Member Accessibility and Overloading

One of the important decisions to make when designing an object is how accessible to make the members. In C#, accessibility can be controlled in several ways.

Class Accessibility

The coarsest level at which accessibility (also known as visibility) can be controlled is at the class. In most cases, the only valid modifiers on a class are public, which means that everybody can see the class, and internal. The exception to this is nesting classes inside of other classes, which is a bit more complicated and is covered in Chapter 7.

Internal is a way of granting access to a wider set of classes without granting access to everybody, and it is most often used when writing helper classes that should be hidden from the ultimate user of the class. In the .NET Runtime world, internal equates to allowing access to all classes that are in the same assembly as this class.

Note In the C++ world, such accessibility is usually granted by the use of friends, which provides access to a specific class. The use of friends provides greater granularity in specifying who can access a class, but in practice the access provided by internal is sufficient. In general, all classes should be internal unless other assemblies need to be able to access them.

Using Internal on Members

The internal modifier can also be used on a member, which then allows that member to be accessible from classes in the same assembly as itself, but not from classes outside the assembly.

This is especially useful when several public classes need to cooperate, but some of the shared members shouldn’t be exposed to the general public. Consider the following example:

```csharp
public class DrawingObjectGroup
{
    public DrawingObjectGroup()
    {
        m_objects = new DrawingObject[10];
        m_objectCount = 0;
    }
}
```
public void AddObject(DrawingObject obj)
{
    if (m_objectCount < 10)
    {
        m_objects[m_objectCount] = obj;
        m_objectCount++;
    }
}

public void Render()
{
    for (int i = 0; i < m_objectCount; i++)
    {
        m_objects[i].Render();
    }
}

DrawingObject[] m_objects;
int m_objectCount;

public class DrawingObject
{
    internal void Render() {}
}

class Test
{
    public static void Main()
    {
        DrawingObjectGroup group = new DrawingObjectGroup();
        group.AddObject(new DrawingObject());
    }
}

Here, the DrawingObjectGroup object holds up to ten drawing objects. It’s valid for the user to have a reference to a DrawingObject, but it would be invalid for the user to call Render() for that object, so this is prevented by making the Render() function internal.

---

This code doesn’t make sense in a real program. The .NET Common Language Runtime has a number of collection classes that would make this sort of code much more straightforward and less error prone. See Chapter 33 for more information.

---

Expanding Internal Accessibility

In certain scenarios, it is useful to provide internal-level access to a class that is not in the same assembly. See the “Expanding Internal Accessibility” section in Chapter 31 for more information.