C# - Namespaces

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

namespace namespace\_name {

// code declarations

}

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows −

namespace\_name.item\_name;

The following program demonstrates use of namespaces −

[Live Demo](http://tpcg.io/SCXDku)

using System;

namespace first\_space {

class namespace\_cl {

public void func() {

Console.WriteLine("Inside first\_space");

}

}

}

namespace second\_space {

class namespace\_cl {

public void func() {

Console.WriteLine("Inside second\_space");

}

}

}

class TestClass {

static void Main(string[] args) {

first\_space.namespace\_cl fc = new first\_space.namespace\_cl();

second\_space.namespace\_cl sc = new second\_space.namespace\_cl();

fc.func();

sc.func();

Console.ReadKey();

}

}

When the above code is compiled and executed, it produces the following result −

Inside first\_space

Inside second\_space

The *using* Keyword

The **using** keyword states that the program is using the names in the given namespace. For example, we are using the **System** namespace in our programs. The class Console is defined there. We just write −

Console.WriteLine ("Hello there");

We could have written the fully qualified name as −

System.Console.WriteLine("Hello there");

You can also avoid prepending of namespaces with the **using** namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code −

Let us rewrite our preceding example, with using directive −

[Live Demo](http://tpcg.io/5TZope)

using System;

using first\_space;

using second\_space;

namespace first\_space {

class abc {

public void func() {

Console.WriteLine("Inside first\_space");

}

}

}

namespace second\_space {

class efg {

public void func() {

Console.WriteLine("Inside second\_space");

}

}

}

class TestClass {

static void Main(string[] args) {

abc fc = new abc();

efg sc = new efg();

fc.func();

sc.func();

Console.ReadKey();

}

}

When the above code is compiled and executed, it produces the following result −

Inside first\_space

Inside second\_space

Nested Namespaces

You can define one namespace inside another namespace as follows −

namespace namespace\_name1 {

// code declarations

namespace namespace\_name2 {

// code declarations

}

}

You can access members of nested namespace by using the dot (.) operator as follows −

[Live Demo](http://tpcg.io/jW9nSY)

using System;

using first\_space;

using first\_space.second\_space;

namespace first\_space {

class abc {

public void func() {

Console.WriteLine("Inside first\_space");

}

}

namespace second\_space {

class efg {

public void func() {

Console.WriteLine("Inside second\_space");

}

}

}

}

class TestClass {

static void Main(string[] args) {

abc fc = new abc();

efg sc = new efg();

fc.func();

sc.func();

Console.ReadKey();

}

}

When the above code is compiled and executed, it produces the following result −

Inside first\_space

Inside second\_space

# What Are Access Modifiers In C#

## Access Modifiers In C#

Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself. For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only.

## Why to use access modifiers?

Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement encapsulation of OOP. Access modifiers allow you to define who does or who doesn't have access to certain features.

In C# there are 6 different types of Access Modifiers.

|  |  |
| --- | --- |
| Modifier | Description |
| public | There are no restrictions on accessing public members. |
| private | Access is limited to within the class definition. This is the default access modifier type if none is formally specified |
| protected | Access is limited to within the class definition and any class that inherits from the class |
| internal | Access is limited exclusively to classes defined within the current project assembly |
| protected internal | Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables. |
| private protected | Access is limited to the containing class or types derived from the containing class within the current assembly. |

1. **using** System;
2. **namespace** AccessModifiers
3. {
4. **class** Program
5. {
6. **class** AccessMod
7. {
8. **public** **int** num1;
9. }
10. **static** **void** Main(**string**[] args)
11. {
12. AccessMod ob1 = **new** AccessMod();
13. //Direct access to public members
14. ob1.num1 = 100;
15. Console.WriteLine("Number one value in main {0}", ob1.num1);
16. Console.ReadLine();
17. }
18. }
19. }

## public modifier

The public keyword is an access modifier for types and type members. Public access is the most permissive access level.

There are no restrictions on accessing public members.

**Accessibility**

* Can be accessed by objects of the class
* Can be accessed by derived classes

**Example:**In the following example num1 is direct access.

## private modifier

Private access is the least permissive access level.

Private members are accessible only within the body of the class or the struct in which they are declared.

**Accessibility**

* Cannot be accessed by object
* Cannot be accessed by derived classes

**Example:**In the following example num2 is not accessible outside the class.

1. **using** System;
2. **namespace** AccessModifiers
3. {
4. **class** Program
5. {
6. **class** AccessMod
7. {
8. **public** **int** num1;
9. **int** num2;
10. }
11. **static** **void** Main(**string**[] args)
12. {
13. AccessMod ob1 = **new** AccessMod();
14. //Direct access to public members
15. ob1.num1 = 100;
16. //Access to private member is not permitted
17. ob1.num2 = 20;
18. Console.WriteLine("Number one value in main {0}", ob1.num1);
19. Console.ReadLine();
20. }
21. }
22. }

The above program will give compilation error, as access to private is not permissible. In the below figure you can see the private member num2 is not available.

**Graphical user interface, text, application, email

Description automatically generated**

## protected modifier

A protected member is accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.

A protected member of a base class is accessible in a derived class only if the access takes place through the derived class type.

**Accessibility**

* Cannot be accessed by object
* By derived classes

1. **using** System;
2. **namespace** AccessModifiers
3. {
4. **class** Program
5. {
6. **class** Base
7. {
8. **protected** **int** num1;
9. }
10. **class** Derived : Base
11. {
12. **public** **int** num2;
13. **static** **void** Main(**string**[] args)
14. {
15. Base ob1 = **new** Base();
16. Derived ob2 = **new** Derived();
17. ob2.num1 = 20;
18. // Access to protected member as it is inherited by the Derived class
19. ob2.num2 = 90;
20. Console.WriteLine("Number2 value {0}", ob2.num2);
21. Console.WriteLine("Number1 value which is protected {0}", ob2.num1);
22. Console.ReadLine();
23. }
24. }
25. }
26. }

In the above program we try to access protected member in main it is not available as shown in the picture below that num1 is not listed in intellisense.

Text

Description automatically generated

## internal modifier

The internal keyword is an access modifier for types and type members. We can declare a class as internal or its member as internal. Internal members are accessible only within files in the same assembly (.dll).

In other words, access is limited exclusively to classes defined within the current project assembly.

**Accessibility**

In same assembly (public)

* Can be accessed by objects of the class
* Can be accessed by derived classes

In other assembly (internal)

* Cannot be accessed by object
* Cannot be accessed by derived classes

## protected internal modifier

The protected internal accessibility means protected OR internal, not protected AND internal.

In other words, a protected internal member is accessible from any class in the same assembly, including derived classes.

The protected internal access modifier seems to be a confusing but is a union of protected and internal in terms of providing access but not restricting. It allows:

* Inherited types, even though they belong to a different assembly, have access to the protected internal members.
* Types that reside in the same assembly, even if they are not derived from the type, also have access to the protected internal members.

**Default access**

A default access level is used if no access modifier is specified in a member declaration. The following list defines the default access modifier for certain C# types:

**enum:**The default and only access modifier supported is public.

**class:**The default access for a class is private. It may be explicitly defined using any of the access modifiers.

**interface:**The default and only access modifier supported is public.

**struct:**The default access is private with public and internal supported as well.

The default access may suffice for a given situation, but you should specify the access modifier you want to use to ensure proper application behavior.

**Note:**Interface and enumeration members are always public and no access modifiers are allowed.

## Conclusion

I hope that this article would have helped you in understanding accessibility modifiers. Your feedback and constructive contributions are welcome.