.NET

.NET Versions

- ** .NET Framework:** The original .NET platform, primarily for Windows desktop and web applications. It is no longer actively developed with new feature releases, but 4.8 is the last major version and continues to receive maintenance updates.
- ** .NET Core:** A cross-platform, open-source, and high-performance framework that was a complete re-architecture. It was designed for modern, cloud-native applications.
- ** .NET:** Starting with .NET 5.0, Microsoft unified .NET Core and the best of .NET Framework into a single, cross-platform product simply called ".NET". There is no .NET 4.x (after .NET Framework 4.8) or .NET Core 4.x. The numbering continued from .NET Core 3.1 to .NET 5.0.
- ** .NET Standard:** A formal specification of .NET APIs that are available on all .NET implementations (like .NET Framework, .NET Core, Xamarin, etc.). It enables developers to build libraries that can be used across different .NET platforms. It's not a runtime itself, but a contract. .NET Standard 2.1 is the last version, as the need for it diminished with the unification of .NET.

Release	.NET Framework	.NET Core	.NET	.NET Standard
Feb 2002	1.0			
Apr 2003	1.1			
Nov 2005	2.0			
Nov 2006	3.0			
Nov 2007	3.5			
Apr 2010	4.0			
Aug 2012	4.5			1.0, 1.1
Oct 2013	4.5.1			
May 2014	4.5.2			1.2

Release	.NET Framework	.NET Core	.NET	.NET Standard
Jul 2015	4.6			
Nov 2015	4.6.1			1.3
Jun 2016		1.0		1.4, 1.5
Aug 2016	4.6.2			1.6
Nov 2016		1.1		
May 2017	4.7			
Aug 2017		2.0		2.0
Oct 2017	4.7.1			
Apr 2018	4.7.2			
May 2018		2.1		
Dec 2018		2.2		
Apr 2019	4.8			
Sep 2019		3.0		2.1
Dec 2019		3.1		
Nov 2020			5.0	
Nov 2021			6.0	
Nov 2022			7.0	
Nov 2023			8.0	

C# Language versions

C# Version	Release Month-Year	Corresponding .NET Platform Version(s)	Most Important New Language Features
C# 1.0	Jan 2002	.NET Framework 1.0, 1.1	The foundational C# language: Classes, Structs, Interfaces, Delegates, Events, Properties, Attributes, basic error handling (try-catch-finally).
C# 2.0	Nov 2005	.NET Framework 2.0, 3.0	**Generics**, Partial Classes, Anonymous Methods, Iterators (`yield return`), Nullable Types (`?`), Covariance/Contravariance for delegates.
C# 3.0	Nov 2007	.NET Framework 3.5	**LINQ (Language Integrated Query)**, **Lambda Expressions**, Extension Methods, Anonymous Types, Object and Collection Initializers, Implicitly Typed Local Variables (`var`), Expression Trees.
C# 4.0	Apr 2010	.NET Framework 4.0	`dynamic` Keyword, Named and Optional Arguments, Generic Covariance and Contravariance (for interfaces and delegates), Embedded Interop Types.
C# 5.0	Aug 2012	.NET Framework 4.5	**Async and Await** (asynchronous programming support).
C# 6.0	Jul 2015	.NET Framework 4.6, .NET Core 1.0, 1.1	**String Interpolation**, Null-Conditional Operator (`?.`), Auto-Property Initializers, Expression-Bodied Members, `using static`, `nameof` operator, Exception Filters.
C# 7.0	Mar 2017	.NET Framework 4.7, .NET Core 2.0, 2.1, 2.2	**Pattern Matching** (`is` expression, `switch` statement enhancements), **Tuples and Deconstruction**, Local Functions, `out` variables, `ref` locals and returns, `throw` expressions.
C# 8.0	Sep 2019	.NET Core 3.0, 3.1	**Nullable Reference Types**, **Default Interface Methods**, Asynchronous Streams (`IAsyncEnumerable`), Indices and Ranges (`^`, `..`), `using` declarations, Switch Expressions, Property Patterns, Positional Patterns.
C# 9.0	Nov 2020	.NET 5.0	**Records**, **Top-level Statements**, Pattern Matching Enhancements (type patterns, relational patterns, logical patterns: `and`, `or`, `not`), `init` only setters, Target-typed `new` expressions.
C# 10.0	Nov 2021	.NET 6.0	**File-Scoped Namespaces**, **Global `using` directives**, Record structs, `const` interpolated strings, Enhanced Lambda Expressions, `With` expressions for structs.
C# 11.0	Nov 2022	.NET 7.0	**Raw String Literals**, Generic Attributes, List Patterns, `required` members, `static abstract` members in interfaces (for generic math), `file` local types.

C# Version	Release Month-Year	Corresponding .NET Platform Version(s)	Most Important New Language Features
C# 12.0	Nov 2023	.NET 8.0	**Primary Constructors** (for non-record types), **Collection Expressions** (`[]` for collections), Alias any type (`using` aliases for `any` type), `ref readonly` parameters, Inline arrays, Interceptors (experimental).

Namespaces in C# .NET

Introduction

Namespaces were introduced in the very first version of C# (2002) as a fundamental organizational feature. They were designed to solve the problem of naming collisions in large projects and to provide a logical hierarchy for organizing related functionality. The concept was borrowed from C++ but implemented with more rigor and better integration with the .NET type system.

Definition and Purpose

1. **Logical Organization**: Namespaces provide a way to organize related classes, structs, interfaces, enums and delegates into logical groups. This organization mirrors the way we organize files into folders in a file system.
2. **Name Collision Prevention**: They prevent naming conflicts by allowing two classes with the same name to coexist as long as they are in different namespaces. For example, both System.IO.File and MyApp.IO.File can exist simultaneously.
3. **Scope Delimitation**: Namespaces define a declarative region that helps limit the scope of names declared within them. This affects visibility and accessibility of types.

Namespace Hierarchy and Structure

1. **Dot Notation Hierarchy**: Namespaces use dot notation to create hierarchies (e.g., System.Collections.Generic). This doesn't imply inheritance - it's purely organizational.
2. **Recommended Conventions**: Microsoft guidelines suggest:
   - Starting with company name (e.g., Microsoft)

    - Then product name (e.g., Office)
    - Then functional area (e.g., Interop)

3. **Physical vs Logical Organization**: While namespaces suggest a physical organization, they don't enforce it. A single assembly can contain multiple namespaces, and a single namespace can span multiple assemblies.

Relationship with Assemblies

1. Many-to-Many Relationship

There's no strict 1:1 relationship between namespaces and assemblies. The System.Data.dll assembly contains multiple namespaces (System.Data, System.Data.Common, System.Data.SqlClient etc.)

2. Common Patterns

- Core functionality in root namespace (System.Data)
- Specialized functionality in child namespaces (System.Data.SqlClient)

3. Assembly Naming Conventions

Assemblies often take the name of their primary namespace (System.Data.dll for System.Data), but this isn't mandatory.

Using Namespaces in Code

1. The 'using' Directive

The 'using' keyword serves two purposes:

- As a directive to import namespaces
- As a statement for resource cleanup

2. Aliasing

Allows resolving naming conflicts or shortening long namespaces:

```
using Excel = Microsoft.Office.Interop.Excel;
```

3. Global Using (C# 10)

New in C# 10, global using directives apply to the entire project.

```
// GlobalUsings.cs
global using System;
global using System.IO;

// All other .cs files in the project
// No need to include using System; or using System.IO;
```

Namespace Design Guidelines

1. **Logical Grouping**: Group types that are commonly used together and change together.
2. **Depth Consideration**: While deep hierarchies are possible, 2-4 levels are typically most usable.
3. **Versioning Considerations**: Namespaces should remain stable across versions to avoid breaking changes.

Advanced Namespace Features

1. **File-Scoped Namespaces (C# 10)**: Reduces indentation by declaring namespace scope for the entire file.

```
namespace MyNamespace;

class MyClass {
    // ... code ...
}
```

2. **Implicit Global Usings**: .NET 6+ projects can automatically include common namespaces.
   - Frequently used namespaces like System, System.Collections.Generic, and System.Linq are implicitly imported.
   - You can now use types from System.Collections.Generic (like IEnumerable) without writing `using System.Collections.Generic;` at the top of your file.

MSDN References

- Namespaces (C# Programming Guide)
- Global Using Directive

Data Types

1. Overview of Data Types C#

1.1 Categories of Data Types

Category	Description	Examples
Value Types	Stored on the stack (direct value).	`int`, `float`, `struct`, `enum`
Reference Types	Stored on the heap (accessed via reference).	`class`, `string`, `delegate`, `array`

1.2 Key Differences

Aspect	Value Types	Reference Types
Memory Location	Stack	Heap
Assignment	Copy entire value	Copy reference (pointer)

Aspect	Value Types	Reference Types
Default Value	`0`, `false`, etc.	`null`
Performance	Faster access	Slightly slower (indirection)

2. Value Types

2.1 C# - Built-in Primitive Types

Type	Size (Bytes)	Range	.NET Alias
`byte`	1	0 to 255	`System.Byte`
`short`	2	-32,768 to 32,767	`System.Int16`
`int`	4	-2.1B to 2.1B	`System.Int32`
`long`	8	-9.2Q to 9.2Q	`System.Int64`
`float`	4	$\pm1.5\times10^{-45}$ to $\pm3.4\times10^{38}$	`System.Single`
`double`	8	$\pm5.0\times10^{-324}$ to $\pm1.7\times10^{308}$	`System.Double`
`decimal`	16	$\pm1.0\times10^{-28}$ to $\pm7.9\times10^{28}$	`System.Decimal`
`bool`	1	`true` or `false`	`System.Boolean`
`char`	2	Unicode (U+0000 to U+FFFF)	`System.Char`

2.2 Type Conversions

1. **Implicit Conversion (Widening)**
   - **Compiler-automated** when no data loss is possible.
   - **Allowed for numeric types** if the target type has a larger range.

- **Examples**

```
int numInt = 100;
long numLong = numInt;  // Implicit (int → long)

float price = 10.5f;
double total = price;   // Implicit (float → double)
```

2. **Supported Implicit Conversions**

From	To
`byte`	`short`, `int`, `long`, `float`, `double`
`int`	`long`, `float`, `double`
`char`	`int`, `long`, `float`, `double`

3. **Explicit Conversions (Narrowing)**

- **Manual** using `(type)` syntax.
- **May cause data loss** (e.g., truncation, overflow).
- **3.2 Examples**

```
double pi = 3.14159;
int intPi = (int)pi;  // Explicit (double → int), truncates to 3

long bigNum = 1_000_000_000;
int smallerNum = (int)bigNum;  // Works if within int range
```

2.3 Conversion Methods

1. `Convert` **Class**
   - Handles `null`, special cases, and cultural formats.

   ```
   string input = "123";
   int number = Convert.ToInt32(input);  // Throws FormatException if invalid
   ```

2. `Parse` **vs.** `TryParse`

```
| **Method** | **Behavior**                         | **Usage**                            |
| ---------- | ------------------------------------ | ------------------------------------ |
| `Parse`    | Throws exception on failure          | `int.Parse("42")`                    |
| `TryParse` | Returns `bool` (safe for user input) | `int.TryParse("42", out var result)` |
```

   - Example

   ```
   if (int.TryParse(Console.ReadLine(), out int age))
   {
       Console.WriteLine($"Age: {age}");
   }
   else
   {
       Console.WriteLine("Invalid input!");
   }
   ```

2.4 Handling Overflow

- Use `checked` to throw exceptions on overflow.

```
checked
{
    int max = int.MaxValue;
    int overflow = (int)(max + 1);  // throws OverflowException
}
```

- The default behavior is `unchecked` i.e. data overflows to opposite sign (positive to negative).

2.5 User-defined Value Types (`struct` and `enum`)

```
public struct Point
{
    public int X;
    public int Y;
}
```

```
public enum LogLevel { Info, Warning, Error }
// by default, underlying type is "int"
```

3. Reference Types

3.1 Built-in Reference Types

Type	Description	Example
`string`	Immutable Unicode text	`string s = "Hello";`

Type	Description	Example
`object`	Base type for all .NET objects	`object o = 10;`
`Array`	Fixed-size collection	`int[] nums = { 1, 2, 3 };`
`Delegate`	Function pointer	`Action<int> print = Console.WriteLine;`

3.2 Custom Reference Types (`class` and `interface`)

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

✔ **Use cases**: Complex objects with behavior.

4. Special Types

4.1 `System.Object` (Root Type)

- All types inherit from `object`.
- Methods:

  ```
  Equals(), GetHashCode(), ToString(), GetType()
  ```

4.2 `System.ValueType` (Base for Value Types)

- Overrides `Equals()` and `GetHashCode()` for value comparison.

4.3 `System.Void` (For Methods with No Return)

- Used in reflection (`MethodInfo.ReturnType`).

5. Best Practices

✔ **Prefer `int` over `Int32`** (language aliases are idiomatic).
✔ **Use `struct` for small, immutable data**.
✔ **Avoid large value types** (copy overhead).

6. MSDN References

- Built-In Types (C#)
- Value Types vs. Reference Types

Boxing and Unboxing

Introduction

Boxing and unboxing have been fundamental concepts in C# since its initial release (2002) as part of the .NET Framework's type system. These mechanisms were designed to bridge the gap between value types and reference types, enabling unified type handling through the object-oriented paradigm.

Definition and Core Concepts

1. Boxing Definition

Boxing is the process of converting a value type to a reference type by wrapping the value inside a reeference type like System.Object. This operation:

- Creates a new object on the managed heap
- Copies the value type's value into this object
- Returns a reference to the new object

```
int num1 = 123;
object obj = num1; // boxing
```

2. Unboxing Definition

Unboxing is the reverse operation that extracts the value type from the object. This requires:

- Checking that the object instance is a boxed value of the correct type
- Copying the value from the heap back to the stack

```
int num2 = obj; // unboxing
```

- Unboxing requires exact type match:

```
object boxed = 42;
long value = (long)boxed;  // Runtime error
```

3. Type System Integration

These operations enable value types to participate in:

- Polymorphic behavior through object references
- Collections that store objects (like ArrayList)
- Reflection and late binding scenarios

Performance Implications

1. Memory Impact

- Boxing allocates memory on the heap (minimum 16 bytes for object header + value)
- Creates garbage collection pressure
- Unboxing doesn't allocate but requires type checking

2. CPU Overhead

- Boxing involves memory allocation and copying
- Unboxing requires type checking and copying
- Both operations are orders of magnitude slower than direct value type operations

3. Hidden (Implicit) Boxing

- Value types in non-generic collections
- Calling GetType() on value types
- String concatenation with value types

Common Usage Patterns

1. **Legacy Collections**: Pre-generics (pre-.NET 2.0) collections like ArrayList required boxing:

```
ArrayList list = new ArrayList();
list.Add(42);  // Boxing occurs
int value = (int)list[0];  // Unboxing occurs
```

2. **Interface Implementation**: Value types implementing interfaces get boxed when cast to the interface type:

```
IComparable comparable = 5;  // Boxing occurs
```

3. **Reflection Scenarios**: Methods like MethodInfo.Invoke() box value type parameters

Modern Alternatives

1. **Generics (.NET 2.0+)**: Generic collections (List) eliminate boxing by working with value types directly
2. **Span (.NET Core 2.1+)**: Provides stack-only, boxing-free access to memory

MSDN References

- Boxing and Unboxing (C# Programming Guide)
- Performance Considerations
- Type System Fundamentals

C# Methods

1. Methods in C#

1.1 Definition

- A **method** is a block of code that performs a specific task and can be reused.
- Can return a value (`void` if no return).

1.2 Syntax

```
// Method with parameters and return type
public int Add(int a, int b) {
    return a + b;
}

// void method (no return)
public void Log(string message) {
    Console.WriteLine(message);
}
```

1.3 Method Components

Part	Example	Purpose
Access Modifier	`public`, `private`	Controls visibility.
Return Type	`int`, `void`	Specifies what the method returns.
Parameters	`(int a, int b)`	Inputs to the method.

2. Method Parameters

2.1 Named and Optional Arguments

- Named arguments enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list.
- The arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

```
void PrintInfo(string name, int age, string addr, string email) {
    Console.WriteLine($"Name: {name}, Age: {age}, Addr: {addr}, Email: {email}");
}

// positional args - args must be passed in sequence
PrintInfo("James Bond", 65, "London", "james@bond.com");

// named args - args can be passed in any order
PrintInfo(age: 65, email: "james@bond.com", name: "James Bond", addr: "London");

// mixed args - combination of args is possible - positional args work as long as their positions are correct
PrintInfo(name:"James Bond", 65, email:"james@bond.com", addr: "London");

// ERROR: Named argument 'email' is used out-of-position but is followed by an unnamed argument
PrintInfo(name: "James Bond", email: "james@bond.com", 65, addr: "London");
```

- Optional arguments enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.
- Optional parameters must appear after all required parameters (like C++).

```
void PrintInfo(string name, int age, string addr = "Anywhere", string email="Unknown")
{
    Console.WriteLine($"Name: {name}, Age: {age}, Addr: {addr}, Email: {email}");
}

// Usage
PrintInfo("James Bond", 65);
PrintInfo(age:65, name:"James Bond");
```

2.3 Parameter Types

Type	Syntax	Behavior
Value	`int a`	Copy of the value passed.
Reference (`ref`)	`ref int a`	Directly modifies the original variable.
Output (`out`)	`out int result`	Assigns a value before returning.
Input (`in`)	`in Vector3 pos`	Read-only reference (performance optimization).

1. `ref` **Parameter**

```
public void Swap(ref int x, ref int y) {
    int temp = x;
    x = y;
    y = temp;
}

// Usage:
```

```
int a = 5, b = 10;
Swap(ref a, ref b);  // a=10, b=5
```

2. `out` **Parameter**

```
public void Multiply(int x, int y, out int result) {
    result = x * y;
}

// Usage:
int num1=22, num2=7, res;
Multiply(num1, num2, out res);
```

3. `in` **Parameter**

```
public double Calculate(in Vector3 point) {
    return point.X + point.Y;  // Cannot modify 'point'
}
```

3. Local Functions (C# 7.0+)

3.1 Definition

- **Nested methods** inside another method.
- Useful for **helper logic** that shouldn't be exposed.

3.2 Syntax

```
public void ProcessData(List<int> data) {
    // Local function
    int Square(int x) {
        return x * x;
    }

    int data = {1, 2, 3, 4};
    foreach (int num in data)
        Console.WriteLine(Square(num));
}
```

- **Access outer method variables**:

```
public void PrintCount() {
    int count = 0;

    void Increment() {
        count++;  // Modifies 'count'
    }
    Increment();
    Console.WriteLine(count);  // 1
}
```

4. Static Local Functions (C# 8.0+)

4.1 Purpose

- **Prevents accidental captures** of outer variables (improves performance).
- Explicitly declares no dependency on enclosing scope.

4.2 Example

```
public void Calculate() {
    int baseValue = 10;

    // Static local function (cannot access 'baseValue')
    static int Scale(int value, int factor) {
        return value * factor;
    }

    Console.WriteLine(Scale(baseValue, 2));  // 20
}
```

4.3 When to Use

- **Pure functions** (no side effects).
- **Performance-critical code** (avoid closure allocations).

5. Best Practices

- **Use `out` for multiple returns** (instead of tuples in simple cases).
- **Prefer local functions** over private methods for one-off helpers.
- **Use `static` locals** to avoid unintended closures.

6. MSDN References

- Methods (C#)
- Local Functions
- Parameter Modifiers

User-Defined Value Types

1. Introduction to User-Defined Value Types

1.1 Definition

- **Value types** store data directly (stack-allocated).
- **Two custom flavors**:
  - `struct`: Composite data type (e.g., `Point`, `DateTime`).
  - `enum`: Named constant set (e.g., `LogLevel`, `HttpStatusCode`).

1.2 Key Characteristics

Feature	Struct	Enum
Memory	Stack (unless boxed)	Stack (underlying integer)
Default	All fields zeroed	First member (0 by default)
Inheritance	No (sealed implicitly)	No
Use Case	Small, immutable data	Finite named options

2. Structs (Custom Value Types)

2.1 Definition & Syntax

```
public struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
        this.X = x;
        this.Y = y;
    }
}
```

2.2 When to Use Structs

✔ **Small size** (< 16 bytes recommended).
✔ **Frequently copied** (e.g., coordinates, RGB colors).
✔ **Immutable patterns** (`readonly struct` have only getters for properties).

2.3 When to Avoid Structs

✖ **Large data** (copy overhead).
✖ **Polymorphism needed** (use `class` instead).

2.4 Performance Implications

- **Stack allocation** → Faster than heap (`class`).
- **No garbage collection** → Reduced GC pressure.

3. Struct vs. Class Decision Table

Criteria	Choose `struct`	Choose `class`
Size	Small (< 16B)	Large
Semantics	Value equality	Reference identity
Allocation	Stack	Heap
Mutation	Immutable preferred	Mutable

4. Advanced Struct Features

- Refer MSDN for more details.

1. **Ref Structs (Stack-Only)**

```csharp
public ref struct StackOnlyStruct { ... }
```

2. `readonly struct` **(Immutable by Design)**

```csharp
public readonly struct ImmutablePoint
{
    public int X { get; }  // No setters allowed
}
```

3. `record struct` **(C# 10+)**

```csharp
public record struct Point(double X, double Y) {
     public double GetDistanceFromOrigin() {
         return Math.Sqrt(X * X + Y * Y);
     }
}
```

5. Enums (Named Constants)

5.1 Definition & Syntax

~~~
```csharp
public enum LogLevel // Underlying type (default: int)
{
    Info = 1,    // Explicit value
    Warning,     // 2 (auto-incremented)
    Error = 10   // Custom value
~~~

```
}
```

```
```
```

5.2 Key Features

✔ **Type-safe**: Compiler prevents invalid values.
✔ **Bit flags support** (`[Flags]` attribute).
✔ **Underlying type control** (`byte`, `short`, etc.).

5.3 Enum Operations

```
// Parsing
LogLevel level = Enum.Parse<LogLevel>("Warning");

// Flags (bitwise)
[Flags]
public enum Permissions
{
    Read = 1,
    Write = 2,
    Execute = 4
}
var perms = Permissions.Read | Permissions.Write;
```

6. MSDN References

- Structs (C#)
- Enums (C#)

Inheritance & Polymorphism
```

## 1. Inheritance: The "is-a" Relationship

### 1.1 Definition

- **Inheritance** allows a class (`derived class`) to inherit fields/methods from another (`base class`).
- Models hierarchical relationships (e.g., `Dog : Animal`).

### 1.2 Syntax

```
public class Animal  // Base class
{
    public string Name { get; set; }
    public void Eat() {
        Console.WriteLine($"{Name} is eating.");
    }
}

public class Dog : Animal // Derived class
{
    public void Bark() {
        Console.WriteLine($" {Name} sounds Woof!");
    }
}
```

1.3 Key Rules

✔ **Single Inheritance**: A class can inherit from **only one** base class.
✔ **Transitivity**: If `C : B` and `B : A`, then `C` inherits from `A`.
✔ **Constructors**: Not inherited (but can be chained with `base()`).

1.4 Inheritance Types

1. **Single Inheritance**: One class inherited from one base class.

```
class Fruit { /* ... */ }
class Mango : Fruit { /* ... */ }
```

2. **Multiple Inheritance**: One class inherited from multiple base classes. Not supported in C# for the classes, but one class can implement multiple interfaces.

```
interface IPrintable { /* ... */ }
interface IFormattable { /* ... */ }
class Document : IPrintable, IFormattable { /* ... */}
```

3. **Hierarchial Inheritance**: Multiple classes inherited from single base class.

```
class Person { /* ... */ }
class Employee : Person { /* ... */ }
class Student : Person { /* ... */ }
```

4. **Multi-level Inheritance**: Multiple levels of inheritance.

```
class Person { /* ... */ }
class Player : Person { /* ... */ }
class Cricketer : Player { /* ... */ }
```

5. **Hybrid Inheritance**: Any combination of the above inheritance types.

2. Polymorphism: Many Forms

2.1 Definition

- **Polymorphism** allows objects of different classes to be treated as objects of a common base class.
- Achieved via:
  - **Method overriding** (runtime polymorphism).
  - **Method overloading** (compile-time polymorphism).

2.2 Method Overriding (`virtual` + `override`)

```
public class Animal {
    public virtual void MakeSound() {
        Console.WriteLine("Some sound");
    }
}

public class Dog : Animal {
    public override void MakeSound() {
        Console.WriteLine("Woof!");
    }
}

// Usage:
Animal myDog = new Dog();
myDog.MakeSound();  // Output: "Woof!" (Runtime decision)
```

2.3 Method Overloading (Compile-Time)

```
public class Logger
{
    public void Log(string message) { ... }
    public void Log(int number) { ... }  // Same name, different params
}
```

3. Abstract Classes & Methods

3.1 Abstract Classes

- Cannot be instantiated.
- Define **partial implementations** for derived classes.

```
public abstract class Shape
{
    public abstract double Area();  // No implementation
}

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

3.2 Interfaces vs. Abstract Classes

Feature	Interface	Abstract Class
Inheritance	Multiple	Single
Methods	No implementation	Partial implementation
Fields	Not allowed	Allowed

4. The `sealed` Keyword

- Prevents further inheritance:

```
public sealed class UltimateClass { }  // Cannot be derived
public class Attempt : UltimateClass { }  // ❌ Compile error
```

- Can also seal individual methods:

```
public override sealed void Method() { }  // No further overrides
```

5. The `new` Keyword

- Hides base class method/property in derived class.

```
class Animal {
    public virtual void MakeSound() {
        Console.WriteLine("Generic animal sound");
    }
}

class Dog : Animal {
    public new void MakeSound() { //Hides the base class's MakeSound method
        Console.WriteLine("Woof!");
    }
    // method can also be "new virtual" in order to override it in Dog's sub-classes.
}
```

```
// In Main()
Animal myDogAsAnimal = new Dog();
```

```
myDogAsAnimal.MakeSound();
//Output: Generic animal sound (because myDogAsAnimal is an Animal reference and MakeSound() is "new" in Dog, not
"override").
```

6. Example: Payment System

```
public abstract class PaymentMethod
{
    /*Other members*/
    public abstract void ProcessPayment(double amount);
}

public class CreditCard : PaymentMethod
{
    public override void ProcessPayment(double amount) {
        Console.WriteLine($"Paid ${amount} via Credit Card");
    }
}

public class PayPal : PaymentMethod
{
    public override void ProcessPayment(double amount) {
        Console.WriteLine($"Paid ${amount} via PayPal");
    }
}

// Usage:
PaymentMethod payment = new CreditCard();
payment.ProcessPayment(100);  // Polymorphic call
```

7. Casting in Inheritance Hierarchies

5.1 Upcasting (Implicit)

- Treating a derived class as its base type (always safe).

```
Dog dog = new Dog();
Animal animal = dog;  // Upcast (Dog → Animal)
```

5.2 Downcasting (Explicit)

- Treating a base type as a derived type (requires check).

```
Animal animal = new Dog();

// Downcast (Animal → Dog) - Unsafe
Dog dog = (Dog)animal;

// Safer with `is` and casting:
if (animal is Dog) {
    Dog d = (Dog)animal;
    d.Bark();
}

// OR Shorthand typesafe downcasting:
if (animal is Dog d) {
    d.Bark();
}
```

5.3 `as` Operator (Safe Cast)

- Returns `null` (instead of throwing) if cast fails.

```
Animal animal = new Cat();
Dog dog = animal as Dog;  // Returns null (not a Dog)
```

7. Best Practices

- **Favor composition over inheritance** when possible.
- **Use `abstract`** for incomplete base implementations.
- **Avoid deep inheritance hierarchies** (>3 levels).
- **Use `is`/`as`** for safe downcasting.

8. MSDN References

- Inheritance (C#)
- Type Conversions (C#)
- Polymorphism (C#)