Declaration statements

Article • 02/08/2023 • 7 minutes to read

A *declaration statement* declares a new variable, and optionally, initializes it. All variables have declared type. You can learn more about types in the article on the .NET type system. Typically, a declaration includes a type and a variable name. It can also include an initialization: the `=` operator followed by an expression. The type may be replaced with `var`. The declaration or the expression may include the `ref` modifier to declare that the new variable refers to an existing storage location.

Implicitly typed local variables

Variables that are declared at method scope can have an implicit "type" `var`. An implicitly typed local variable is strongly typed as if you had declared the type yourself, but the compiler determines the type. The following two declarations of `a` and `b` are functionally equivalent:

C#

```csharp
var a = 10; // Implicitly typed.
int b = 10; // Explicitly typed.
```

Important

When `var` is used with **nullable reference types** enabled, it always implies a nullable reference type even if the expression type isn't nullable. The compiler's null state analysis protects against dereferencing a potential `null` value. If the variable is never analysis protects against dereferencing a potential `null` value. If the variable is never assigned to an expression that maybe null, the compiler won't emit any warnings. If you assign the variable to an expression that might be null, you must test that it isn't null before dereferencing it to avoid any warnings.

A common use of the `var` keyword is with constructor invocation expressions. The use of `var` allows you to not repeat a type name in a variable declaration and object instantiation, as the following example shows:

C#

```csharp
var xs = new List<int>();
```

Beginning with C# 9.0, you can use a target-typed new expression as an alternative:

C#

```csharp
List<int> xs = new();
List<int>? ys = new();
```

In pattern matching, the `var` keyword is used in a var pattern.

The following example shows two query expressions. In the first expression, the use of `var` is permitted but isn't required, because the type of the query result can be stated explicitly as an `IEnumerable<string>`. However, in the second expression, `var` allows the result to be a collection of anonymous types, and the name of that type isn't accessible except to the compiler itself. Use of `var` eliminates the requirement to create a new class for the result. In Example #2, the `foreach` iteration variable `item` must also be implicitly typed.

C#

```csharp
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{


    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

Ref locals

You add the `ref` keyword before the type of a variable to declare a `ref` local. A `ref` local is a variable that *refers to* other storage. Assume the `GetContactInformation` method is declared as a ref return:

C#

```csharp
public ref Person GetContactInformation(string fname, string lname)
```

Let's contrast these two assignments:

C#

```csharp
Person p = contacts.GetContactInformation("Brandie", "Best");
ref Person p2 = ref contacts.GetContactInformation("Brandie", "Best");
```

The variable `p` holds a *copy* of the return value from `GetContactInformation`. It's a separate storage location from the `ref` return from `GetContactInformation`. If you change any property of `p`, you are changing a copy of the `Person`.

The variable `p2` *refers to* the storage location for the `ref` return from `GetContactInformation`. It's the same storage as the `ref` return from `GetContactInformation`. If you change any property of `p2`, you are changing that single instance of a `Person`.

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how one can define a ref local value that is used to reference a value.

C#

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

The `ref` keyword is used both before the local variable declaration *and* before the value in the second example. Failure to include both `ref` keywords in the variable declaration and assignment in both examples results in compiler error CS8172, "Can't initialize a by-reference variable with a value."

C#

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to dif-
ferent storage.
```

Ref local variables must still be initialized when they're declared.

The following example defines a `NumberStore` class that stores an array of integer values. The `FindNumber` method returns by reference the first number that is greater than or equal to the number passed as an argument. If no number is greater than or equal to the argument, the method returns the number in index 0.

C#

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

The following example calls the `NumberStore.FindNumber` method to retrieve the first value that is greater than or equal to 16. The caller then doubles the value returned by the method. The output from the example shows the change reflected in the value of the array elements of the `NumberStore` instance.

C#

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//       Original sequence: 1 3 7 15 31 63 127 255 511 1023
//       New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Without support for reference return values, such an operation is performed by returning the index of the array element along with its value. The caller can then use this index to modify the value in a separate method call. However, the caller can also modify the index to access and possibly modify other array values.

The following example shows how the `FindNumber` method could be rewritten to use ref local reassignment:

C#

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && (numbers[ctr] >= target))
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

This second version is more efficient with longer sequences in scenarios where the number sought is closer to the end of the array, as the array is iterated from end towards the beginning, causing fewer items to be examined.

The compiler enforces scope rules on `ref` variables: `ref` locals, `ref` parameters, and `ref` fields in `ref struct` types. The rules ensure that a reference doesn't outlive the object to which it refers. See the section on scoping rules in the article on method parameters.

ref and readonly

The `readonly` modifier can be applied to `ref` local variables and `ref` fields. The `readonly` modifier affects the expression to its right. See the following example declarations:

```C#
ref readonly int aConstant; // aConstant can't be value-reassigned.
readonly ref int Storage; // Storage can't be ref-reassigned.
readonly ref readonly int CantChange; // CantChange can't be value-reassigned
or ref-reassigned.
```

- *value reassignment* means the value of the variable is reassigned.
- *ref assignment* means the variable now refers to a different object.

The `readonly ref` and `readonly ref readonly` declarations are valid only on `ref` fields in a `ref struct`.

scoped ref

The contextual keyword `scoped` restricts the lifetime of a value. The `scoped` modifier restricts the *ref-safe-to-escape* or *safe-to-escape* lifetime, respectively, to the current method. Effectively, adding the `scoped` modifier asserts that your code won't extend the lifetime of the variable.