Access Modifiers

Access modifiers are keywords that you add to a type or a type member's declaration to specify its accessibility from other types and other assemblies (compiled `.dll` or `.exe` files). They allow you to promote **encapsulation**, which is the principle of hiding the internal implementation details of an object and exposing only what is necessary for other parts of the program to interact with it.

Here are the primary access modifiers in C#, ordered roughly from most accessible to least accessible:

`public`

- **Fully accessible.**
- A `public` type or member can be accessed by any other code, regardless of the assembly it resides in.
- This is the implicit accessibility for members of an `enum` or an `interface`.

`internal`

- Accessible **only within the containing assembly** or from **friend assemblies**.
- This is the default accessibility for non-nested types (classes, structs, enums, interfaces) if no access modifier is explicitly specified.
- It's commonly used for components that are part of a library but not intended for direct use by external consumers of that library.

`private`

- Accessible **only within the containing type**.
- This is the default accessibility for members (fields, methods, properties, etc.) of a `class` or `struct` if no access modifier is explicitly specified.
- `private` members are the internal implementation details of a class and cannot be accessed from outside that class, even by derived classes.

`protected`

- Accessible **only within the containing type or its subclasses (derived types)**.

- `protected` members allow derived classes to access or extend behavior while keeping it hidden from unrelated types.

`protected internal`

- This is a **union** of `protected` and `internal` accessibility.
- A member that is `protected internal` is accessible:
    - Within the containing type.
    - From subclasses, regardless of their assembly.
    - From any code within the same assembly.

`private protected`

- This is an **intersection** of `protected` and `internal` accessibility.
- A member that is `private protected` is accessible:
    - Only within the containing type.
    - From subclasses *that reside within the same assembly*.
- This makes it *less accessible* than `protected` alone (which allows subclasses in any assembly) or `internal` alone (which allows any type in the same assembly). It's used for highly constrained internal access for derived types.

`file` (from C# 11)

- Accessible **only from within the same file**.
- This modifier can *only* be applied to **type declarations** (classes, structs, interfaces, enums, delegates).
- It's primarily intended for use by source generators, allowing them to create helper types that are truly local to a single generated file, preventing naming conflicts or accidental exposure.

Examples of Access Modifiers

Let's illustrate these with some code examples:

Type Accessibility:

```
// Class1 is internal (default accessibility for non-nested types)
// It cannot be accessed by code outside this assembly.
class Class1 { }

// Class2 is public and can be accessed by code from any assembly.
public class Class2 { }
```

Member Accessibility:

```
class ClassA
{
    int x; // x is private (default accessibility for class/struct members)
           // Only code within ClassA can access x.
}

class ClassB
{
    internal int x; // x is internal
                     // Any code within the same assembly as ClassB can access
x.
}
```

Inheritance and protected:

```
class BaseClass
{
    void Foo() { } // Foo is private (default)
                   // Only code within BaseClass can call Foo.

    protected void Bar() { } // Bar is protected
                             // Code within BaseClass AND any subclasses can
call Bar.
}

class Subclass : BaseClass
```

```
{
    void Test1()
    {
        // Foo(); // Error: Cannot access Foo because it is private in
BaseClass.
    }

    void Test2()
    {
        Bar(); // OK: Can access Bar because it is protected and Subclass is a
subclass.
    }
}
```

Friend Assemblies

You can allow `internal` members to be accessible from other assemblies by using the `System.Runtime.CompilerServices.InternalsVisibleTo` assembly attribute. This is often called creating **"friend assemblies."**

```
// In AssemblyA's AssemblyInfo.cs or .csproj:
[assembly:
System.Runtime.CompilerServices.InternalsVisibleTo("FriendAssemblyB")]

// Now, code in FriendAssemblyB can access internal members of AssemblyA.
```

If the friend assembly has a strong name (a cryptographic identity), you must specify its full public key in the attribute.

Accessibility Capping

A type's accessibility acts as an **upper cap** for the accessibility of its declared members. This means a member cannot be *more* accessible than its containing type.

The most common scenario is an `internal` type with `public` members:

```
// Class C is internal by default
class C
{
    public void Foo() { } // Foo is declared public, but...
}
```

Because C is internal, its public member Foo is effectively also internal. You cannot access Foo from outside the assembly where C is defined, even if Foo itself is declared public. A common reason for marking Foo as public in such a scenario is to facilitate refactoring; if C later becomes public, Foo will automatically become truly public without requiring a code change.

Restrictions on Access Modifiers

The C# compiler enforces strict rules to prevent inconsistent use of access modifiers:

- **Overriding Members:** When overriding a base class function (using override), the accessibility of the overridden function **must be identical** to the base class's virtual member. You cannot, for example, override a protected method with a public one.

  ```
  class BaseClass              { protected virtual void Foo() {} }
  class Subclass1 : BaseClass { protected override void Foo() {} } // OK
  // class Subclass2 : BaseClass { public override void Foo() {} } //
  Error: Cannot change accessibility
  ```

- (A specific exception exists: if overriding a protected internal method from another assembly, the override must simply be protected.)

- **Subclass Accessibility:** A derived class cannot be *more* accessible than its base class. It can be equally or less accessible.

```
internal class A {} // Base class is internal
// public class B : A {} // Error: Cannot derive public class B from
internal class A
```

- This prevents exposing internal implementation details through a public interface.

Properly applying access modifiers is fundamental to designing maintainable, secure, and robust software systems in C#. They are a primary mechanism for enforcing encapsulation and defining clear contracts between different parts of your codebase.