Table of Contents

Interfaces in C#

Definition

An **interface** is a contract that defines a set of method signatures, properties, events, or indexers that a class or struct must implement. Interfaces contain no implementation details - they only define **what** should be done, not **how** it should be done.

Key Characteristics of Interfaces

Important Features

1. **No Implementation**: Interfaces cannot contain implementation code (though C# 8.0+ allows default interface methods with private access)
2. **Multiple Inheritance**: A class can implement multiple interfaces (solving the diamond problem)
3. **Contract Definition**: Acts as a blueprint that implementing classes must follow
4. **Polymorphism**: Enables polymorphic behavior through interface references
5. **No Access Modifiers**: Interface members are implicitly public
6. **No Fields**: Interfaces cannot contain instance fields (only properties)

Syntax

```
interface IStudent
{
    void Register();
    // Private methods with implementation (C# 8.0+)
    private void Show()
    {
        Console.WriteLine("ok");
    }
}
```

Why Use Interfaces?

1. **Abstraction**: Hide implementation details and expose only necessary functionality
2. **Loose Coupling**: Reduce dependencies between components
3. **Testability**: Easy to mock for unit testing
4. **Flexibility**: Change implementations without affecting client code
5. **Design by Contract**: Ensure classes follow specific contracts

Interface vs Abstract Class

Feature	Interface	Abstract Class
Multiple Inheritance	Yes	No
Implementation	No (except default methods)	Yes
Fields	No	Yes
Constructors	No	Yes
Access Modifiers	Public only	Any
When to Use	"Can do" relationship	"Is a" relationship

Multiple Interface Implementation

Concept

A single class can implement multiple interfaces simultaneously, inheriting all their contracts. This is C#'s solution to multiple inheritance while avoiding the diamond problem.

Syntax

```
class ABC : TypeA, IPlayer, IStudent
{
    // Must implement all members from IPlayer and IStudent
}
```

Inheritance Order Rules

1. **Base Class First**: If inheriting from a class, it must come first
2. **Interfaces After**: All interfaces follow the base class
3. **Comma Separation**: Multiple interfaces separated by commas

Example from Code

```
class abc : typea, iplayer, istudent
{
    // Implements methods from both IPlayer and IStudent
    public void TakePosition(int position) { }
    public int ShowScore() { }
    public void Register() { }
}
```

Benefits

1. **Flexibility**: Object can play multiple roles
2. **Polymorphism**: Can be referenced by any implemented interface type
3. **Composition over Inheritance**: Build complex behaviors from simple contracts
4. **Avoid Diamond Problem**: No ambiguity in method resolution

Real-World Scenario

A `StudentAthlete` class might implement both `IStudent` (for academic operations) and `IAthlete` (for sports operations), representing a person who fulfills both roles.

Explicit Interface Implementation

What is Explicit Interface Implementation?

When a class implements multiple interfaces that have methods with the same signature, **explicit interface implementation** allows you to provide separate implementations for each interface.

Problem It Solves

```
interface IPlayer
{
    void Register();
}

interface IStudent
{
    void Register();
}

// Both have Register() - which one do we implement?
```

Syntax

```
class ABC : IPlayer, IStudent
{
    // Explicit implementation for IPlayer
    void IPlayer.Register()
    {
        Console.WriteLine("Register as player");
    }

    // Explicit implementation for IStudent
    void IStudent.Register()
    {
        Console.WriteLine("Register as student");
    }
}
```

Key Characteristics

1. **No Access Modifier**: Explicit implementations cannot have access modifiers
2. **No Public Keyword**: They are implicitly private to the class
3. **Interface Qualification**: Must prefix method name with interface name
4. **Access Through Interface**: Can only be called through interface reference

Usage

```
ABC obj = new ABC();
// obj.Register(); // ERROR - which Register?
```

```csharp
IPlayer player = obj;
player.Register(); // Calls IPlayer.Register()

IStudent student = obj;
student.Register(); // Calls IStudent.Register()
```

Implicit vs Explicit Implementation

Implicit Implementation (Public):

```csharp
public void Register()
{
    Console.WriteLine("Register");
}
```

- Accessible through class instance
- Satisfies all interfaces with this signature

Explicit Implementation:

```csharp
void IPlayer.Register()
{
    Console.WriteLine("Player registration");
}
```

- Only accessible through interface reference
- Provides specific implementation per interface

When to Use Explicit Implementation

1. **Name Conflicts**: Multiple interfaces have same method signature
2. **Hide Implementation**: Don't want method accessible from class instance
3. **Different Behaviors**: Need different logic for different interface contexts
4. **Interface Segregation**: Keep interface-specific logic separate

---

IComparable Interface

ⓘ **Overview**

`IComparable` and `IComparable<T>` are built-in .NET interfaces that enable objects to define their natural ordering, allowing them to be sorted.

Two Versions

Non-Generic Version

IComparable - From System namespace

- Takes `object` parameter
- Requires type casting
- Less type-safe

Generic Version (Recommended)

IComparable<T> - Type-safe version

- Takes `T` parameter
- No casting needed
- Compile-time type checking

Interface Definition

Non-Generic Version:

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

Generic Version (Type-Safe):

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

CompareTo Return Values

The `CompareTo` method returns an integer that indicates the relationship between objects:

Return Value	Meaning	Example
Negative (typically -1)	Current instance is **less than** the parameter	`5.CompareTo(10)` returns -1
Zero (0)	Current instance **equals** the parameter	`5.CompareTo(5)` returns 0
Positive (typically 1)	Current instance is **greater than** the parameter	`10.CompareTo(5)` returns 1

Visual Representation:

```
this < other  →  return -1 (or negative)
this = other  →  return  0
this > other  →  return  1 (or positive)
```

Implementation Example

```
public class Student : IComparable<Student>
{
    public int Age { get; set; }

    public int CompareTo(Student other)
    {
        // Compare by age
        return Age.CompareTo(other.Age);

        // Manual implementation:
        // if (Age < other.Age) return -1;
        // if (Age > other.Age) return 1;
        // return 0;
    }
}
```

Using Built-in CompareTo

Most primitive types (int, string, DateTime, etc.) already implement IComparable:

```
return Age.CompareTo(other.Age);
```

This delegates the comparison logic to the int type's built-in comparison.

Sorting with IComparable

```
Student[] students = new Student[5] { /* ... */ };
Array.Sort(students); // Uses CompareTo method automatically
```

Multiple Sorting Criteria

```
public int CompareTo(Student other)
{
    // Sort by Age first
    int ageComparison = Age.CompareTo(other.Age);
    if (ageComparison != 0)
        return ageComparison;

    // If ages equal, sort by Name
    return Name.CompareTo(other.Name);
}
```

IComparer vs IComparable

- **IComparable**: Defines the **default/natural** ordering within the class itself
- **IComparer**: Defines **alternative** sorting logic externally

Best Practices

1. **Consistency**: Ensure CompareTo is consistent with Equals()
2. **Null Handling**: Check for null parameters in non-generic version
3. **Performance**: Keep comparison logic efficient for large collections
4. **Documentation**: Document the ordering criteria clearly

IDisposable Interface and Resource Management

The Problem: Unmanaged Resources

.NET has automatic memory management through **Garbage Collection (GC)**, but some resources are **unmanaged** and require explicit cleanup:

- **File Handles** (FileStream)
- **Database Connections** (SqlConnection)
- **Network Sockets**

- **Graphics Handles** (Bitmap, Pen)
- **Native Memory** (allocated through P/Invoke)

Why it matters: These resources are limited and must be released promptly to avoid leaks!

What is IDisposable?

Definition

`IDisposable` is an interface that provides a mechanism for releasing unmanaged resources **deterministically** (immediately), rather than waiting for the garbage collector.

Interface Definition

```
public interface IDisposable
{
    void Dispose();
}
```

Implementation Pattern

```
public class Student : IDisposable
{
    private FileStream txt; // Unmanaged resource
    private bool disposed = false; // Track disposal state

    public Student()
    {
        txt = new FileStream("txt.txt", FileMode.OpenOrCreate);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this); // Tell GC not to call finalizer
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
```

```
        {
            if (disposing)
            {
                // Dispose managed resources
                txt?.Dispose();
            }

            // Free unmanaged resources here (if any)

            disposed = true;
        }
    }

    ~Student() // Finalizer
    {
        Dispose(false);
    }
}
```

The Dispose Pattern (Full Implementation)

This is the **recommended pattern** for implementing IDisposable:

```
public class ResourceHolder : IDisposable
{
    private FileStream managedResource;
    private IntPtr unmanagedResource; // Native resource
    private bool disposed = false;

    // Public Dispose method
    public void Dispose()
    {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }

    // Protected Dispose method
    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose managed resources
                managedResource?.Dispose();
```

```csharp
            }

            // Free unmanaged resources
            if (unmanagedResource != IntPtr.Zero)
            {
                // Free native memory
                Marshal.FreeHGlobal(unmanagedResource);
                unmanagedResource = IntPtr.Zero;
            }

            disposed = true;
        }
    }

    // Finalizer (destructor)
    ~ResourceHolder()
    {
        Dispose(disposing: false);
    }
}
```

Key Components Explained

1. Dispose(bool disposing) Parameter

- **disposing = true**: Called from `Dispose()` method (explicit cleanup)
  - Safe to dispose both managed and unmanaged resources
- **disposing = false**: Called from finalizer (GC cleanup)
  - Only free unmanaged resources
  - Don't touch managed objects (they may already be collected)

2. GC.SuppressFinalize(this)

```csharp
GC.SuppressFinalize(this);
```

⚠ **Critical Performance Tip**

This tells the garbage collector: **"Don't call the finalizer for this object because I've already cleaned up."**

Why it's important:

- Finalizers are expensive (performance cost)

- If Dispose() already cleaned up, finalizer is unnecessary
- Removes object from finalization queue
- **Prevents two GC cycles** for object collection

3. Disposed Flag

```
private bool disposed = false;
```

Prevents multiple disposal attempts, which could cause errors.

Usage Patterns

Manual Disposal

```
Student s = new Student();
try
{
    // Use the object
}
finally
{
    s.Dispose(); // Explicit cleanup
}
```

Using Statement (Recommended)

```
using (Student s = new Student())
{
    Console.WriteLine(s.Id);
} // Dispose() automatically called here
```

Using Declaration (C# 8.0+)

```
using Student s = new Student();
Console.WriteLine(s.Id);
// Dispose() called at end of scope
```

Why IDisposable Matters

1. **Timely Cleanup**: Don't wait for GC (which is non-deterministic)
2. **Resource Leaks**: Prevent file locks, connection exhaustion, memory leaks
3. **Performance**: Release resources immediately when done
4. **Predictability**: Control exactly when cleanup happens

Common Mistakes

❌ **Common Errors to Avoid**

1. **Forgetting to Call Dispose**: Resource leak

   ```
   var stream = new FileStream("file.txt", FileMode.Open);
   // Forgot to dispose - file handle remains open!
   ```

2. **Using After Disposal**: ObjectDisposedException

   ```
   stream.Dispose();
   stream.Read(buffer, 0, 100); // CRASH!
   ```

3. **Not Implementing Finalizer**: Unmanaged resources may never be freed
4. **Disposing Twice**: Use disposed flag to prevent errors

   ```
   stream.Dispose();
   stream.Dispose(); // Could cause error without proper flag
   ```

Finalizers (Destructors)

What is a Finalizer?

A **finalizer** (also called destructor in C#) is a special method that is automatically called by the **Garbage Collector** before an object is destroyed and its memory is reclaimed.

Syntax

```
class Student
{
    ~Student() // Finalizer
    {
        // Cleanup code
        Console.WriteLine("Finalizer called");
```

```
    }
}
```

Key Characteristics

1. **Automatic Invocation**: Called by GC, not by you
2. **Non-Deterministic**: You don't know when it will run
3. **No Parameters**: Cannot accept parameters
4. **No Access Modifiers**: Cannot be public, private, etc.
5. **No Manual Call**: Cannot be called explicitly
6. **One Per Class**: Only one finalizer per class
7. **No Inheritance**: Not inherited by derived classes

How Finalizers Work

1. Object becomes unreachable (no references)
2. GC marks object for finalization
3. Object moved to **finalization queue**
4. Separate **finalizer thread** runs finalizers
5. After finalizer runs, object eligible for collection on next GC cycle

Performance Impact

Finalizers are **expensive**:

1. **Two GC Cycles**: Object survives first collection, removed in second
2. **Separate Thread**: Finalizer thread must run
3. **Queue Management**: Overhead of finalization queue
4. **Delayed Collection**: Object stays in memory longer

The Finalizer Queue

```
Generation 0 → Finalization Queue → Generation 1 → Collected
    ↑                ↓
 Object         Finalizer Runs
 Created
```

Finalizer vs Dispose

Aspect	Finalizer	Dispose
Call Time	Non-deterministic	Deterministic
Performance	Expensive	Fast
Called By	Garbage Collector	Developer
Purpose	Safety net	Primary cleanup
Managed Resources	DON'T dispose	Dispose

Example from Code

```
~Student()
{
    txt.Dispose(); // Release file handle
    Console.WriteLine("Finalizer");
}
```

Best Practices

1. **Avoid if Possible**: Only use if you have unmanaged resources
2. **Use with IDisposable**: Finalizer as backup for Dispose
3. **Don't Touch Managed Objects**: They may already be collected
4. **Keep It Simple**: Minimal logic in finalizers
5. **No Exceptions**: Never throw exceptions from finalizers

When Finalizers Run

Finalizers run when:

- GC determines object is unreachable
- Application domain unloads
- Application exits (usually, but not guaranteed)

Important: GC may never run before application exits, so finalizers may never execute!

Suppressing Finalization

```
public void Dispose()
{
    Cleanup();
```

```
    GC.SuppressFinalize(this); // Skip finalizer
}
```

This removes the object from the finalization queue, improving performance.

Garbage Collection

What is Garbage Collection?

Garbage Collection (GC) is .NET's automatic memory management system that identifies and reclaims memory occupied by objects that are no longer in use.

How GC Works

1. Managed Heap

All reference types are allocated on the **managed heap**:

```
[Object A] [Object B] [Object C] [Free Space...]
```

2. Reachability Analysis

GC determines if objects are "reachable" from **root references**:

Roots include:

- Local variables (stack)
- Static variables
- CPU registers
- GC handles

3. Mark Phase

```
1. Start from roots
2. Mark all reachable objects
3. Unmarked objects = garbage
```

4. Compact Phase

```
Before:  [A] [X] [B] [X] [C] [Free...]
```

```
After:   [A] [B] [C] [Free Space...]
```

GC compacts memory, moving objects together and updating references.

Generational Garbage Collection

.NET uses **generational GC** for performance optimization:

Generation 0 (Gen 0) - The Nursery

- **Youngest objects** (newly created)
- Recently allocated
- **Most frequent collections** (happens often)
- Most objects die young (Short-lived objects)
- **Fastest to collect**

🔄 Collection Frequency: **Very High**

Generation 1 (Gen 1) - The Buffer

- **Middle generation**
- Survived one Gen 0 collection
- **Buffer between Gen 0 and Gen 2**
- Medium lifespan objects

🔄 Collection Frequency: **Medium**

Generation 2 (Gen 2) - The Elders

- **Oldest objects** (long-lived)
- Survived multiple collections
- **Least frequent collections**
- Most expensive to collect
- Static objects, cached data

🔄 Collection Frequency: **Low** (but costly!)

Visual Flow:

```
New Object → Gen 0 → (Survives) → Gen 1 → (Survives) → Gen 2
                ↓          ↓                      ↓
         (Collected) (Collected)            (Long-lived)
```

Collection Triggers

GC runs when:

1. **Gen 0 full**: Most common trigger
2. **Memory pressure**: System low on memory
3. **Explicit call**: `GC.Collect()` (not recommended)
4. **Large object allocation**: Special large object heap (LOH)

GC Methods

```
// Force collection (avoid in production!)
GC.Collect();

// Collect specific generation
GC.Collect(0); // Gen 0 only

// Wait for pending finalizers
GC.WaitForPendingFinalizers();

// Get object generation
int gen = GC.GetGeneration(myObject);

// Suppress finalization
GC.SuppressFinalize(this);

// Get total memory
long memory = GC.GetTotalMemory(false);
```

GC.SuppressFinalize Explained

```
public void Dispose()
{
    txt?.Dispose();
    Console.WriteLine("Dispose called");
    GC.SuppressFinalize(this); // Important!
}
```

What it does:

1. Removes object from finalization queue
2. Prevents finalizer from running
3. Improves performance (no second GC cycle)
4. Use when Dispose() already cleaned up everything

Memory Generations Example

```
class Example
{
    static List<object> static_list = new List<object>(); // Gen 2

    void Method()
    {
        object temp = new object(); // Gen 0
        static_list.Add(temp); // Now will survive to Gen 1, then Gen 2

        object temp2 = new object(); // Gen 0
        // temp2 not referenced, will be collected
    }
}
```

Performance Considerations

✓ **Good Practices** ✅

- [x] ~~**Minimize allocations**: Reuse objects when possible~~
- [x] ~~**Avoid Gen 2 collections**: They're expensive~~
- [x] ~~**Use object pooling**: For frequently created objects~~
- [x] ~~**Dispose properly**: Release unmanaged resources~~
- [x] ~~**Avoid GC.Collect()**: Let GC decide when to collect~~

✕ **Bad Practices** ❌

- [ ] **Excessive small allocations**
- [ ] **Holding references unnecessarily**
- [ ] **Large object allocations** (>85KB go to LOH)
- [ ] **Forcing collections with GC.Collect()**
- [ ] **Creating many short-lived objects in loops**

Large Object Heap (LOH)

Objects **> 85,000 bytes** go to special heap:

- Not compacted (by default)
- Collected only with Gen 2
- Can cause fragmentation

Monitoring GC

```
// Get collection count
int gen0Collections = GC.CollectionCount(0);
int gen1Collections = GC.CollectionCount(1);
int gen2Collections = GC.CollectionCount(2);

Console.WriteLine($"Gen 0: {gen0Collections}");
Console.WriteLine($"Gen 1: {gen1Collections}");
Console.WriteLine($"Gen 2: {gen2Collections}");
```

Using Statement

What is the Using Statement?

The `using` statement provides a convenient syntax for **automatically calling Dispose()** on objects that implement `IDisposable`.

Syntax

```
using (Student s = new Student())
{
    Console.WriteLine(s.Id);
    // Use the object
} // Dispose() automatically called here
```

What It Actually Does

> **Syntactic Sugar Revealed**
>
> The using statement is **syntactic sugar** that the compiler automatically expands to try-finally block:

What you write:

```
using (Student s = new Student())
{
    Console.WriteLine(s.Id);
}
```

What the compiler generates:

```
Student s = new Student();
try
{
    Console.WriteLine(s.Id);
}
finally
{
    if (s != null)
    {
        ((IDisposable)s).Dispose();
    }
}
```

Key Features

1. **Automatic Disposal**: Dispose() called even if exception occurs
2. **Scope-Based**: Object disposed at end of using block
3. **Null-Safe**: Won't crash if object is null
4. **Exception-Safe**: Ensures cleanup in finally block
5. **IDisposable Required**: Only works with IDisposable types

Multiple Objects

Separate Statements

```
using (FileStream fs1 = new FileStream("file1.txt", FileMode.Open))
using (FileStream fs2 = new FileStream("file2.txt", FileMode.Open))
{
    // Use both streams
} // Both disposed in reverse order
```

Same Type (C# 8.0+)

```csharp
using (FileStream fs1 = File.Open("file1.txt"),
       FileStream fs2 = File.Open("file2.txt"))
{
    // Use both
}
```

Using Declaration (C# 8.0+)

Simplified syntax without braces:

```csharp
void Method()
{
    using Student s = new Student();
    Console.WriteLine(s.Id);
    // ... more code
} // Dispose() called at end of method scope
```

Benefits:

- Less nesting
- Cleaner code
- Disposal at method end

Using with var

```csharp
using var stream = new FileStream("data.txt", FileMode.Open);
// Compiler infers type
```

When to Use

Use the using statement when:

1. Object implements IDisposable
2. You want automatic cleanup
3. Object is short-lived (scope-based lifetime)
4. Working with files, streams, connections, etc.

Don't use when:

1. Object doesn't implement IDisposable
2. Object needs to live beyond current scope
3. Object is cached or shared

Common Types Used with Using

```
// File I/O
using (FileStream fs = new FileStream("file.txt", FileMode.Open)) { }
using (StreamReader sr = new StreamReader("file.txt")) { }
using (StreamWriter sw = new StreamWriter("file.txt")) { }

// Database
using (SqlConnection conn = new SqlConnection(connectionString)) { }
using (SqlCommand cmd = new SqlCommand(query, conn)) { }

// Graphics
using (Bitmap bmp = new Bitmap(100, 100)) { }
using (Graphics g = Graphics.FromImage(bmp)) { }

// Network
using (WebClient client = new WebClient()) { }
using (HttpClient http = new HttpClient()) { }

// Custom disposable
using (Student s = new Student()) { }
```

Nested Using Statements

```
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        using (SqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())
            {
                // Process data
            }
        } // reader disposed
    } // cmd disposed
} // conn disposed
```

Exception Handling with Using

```
try
{
    using (FileStream fs = new FileStream("file.txt", FileMode.Open))
```

```
    {
        // May throw exception
        throw new Exception("Error!");
    } // Dispose() STILL CALLED
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Important: Even if exception occurs, Dispose() is always called!

Using vs Try-Finally

These are equivalent:

```
// Using statement
using (var resource = new MyResource())
{
    resource.DoWork();
}

// Equivalent try-finally
var resource = new MyResource();
try
{
    resource.DoWork();
}
finally
{
    resource?.Dispose();
}
```

The using statement is just cleaner syntax!

Scope Blocks

What is a Scope Block?

A **scope block** is a region of code enclosed by curly braces `{ }` that defines the **lifetime and visibility** of variables declared within it.

Basic Syntax

```
{
    // This is a scope block
    int x = 5;
    Console.WriteLine(x);
} // x no longer exists here
```

Variable Lifetime

Variables declared in a scope exist only within that scope:

```
{
    Student s = new Student();
    s.Id = 1;
} // s goes out of scope here

// s.Id = 2; // ERROR: 's' doesn't exist
```

Automatic Resource Cleanup

When variables go out of scope:

1. **Reference types**: Reference removed, object eligible for GC
2. **Value types**: Memory immediately reclaimed
3. **IDisposable types**: Should be disposed (manually or via using)

Example from Code

```
// Scope operator
{
    Student s = new Student();
    // s only exists within this block
}
// s is now out of scope and eligible for garbage collection
```

Nested Scopes

```
{
    int x = 10;
    {
        int y = 20;
        Console.WriteLine(x); // OK: x is in outer scope
        Console.WriteLine(y); // OK: y is in current scope
    }
```

```
    Console.WriteLine(x); // OK
    // Console.WriteLine(y); // ERROR: y out of scope
}
```

Scope Rules

1. Variable Shadowing Not Allowed

```
{
    int x = 5;
    {
        // int x = 10; // ERROR: Cannot redeclare 'x'
    }
}
```

2. Inner Scope Access

Inner scopes can access outer scope variables:

```
{
    int outer = 10;
    {
        int inner = outer * 2; // OK: Can access 'outer'
    }
}
```

3. Block-Level Scope

C# uses **block-level scoping**, not function-level:

```
void Method()
{
    if (true)
    {
        int x = 5;
    }
    // x doesn't exist here
}
```

Common Use Cases

1. Limiting Variable Lifetime

```
{
    byte[] largeBuffer = new byte[1000000];
    ProcessData(largeBuffer);
} // largeBuffer eligible for GC immediately
```

2. Organizing Code

```
void Process()
{
    // Phase 1
    {
        var tempData = LoadData();
        Validate(tempData);
    }

    // Phase 2
    {
        var results = Calculate();
        Save(results);
    }
}
```

3. Resource Management

```
{
    FileStream fs = new FileStream("file.txt", FileMode.Open);
    // Use fs
    fs.Dispose();
} // fs reference cleared
```

Scope vs Using Statement

```
// Manual scope
{
    Student s = new Student();
    try
    {
        // Use s
    }
    finally
    {
        s.Dispose();
    }
```

```
}

// Using statement (better)
using (Student s = new Student())
{
    // Use s
} // Automatically disposed
```

The using statement combines scope with automatic disposal!

Different Types of Scopes

1. Namespace Scope

```
namespace MyNamespace
{
    // Classes, interfaces, etc.
}
```

2. Class Scope

```
class MyClass
{
    private int field; // Class scope
}
```

3. Method Scope

```
void MyMethod()
{
    int local; // Method scope
}
```

4. Block Scope

```
{
    int x; // Block scope
}
```

5. Loop Scope

```
for (int i = 0; i < 10; i++)
{
    // i exists only in loop
}
```

Performance Considerations

Good Practice

```
{
    var largeObject = CreateLarge();
    UseObject(largeObject);
} // Object eligible for GC immediately
DoOtherWork(); // Memory potentially freed
```

Bad Practice

```
var largeObject = CreateLarge();
UseObject(largeObject);
DoOtherWork(); // Object still in memory
DoMoreWork();
DoEvenMoreWork();
// Object held until method ends
```

Scope and Memory Management

```
void Example()
{
    List<int> numbers = new List<int>();

    for (int i = 0; i < 1000; i++)
    {
        // Each iteration creates new object
        {
            var temp = new byte[1000];
            ProcessBytes(temp);
        } // temp eligible for GC after each iteration
    }
}
```

Properties in C#

What is a Property?

A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties use **accessor methods** (get and set) but are used like fields.

Why Use Properties?

1. **Encapsulation**: Hide internal implementation
2. **Validation**: Control what values can be set
3. **Computed Values**: Calculate values on-the-fly
4. **Read-Only/Write-Only**: Control access level
5. **Change Notification**: Trigger events when values change
6. **Future Flexibility**: Change implementation without breaking code

Basic Property Syntax

```
class Student
{
    private int age; // Backing field

    public int Age // Property
    {
        get { return age; }
        set { age = value; }
    }
}
```

Auto-Implemented Properties

When no additional logic is needed:

```
public int Id { get; set; }
public string Name { get; set; }
public int Age { get; set; }
```

The compiler automatically creates a hidden backing field.

Property Accessors

Get Accessor

Returns the property value:

```
public int Age
{
    get
    {
        return age;
    }
}
```

Set Accessor

Assigns a value using the implicit `value` parameter:

```
public int Age
{
    set
    {
        age = value; // 'value' is the incoming value
    }
}
```

Property Access Modifiers

Different Accessor Levels

```
public int Age
{
    get { return age; }
    private set { age = value; } // Private setter
}
```

Common Patterns:

- `public get; private set;` - Read-only from outside class
- `public get; protected set;` - Writable in derived classes
- `private get; public set;` - Write-only (rare)

Read-Only Properties

Method 1: No Setter

```
public int Age
{
    get { return age; }
    // No setter
}
```

Method 2: Private Setter

```
public int Age { get; private set; }
```

Method 3: Expression-Bodied (Computed)

```
public string FullName => $"{FirstName} {LastName}";
```

Validation in Properties

```
private int age;

public int Age
{
    get { return age; }
    set
    {
        if (value < 0 || value > 120)
            throw new ArgumentException("Invalid age");

        age = value;
    }
}
```

Expression-Bodied Properties (C# 6.0+)

Read-Only

```
public string FullName => $"{FirstName} {LastName}";
```

With Getter and Setter

```
private int age;

public int Age
```

```
{
    get => age;
    set => age = value < 0 ? 0 : value;
}
```

Init-Only Properties (C# 9.0+)

Can only be set during object initialization:

```
public class Student
{
    public int Id { get; init; }
    public string Name { get; init; }
}

// Usage
var student = new Student
{
    Id = 1,
    Name = "Ali"
};

// student.Id = 2; // ERROR: Can't modify after initialization
```

Computed Properties

Properties without backing fields:

```
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    // Computed property
    public int Area => Width * Height;

    // Computed property with logic
    public string Description
    {
        get
        {
            if (Width == Height)
                return "Square";
            return "Rectangle";
        }
```

```
    }
}
```

Properties vs Fields

Aspect	Field	Property
Access Control	Same for all	Different get/set
Validation	No	Yes
Interface	No	Yes
Inheritance	Not virtual	Can be virtual
Debugging	Harder	Easier
Binary Compatibility	Changes break	More flexible

Example from Code

```
private int x;

public int X
{
    get
    {
        return x;
    }
    set
    {
        x = value;
    }
}
```

This is a **full property** with explicit backing field. Usually, you'd use auto-implemented property:

```
public int X { get; set; }
```

Property Naming Conventions

```
private int age;        // Field: camelCase
public int Age { get; set; }  // Property: PascalCase
```

Rules:

- Fields: camelCase with optional underscore prefix (`_age`)
- Properties: PascalCase
- Property name typically matches field name (capitalized)

Virtual Properties

Properties can be virtual for polymorphism:

```
public class Animal
{
    public virtual string Sound => "Some sound";
}

public class Dog : Animal
{
    public override string Sound => "Bark";
}
```

Properties in Interfaces

Interfaces can declare properties:

```
interface IStudent
{
    int Id { get; set; }
    string Name { get; }  // Read-only in interface
}

class Student : IStudent
{
    public int Id { get; set; }
    public string Name { get; private set; }
}
```

Indexer Properties

Special properties accessed with index notation:

```
public class StudentCollection
{
    private Student[] students = new Student[100];

    // Indexer property
    public Student this[int index]
```

```
    {
        get { return students[index]; }
        set { students[index] = value; }
    }
}

// Usage
var collection = new StudentCollection();
collection[0] = new Student(); // Calls setter
Student s = collection[0];     // Calls getter
```

Lazy Initialization

```
private List<string> courses;

public List<string> Courses
{
    get
    {
        if (courses == null)
            courses = new List<string>();
        return courses;
    }
}
```

Or with null-coalescing:

```
public List<string> Courses => courses ??= new List<string>();
```

Best Practices

1. **Use Auto-Properties**: When no additional logic needed
2. **Validate in Setters**: Ensure data integrity
3. **Avoid Complex Logic**: Keep getters/setters simple
4. **Use Expression Bodies**: For simple computed properties
5. **Private Setters**: For read-only data from outside
6. **Init-Only**: For immutable data
7. **Notify Changes**: Implement INotifyPropertyChanged for UI binding

Property Pattern Matching (C# 8.0+)

```
public static string GetStatus(Student student) => student switch
{
    { Age: < 18 } => "Minor",
    { Age: >= 18 and < 65 } => "Adult",
    { Age: >= 65 } => "Senior",
    _ => "Unknown"
};
```

Summary

> 📋 **Key Takeaways**
>
> This code demonstrates several critical C# concepts for building robust applications:

Core Concepts Covered

Concept	Purpose	Key Benefit
🔌 **Interfaces**	Contracts for implementation	Abstraction & loose coupling
🔗 **Multiple Implementation**	Implementing several interfaces	Flexibility & composition
🎯 **Explicit Implementation**	Resolving method conflicts	Precise interface control
📊 **IComparable**	Defining object ordering	Automatic sorting capability
🧹 **IDisposable**	Managing resources	Deterministic cleanup
📋 **ICloneable**	Creating object copies	Deep/shallow copying
⚰️ **Finalizers**	GC cleanup mechanism	Safety net for resources
🗑️ **Garbage Collection**	Automatic memory management	No manual memory management
🔒 **Using Statement**	Automatic disposal	Exception-safe cleanup
📦 **Scope Blocks**	Controlling lifetime	Memory efficiency
🏷️ **Properties**	Controlled field access	Encapsulation & validation

> ✓ **Why This Matters**
>
> Understanding these concepts is **essential** for writing:

- ✅ **Robust** code that handles resources properly
- ✅ **Efficient** code that manages memory well
- ✅ **Maintainable** code that's easy to understand
- ✅ **Professional** code that follows best practices

By Abdullah Ali

Contact : +201012613453