Microsoft

Microsoft Visual C# Step by Step

Tenth Edition

Professional

John Sharp

Microsoft

Microsoft Visual C# Step by Step

Tenth Edition

John Sharp

In this case, if c is null, nothing is written to the command window. Alternatively, you could use the null-conditional operator on the Circle object before you attempt to call the Circle.Area method:

```
Console.WriteLine($"The area of circle c is {c?.Area()}");
```

The null-conditional operator tells the C# runtime to ignore the current statement if the variable you have applied the operator to is null. In this case, the command window would display the following text:

```
The area of circle c is
```

Both approaches are valid and might meet your needs in different scenarios. The null-conditional operator can help you keep your code concise, particularly when you deal with complex properties with nested reference types that could all be null valued.

Alongside the null-conditional operator, C# provides two null-coalescing operators. The first of these, ??, is a binary operator that returns the value of the operand on the left if it isn't null; otherwise, it returns the value of the operand on the right. In the following example, variable c2 is assigned a reference to c if c isn't null; otherwise, it is assigned a reference to a new Circle object:

```
Circle c = ...; // might be null, might be a new Circle object
...
var c2 = c ?? new Circle(42) ;
```

The null-coalescing assignment operator, ??=, assigns the value of the operand on the right to the operand on the left only if the left operand is null. If the left operand references some other value, it is unchanged.

```
Circle c = ...; // might be null, might be a new Circle object
Circle c3 = ...; // might be null, might be a new Circle object
...
var c3 ??= c; // Only assign c3 if it is null, otherwise leave unchanged;
```

Using nullable types

The null value is very useful for initializing reference types. Sometimes, though, you need an equivalent value for value types. null is itself a reference, so you cannot assign it to a value type. The following statement is therefore illegal in C#:

```
int i = null; // illegal
```

However, C# defines a modifier that you can use to declare that a variable is a *nullable* value type. A nullable value type behaves similarly to the original value type, but you can assign the null value to it. You use the question mark (?) to indicate that a value type is nullable, like this:

```
int? i = null; // legal
```

You can ascertain whether a nullable variable contains null by testing it in the same way as you test a reference type.

```
if (i is null)
      ...
```

You can assign an expression of the appropriate value type directly to a nullable variable. The following examples are all legal:

```
int? i = null;
int j = 99;
i = 100; // Copy a value type constant to a nullable type
i = j; // Copy a value type variable to a nullable type
```

You should note that the converse is not true. You cannot assign a nullable variable to an ordinary value type variable. So, given the definitions of variables `i` and `j` from the preceding example, the following statement is not allowed:

```
j = i; // illegal
```

This makes sense when you consider that the variable `i` might contain `null`, and `j` is a value type that cannot contain `null`. This also means that you cannot use a nullable variable as a parameter to a method that expects an ordinary value type. If you recall, the `Pass.Value` method from the preceding exercise expects an ordinary `int` parameter, so the following method call will not compile:

```
int? i = 99;
Pass.Value(i); // Compiler error
```

Note Take care not to confuse nullable types with the null-conditional operator. Nullable types are indicated by appending a question mark to the type name, whereas the null-conditional operator is appended to the variable name.

Understanding the properties of nullable types

A nullable type exposes a pair of properties that you can use to determine whether the type actually has a non-null value and what this value is. The `HasValue` property indicates whether a nullable type contains a value or is `null`. You can retrieve the value of a non-null nullable type by reading the `Value` property, like this:

```
int? i = null;
...
if (!i.HasValue)
{
    // If i is null, then assign it the value 99
    i = 99;
}
else
{
    // If i is not null, then display its value
    Console.WriteLine(i.Value);
}
```

In Chapter 4, "Using decision statements," you saw that the NOT operator (`!`) negates a Boolean value. The preceding code fragment tests the nullable variable `i`, and if it does not have a value (it is `null`), it assigns it the value 99; otherwise, it displays the value of the variable. In this example, using the `HasValue` property does not provide any benefit over testing for a `null` value directly. Additionally, reading the `Value` property is a long-winded way of reading the contents of the variable. However, these apparent shortcomings are caused by the fact that `int?` is a very simple nullable type. You can create more complex value types and use them to declare nullable variables where the advantages of using the `HasValue` and `Value` properties become more apparent. You'll see some examples in Chapter 9, "Creating value types with enumerations and structures."

Note The `Value` property of a nullable type is read-only. You can use this property to read the value of a variable but not to modify it. To update a nullable variable, use an ordinary assignment statement.

Using ref and out parameters

Ordinarily, when you pass an argument to a method, the corresponding parameter is initialized with a copy of the argument. This is true regardless of whether the parameter is a value type (such as an `int`), a nullable type (such as `int?`), or a reference type (such as a `WrappedInt`). This arrangement means that it's impossible for any change to the parameter to affect the value of the argument passed in. For example, in the following code, the value output to the console is 42, not 43. The `doIncrement` method increments a copy of the argument (`arg`) and *not* the original argument, as demonstrated here:

```
static void doIncrement(int param)
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(arg);
    Console.WriteLine(arg); // writes 42, not 43
}
```

In the preceding exercise, you saw that if the parameter to a method is a reference type, any changes made by using that parameter change the data referenced by the argument passed in. The key point is this: although the data that was referenced changed, the argument passed in as the parameter did not. It still references the same object. In other words, although it's possible to modify the object that the argument refers to through the parameter, it's not possible to modify the argument itself—for example, to set it to refer to a completely different object. Most of the time, this guarantee is very useful and can help reduce the number of bugs in a program. Occasionally, however, you might want to write a method that actually needs to modify an argument. C# provides the `ref` and `out` keywords so that you can do this.

Creating ref parameters

If you prefix a parameter with the `ref` keyword, the C# compiler generates code that passes a reference to the actual argument rather than a copy of the argument. When using a `ref` parameter, anything you do to the parameter you also do to the original argument because the parameter and the argument both reference the same data.

When you pass an argument as a `ref` parameter, you must also prefix the argument with the `ref` keyword. This syntax provides a useful visual cue to the programmer that the argument might change. Here's the preceding example again, this time modified to use the `ref` keyword:

```
static void doIncrement(ref int param) // using ref
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(ref arg); // using ref
    Console.WriteLine(arg); // writes 43
}
```

This time, the `doIncrement` method receives a reference to the original argument rather than a copy, so any changes the method makes by using this reference actually change the original value. That's why the value `43` is displayed on the console.

Remember that C# enforces the rule that you must assign a value to a variable before you can read it. This rule also applies to method arguments; you cannot pass an uninitialized value as an argument to a method even if an argument is defined as a `ref` argument. For example, in the following example, `arg` is not initialized, so this code will not compile. This failure occurs because the statement `param++;` within the `doIncrement` method is really an alias for the statement `arg++;` and this operation is allowed only if `arg` has a defined value:

```
static void doIncrement(ref int param)
{
    param++;
}

static void Main()
{
    int arg; // not initialized
    doIncrement(ref arg);
    Console.WriteLine(arg);
}
```

Creating out parameters

The compiler checks whether a `ref` parameter has been assigned a value before calling the method. However, there might be times when you want the method itself to initialize the parameter. You can do this with the `out` keyword.

The `out` keyword is syntactically similar to the `ref` keyword. You can prefix a parameter with the `out` keyword so that the parameter becomes an alias for the argument. As when using `ref`, anything you do to the parameter, you also do to the original argument. When you pass an argument to an `out` parameter, you must also prefix the argument with the `out` keyword.

The keyword `out` is short for *output*. When you pass an `out` parameter to a method, the method *must* assign a value to it before it finishes or returns, as shown in the following example:

```
static void doInitialize(out int param)
{
    param = 42; // Initialize param before finishing
}
```

The following example does not compile because `doInitialize` does not assign a value to `param`:

```
static void doInitialize(out int param)
{
    // Do nothing
}
```

Because an `out` parameter must be assigned a value by the method, you're allowed to call the method without initializing its argument. For example, the following code calls `doInitialize` to initialize the variable `arg`, which is then displayed on the console:

```
static void doInitialize(out int param)
{
    param = 42;
}

static void Main()
{
    int arg; // not initialized
    doInitialize(out arg); // legal
    Console.WriteLine(arg); // writes 42
}
```

Note You can combine the declaration of an `out` variable with its use as a parameter rather than performing these tasks separately. For example, you could replace the first two statements in the `Main` method in the previous example with this single line of code:

```
doInitialize(out int arg);
```

In the next exercise, you'll practice using `ref` parameters.

To use ref parameters

1. Return to the Parameters project in Visual Studio 2022.
2. Display the Pass.cs file in the Code and Text Editor window.

3. Edit the `Value` method to accept its parameter as a `ref` parameter.

   The `Value` method should look like this:

```
class Pass
{
    public static void Value(ref int param)
    {
        param = 42;
    }
    ...
}
```

4. Display the Program.cs file in the Code and Text Editor window.

5. Uncomment the first four statements.

   Notice that the third statement of the `doWork` method, `Pass.Value(i)`, indicates an error. The error occurs because the `Value` method now expects a `ref` parameter.

6. Edit this statement so that the `Pass.Value` method call passes its argument as a `ref` parameter.

Note Leave the four statements that create and test the WrappedInt object as they are.

The `doWork` method should now look like this:

```
class Program
{
  static void doWork()
  {
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(ref i);
    Console.WriteLine(i);
    ...
  }
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

   This time, the first two values written to the console window are 0 and 42. This result shows that the call to the `Pass.Value` method has successfully modified the argument `i`.

8. Press **Enter** to close the application and return to Visual Studio 2022.

Note You can use the `ref` and `out` modifiers on reference type parameters as well as on value type parameters. The effect is the same: the parameter becomes an alias for the argument.