of methods that must be implemented by different classes. |

**Use Case for Abstraction**

1. **Abstract Class**:
   * When you have **common functionality** that you want to share among different classes.
   * Example: Shape class with methods like Display() that can be used by multiple derived shapes (Circle, Rectangle, etc.), while forcing them to implement the Draw() method.
2. **Interface**:
   * When you want to **define a contract** for classes that are **unrelated** (do not share a common base class) but must implement a specific set of methods.
   * Example: An IDrawable interface that can be implemented by any class (Circle, Rectangle, etc.) that needs to be drawn, regardless of the class hierarchy.

**Encapsulation in C#**

**Encapsulation** is one of the fundamental principles of Object-Oriented Programming (OOP). It is the concept of **wrapping data (fields)** and the **methods (functions)** that operate on the data into a single unit (class), and restricting direct access to some of the object's components. This is achieved by using **access modifiers** to control how the data is accessed and modified.

The main goal of **encapsulation** is to protect the internal state of an object from outside interference and misuse. It provides a way to ensure that the object’s data is always in a valid state.

**Problem Encapsulation Solves**

Encapsulation addresses the following problems:

1. **Data Protection**: It restricts direct access to sensitive data, allowing only certain operations on the data, thereby ensuring that the internal state of the object is not changed in an unexpected or invalid way.
2. **Code Maintenance**: It allows internal implementation changes without affecting other parts of the program that rely on the class, because external code interacts with the class only through well-defined interfaces.
3. **Abstraction**: It hides the implementation details from the outside world, exposing only the necessary functionality, which makes the system easier to understand and use.
4. **Data Integrity**: It ensures that only valid data can be assigned to an object’s properties, preventing invalid or corrupted data from being stored.

**How Encapsulation Works in C#**

Encapsulation in C# is implemented using **classes**, **properties**, and **access modifiers** like private, protected, and public.

* **Private fields**: These are the data members of a class that cannot be directly accessed from outside the class.
* **Public methods (getters and setters)**: These methods are used to access or modify the private fields.

**Code Example: Encapsulation in C#**

class Person

{

// Private fields (encapsulation)

private string name;

private int age;

// Public property to get and set 'name'

public string Name

{

get { return name; }

set { name = value; } // Can add validation here

}

// Public property to get and set 'age'

public int Age

{

get { return age; }

set

{

if (value >= 0) // Validation logic

age = value;

else

Console.WriteLine("Age cannot be negative!");

}

}

// Method to display the person's details

public void DisplayDetails()

{

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

}

}

class Program

{

static void Main()

{

// Create an instance of the Person class

Person person = new Person();

// Set properties using encapsulated setter methods

person.Name = "John";

person.Age = 30;

// Display person's details

person.DisplayDetails();

// Try to set an invalid age

person.Age = -5; // This will print an error message

// Display person's details after invalid age attempt

person.DisplayDetails();

}

}

**Output:**

*Name: John, Age: 30*

*Age cannot be negative!*

*Name: John, Age: 30*

**Explanation:**

* The name and age fields are **private**, meaning they cannot be accessed directly from outside the Person class.
* The **properties** Name and Age provide controlled access to these private fields. You can add additional logic in the **setter** (like validation for age), ensuring the object’s state remains valid.
* The method DisplayDetails() is used to display the details of the person.
* If you try to set an invalid age (like -5), the **setter of Age** will prevent it, protecting the object’s integrity.

**Advantages of Encapsulation**

1. **Data Security**: By restricting access to the internal state of an object, you ensure that only valid data is used. For example, you can validate data before it is set or processed.
2. **Flexibility**: Internal implementation details can change without affecting the outside code that uses the class. The users of the class only interact with the public methods or properties, so internal changes are hidden.
3. **Control Over Data**: You can add logic to the getters and setters, allowing you to control how data is accessed or updated. For example, in the Age property, we ensure that the age cannot be set to a negative value.
4. **Code Reusability and Maintenance**: Encapsulation improves modularity, making the code more reusable and easier to maintain. You can modify or extend the internal logic of a class without affecting how it is used by other parts of the program.

**Encapsulation vs. Access Modifiers**

* **Private**: The member can only be accessed within the class itself. It's the most restrictive access modifier.
* **Protected**: The member can be accessed within the class and by derived classes.
* **Public**: The member is accessible from anywhere in the program.