Method Parameters (C# Reference)

Article • 04/12/2023

In C#, arguments can be passed to parameters either by value or by reference. Remember that C# types can be either reference types (`class`) or value types (`struct`):

- *Pass by value* means **passing a copy of the variable** to the method.
- *Pass by reference* means **passing access to the variable** to the method.
- A variable of a *reference type* contains a reference to its data.
- A variable of a *value type* contains its data directly.

Because a struct is a value type, when you pass a struct by value to a method, the method receives and operates on a copy of the struct argument. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a reference type, not a value type. When a reference type is passed by value to a method, the method receives a copy of the reference to the class instance. That is, the called method receives a copy of the address of the instance, and the calling method retains the original address of the instance. The class instance in the calling method has an address, the parameter in the called method has a copy of the address, and both addresses refer to the same object. Because the parameter contains only a copy of the address, the called method cannot change the address of the class instance in the calling method. However, the called method can use the copy of the address to access the class members that both the original address and the copy of the address reference. If the called method changes a class member, the original class instance in the calling method also changes.

The output of the following example illustrates the difference. The value of the `willIChange` field of the class instance is changed by the call to method `ClassTaker` because the method uses the address in the parameter to find the specified field of the class instance. The `willIChange` field of the struct in the calling method is not changed by the call to method `StructTaker` because the value of the argument is a copy of the struct itself, not a copy of its address. `StructTaker` changes the copy, and the copy is lost when the call to `StructTaker` is completed.

C#

```
class TheClass
{
    public string? willIChange;
}
```

```
struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}",
testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.will-
IChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

How an argument is passed, and whether it's a reference type or value type controls what modifications made to the argument are visible from the caller.

Pass a value type by value

When you pass a *value* type *by value*:

- If the method assigns the parameter to refer to a different object, those changes **aren't** visible from the caller.
- If the method modifies the state of the object referred to by the parameter, those changes **aren't** visible from the caller.

The following example demonstrates passing value-type parameters by value. The variable `n` is passed by value to the method `SquareIt`. Any changes that take place inside the method have no effect on the original value of the variable.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}",
n);

SquareIt(n);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}",
n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

The variable `n` is a value type. It contains its data, the value `5`. When `SquareIt` is invoked, the contents of `n` are copied into the parameter `x`, which is squared inside the method. In `Main`, however, the value of `n` is the same after calling the `SquareIt` method as it was before. The change that takes place inside the method only affects the local variable `x`.

Pass a value type by reference

When you pass a *value* type *by reference*:

- If the method assigns the parameter to refer to a different object, those changes **aren't** visible from the caller.
- If the method modifies the state of the object referred to by the parameter, those changes **are** visible from the caller.

The following example is the same as the previous example, except that the argument is passed as a `ref` parameter. The value of the underlying argument, `n`, is changed when `x` is changed in the method.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}",
n);

SquareIt(ref n);  // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}",
n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

In this example, it is not the value of `n` that is passed; rather, a reference to `n` is passed. The parameter `x` is not an int; it is a reference to an `int`, in this case, a reference to `n`. Therefore, when `x` is squared inside the method, what actually is squared is what `x` refers to, `n`.

Pass a reference type by value

When you pass a *reference* type *by value*:

- If the method assigns the parameter to refer to a different object, those changes **aren't** visible from the caller.

- If the method modifies the state of the object referred to by the parameter, those changes **are** visible from the caller.

The following example demonstrates passing a reference-type parameter, `arr`, by value, to a method, `Change`. Because the parameter is a reference to `arr`, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the
first element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the
first element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888;  // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 };   // This change is
local.
    System.Console.WriteLine("Inside the method, the first element
is: {0}", pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

In the preceding example, the array, `arr`, which is a reference type, is passed to the method without the `ref` parameter. In such a case, a copy of the reference, which points to `arr`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element, in this case from `1` to `888`. However, allocating a new portion of memory by using the new operator inside the `Change` method makes the variable `pArray` reference a new array. Thus, any changes after that will not affect the original array, `arr`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

Pass a reference type by reference

When you pass a *reference* type *by reference*:

- If the method assigns the parameter to refer to a different object, those changes **are** visible from the caller.
- If the method modifies the state of the object referred to by the parameter, those changes **are** visible from the caller.

The following example is the same as the previous example, except that the `ref` keyword is added to the method header and call. Any changes that take place in the method affect the original variable in the calling program.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the
first element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the
first element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original vari-
ables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element
is: {0}", pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

All of the changes that take place inside the method affect the original array in `Main`. In fact, the original array is reallocated using the `new` operator. Thus, after calling the `Change` method, any reference to `arr` points to the five-element array, which is created in the `Change` method.

Scope of references and values

Methods can store the values of parameters in fields. When parameters are passed by value, that's always safe. Values are copied, and reference types are reachable when stored in a field. Passing parameters by reference safely requires the compiler to define when it's safe to assign a reference to a new variable. For every expression, the compiler

defines a *scope* that bounds access to an expression or variable. The compiler uses two scopes: *safe_to_escape* and *ref_safe_to_escape*.

- The *safe_to_escape* scope defines the scope where any expression can be safely accessed.
- The *ref_safe_to_escape* scope defines the scope where a *reference* to any expression can be safely accessed or modified.

Informally, you can think of these scopes as the mechanism to ensure your code never accesses or modifies a reference that's no longer valid. A reference is valid as long as it refers to a valid object or struct. The *safe_to_escape* scope defines when a variable can be assigned or reassigned. The *ref_safe_to_escape* scope defines when a variable can be *ref* assigned or *ref* reassigned. Assignment assigns a variable to a new value; *ref assignment* assigns the variable to *refer to* a different storage location.

Modifiers

Parameters declared for a method without in, ref or out, are passed to the called method by value. The `ref`, `in`, and `out` modifiers differ in assignment rules:

- The argument for a `ref` parameter must be definitely assigned. The called method may reassign that parameter.
- The argument for an `in` parameter must be definitely assigned. The called method can't reassign that parameter.
- The argument for an `out` parameter needn't be definitely assigned. The called method must assign the parameter.

This section describes the keywords you can use when declaring method parameters:

- params specifies that this parameter may take a variable number of arguments.
- in specifies that this parameter is passed by reference but is only read by the called method.
- ref specifies that this parameter is passed by reference and may be read or written by the called method.
- out specifies that this parameter is passed by reference and is written by the called method.

See also

- C# Reference
- C# Keywords

- Argument lists in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.