Working with Scope and Accessibility Modifiers

Scope of Variables

The *scope* of any variable is a way of describing the timeframe in which the variable exists as well as where it can be accessed. This timeframe is based on where in your program you declare the variable. While a variable is in scope, you are able to use it. In this lesson, you will learn about three different levels of scoping in C#: method-level, block-level, and class-level.

Block Scoping

In previous lessons, you've seen code blocks for *if* statements, *while* loops, *for* loops, and you'll see more going forward. These blocks are often surrounded by curly braces (`{ }`); the exception to this is single-line code blocks. Code blocks are the most limited (and thus overriding) type of scoping you will learn in this lesson. In C#, when you declare a variable inside of a block, it exists from the line where it is declared until the end of the block. This allows you to declare short-lived variables in your code. The example you've seen often in previous lessons is the *loop control variable* from `for` loops. While thinking about block-level scoping, consider this example that shows two loops:

```
public static void Main()
{
    for (int i = 0; i < 10; i++)
    {
        string message = $"I ran this loop {i} times already.";
        Console.WriteLine(message);
    }
    for (int i = 0; i < 10; i++)
    {
        string message = $"I ran this loop {i} times already.";
        Console.WriteLine(message);
    }
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        for (int i = 0; i < 10; i++)
        {
            string message = $"I ran this loop {i} times already.";
            Console.WriteLine(message);
        }
        for (int i = 0; i < 10; i++)
```

```
        {
            string message = $"I ran this loop {i} times already.";
            Console.WriteLine(message);
        }
    }
}
```

Notice that each loop has a variable named `message`. Since each variable exists within the scope of the loop it's declared in, we're allowed to use the same names for them. By the time we reach the second `for` loop, all variables declared in the first loop are gone. In fact, the scope of the variables declared in these loops is just one iteration through the loop. This means that each time through the loop, we're getting a new instance of `string` named `message`.

Note {.note}
A special circumstance exists for variables declared within the `for` loop's declaration. You may have noticed that there are also two `i` variables declared. These have a slightly modified block-level scope. They also only exist within their respective loop, however, these persist through iterations of the loop. That's how you're able to modify the variable's value and have the change persist the next time through the loop (allowing it to count up).

If, outside the `for` loop, you tried to access a variable created inside the loop, you would receive a compiler error. The following code is an example that will generate this error:

```
public static void Main()
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(i);
    }
    Console.WriteLine(i); // Won't compile
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
        // Try uncommenting this WriteLine - it won't compile
        //Console.WriteLine(i);
    }
}
```

Method Scoping

Any variables declared within a method exist from that point forward in the method, and each time the method is called, new instances of the variables get created.

```
public static void DoesCompile()
{
    int width = 10;
    int length = 4;
    int area = width * length;
    Console.WriteLine($"{nameof(area)}:{area}");
}
public static void DoesNotCompile()
{
    int area = width * length; // width and length don't exist yet
    int width = 10;
    int length = 4;
    Console.WriteLine($"{nameof(area)}:{area}");
}
public static void AlsoDoesNotCompile()
{
    int area = width * length; // width and length not in this method
    Console.WriteLine($"{nameof(area)}:{area}");
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        DoesCompile();
        DoesNotCompile();
        AlsoDoesNotCompile();
    }
    public static void DoesCompile()
    {
        int width = 10;
        int length = 4;
        int area = width * length;
        Console.WriteLine($"{nameof(area)}:{area}");
    }
    public static void DoesNotCompile()
    {
        int area = width * length; // width and length don't exist yet
        int width = 10;
        int length = 4;
        Console.WriteLine($"{nameof(area)}:{area}");
    }
    public static void AlsoDoesNotCompile()
    {
        int area = width * length; // width and length not in this method
        Console.WriteLine($"{nameof(area)}:{area}");
```

```
    }
}
```

Within a method, there can only be one object of any given name. We got away with reusing the same variable names using the block level scoping of our loop control variables in an earlier example, however, an object of the same name outside of the block scope will show why that does not work. See this example showing this naming conflict:

```
public static void DoWork()
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(i);
    }
    int i = 777; // Compiler error here
    Console.WriteLine(i);
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        DoWork();
    }
    public static void DoWork()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
        int i = 777; // Compiler error here
        Console.WriteLine(i);
    }
}
```

Class Scoping

Some objects you create will have a lifetime based on the lifetime of the class in which they're declared. These will often be the fields and properties that you declare on your objects, which only exist as long as the object containing them still exists. Up to this point, the order in which objects are declared has mattered. Within a method or a block, a variable must be declared before being used. For class-level scoping, you may reference members declared below the code you're writing. The following is an example class showing some class-level scoping.

```
public class Circle
{
    public Circle(decimal radius)
```

```
    {
        Radius = radius; // Using Radius before it's declared
    }

    public decimal Radius { get; private set; }

    public decimal Perimeter()
    {
        return 2 * _pi * this.Radius; // Using _pi before it's declared
    }

    private const decimal _pi = 3.14159m;
}



using System;

public class Program
{
    public static void Main()
    {
        var circle = new Circle(5m);
        Console.WriteLine(circle.Perimeter());
    }
}

public class Circle
{
    public Circle(decimal radius)
    {
        Radius = radius; // Using Radius before it's declared
    }

    public decimal Radius { get; private set; }

    public decimal Perimeter()
    {
        return 2 * _pi * this.Radius; // Using _pi before it's declared
    }

    private const decimal _pi = 3.14159m;
}
```

When an object of this type is created, a variable is created to store the value of `Radius`. That variable exists and is accessible during the lifetime of that object instance.

Access Modifiers

As your programs grow and become more complex, there will be times when you need to limit access to some of your code. These limitations you impose determine which other code is allowed to access the code you're restricting. You impose these restrictions using *access modifiers*, which you may have noticed affecting namespaces, classes, methods, properties, and fields in earlier examples.

In this lesson, you'll learn what access modifiers are and how to use them. You'll learn when and why to use these modifiers to effectively limit access to information and behavior in your programs in the Encapsulation and Object-Oriented Design lesson. Consider this example:

```
public class Person
{
    public Person(string firstName, string lastName, DateTime dateOfBirth)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.DateOfBirth = dateOfBirth;
    }

    protected string FirstName { get; private set; }
    protected string LastName { get; private set; }
    public DateTime DateOfBirth { get; private set; }

    public string FullName { get { return $"{this.FirstName} {this.LastName}"; } }

    public bool IsAnAdult()
    {
        var eighteenYearsAgo = DateTime.Today.AddYears(-18);
        return this.DateOfBirth < eighteenYearsAgo;
    }
}

public class Student : Person
{
    public Student (string firstName, string lastName, DateTime dateOfBirth)
        : base(firstName, lastName, dateOfBirth)
    { }
    public string SchoolName { get; set; }

    public string RosterName { get { return $"{this.LastName}, {this.FirstName}"; } }
}



using System;

public class Program
{
    public static void Main()
    {
        var jimmy = new Student("Jimmy", "Jones", new DateTime(1990, 3, 15));
        Console.WriteLine(jimmy.RosterName);
    }
}
```

```
public class Person
{
    public Person(string firstName, string lastName, DateTime dateOfBirth)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.DateOfBirth = dateOfBirth;
    }

    protected string FirstName { get; private set; }
    protected string LastName { get; private set; }
    public DateTime DateOfBirth { get; private set; }

    public string FullName { get { return $"{this.FirstName} {this.LastName}"; } }

    public bool IsAnAdult()
    {
        var eighteenYearsAgo = DateTime.Today.AddYears(-18);
        return this.DateOfBirth < eighteenYearsAgo;
    }
}

public class Student : Person
{
    public Student (string firstName, string lastName, DateTime dateOfBirth)
        : base(firstName, lastName, dateOfBirth)
    { }
    public string SchoolName { get; set; }

    public string RosterName { get { return $"{this.LastName}, {this.FirstName}"; } }
}
```

The access modifiers used in this example are `public`, `private`, and `protected`. This lesson will also explain the `internal` modifier, but that's more related to the Understanding Namespaces lesson.

> **Tip** {.tip .cpp}
> Instead of grouping members of classes with the same access level, C# includes the access modifier for each member.

Notice that the access modifiers are the first word in the declaration of the classes, methods, and properties. This holds for most cases, however, you'll notice one exception to this in the example:

```
protected string FirstName { get; private set; }
```

With properties, it's possible to further restrict the setter method, so that it's less accessible than the getter method.

public

Code that's available for use by any other code should use the `public` access modifier. This, as its name suggests, makes it available publicly for any other code to use. It's simple to understand and work with. The `IsAnAdult()` method in the example above uses this access modifier, because any other code is allowed to call it:

```
public bool IsAnAdult()
{
    var eighteenYearsAgo = DateTime.Today.AddYears(-18);
    return this.DateOfBirth < eighteenYearsAgo;
}
```

private

For code that should only be usable by other code in the same class, `private` is the correct access modifier. This is the most restrictive of the access modifiers, and is used to restrict this property `set` method in the example above:

```
public DateTime DateOfBirth { get; private set; }
```

Because it is `private`, only the `Person` class can set this value; the inheriting `Student` is not even able to modify the value.

protected

When dealing with inheritance, the `protected` access modifier is often useful. It allows a child class to use some of the otherwise restricted members of the parent. In the example above, `FirstName` and `LastName` are only accessible from within `Person` or its child classes, as you can see in the `Student` class's `RosterName` property.

```
public string RosterName { get { return $"{this.LastName}, {this.FirstName}"; } }
```

internal

Like `protected`, `internal` is more accessible than `private`, but less than `public`. When using this access modifier, the code being modified may be used by any other code in the same assembly. These keep developers without access to your assembly's source code from using the `internal` code.

Tip {.tip .java}
The `internal` access modifier is the C# equivalent of the `default` access modifier in Java.

Tip {.tip .vb}
The `internal` access modifier is the C# equivalent of the `Friend` access modifier in Visual Basic.

Note {.note}
The `internal` and `protected` may be used in combination. Doing this will create the union of the two allowances rather than the limitations, meaning access is provided to inheriting classes as well as within the same assembly.