Contents

Reference Types

Motivation

There is a fundamental difference between *value types* and *reference types* in C#. For example, compare:

```
int x = 10;
int y = x;
y = 11;
Console.WriteLine($"x is {x}, y is {y}.");
// Displays "x is 10, y is 11.".
```

and

```
int[] a = { 10 };
int[] b = a;
b[0] = 11;
Console.WriteLine($"a[0] is {a[0]}, b[0] is {b[0]}.");
// Displays "a[0] is 11, b[0] is 11.".
```

In the first case (with `int`s), the value of x will remain `11`, but in the second (with arrays of `int`s), `a[0]` will now contain `11` as well. That is because when `y = x` was executed, the *value* of x was copied, but when `b = a` is executed, the *reference to the array* was copied.

All the built-in types are *value types*: numerical types, `char` and `bool` contains *values*. On the other hand, objects, `string` and arrays, for example, are *reference types*.

null Value

Reference types can contain a special value, called `null`, that intuitively means that it references nothing. It can be used as follows:

```
int[] c = null;
```

Any reference type must be handled with great care, since for example

```
Console.WriteLine(c.Length);
```

would compile but would throw a `NullReferenceException` exception (a `null` reference doesn't have any `Length` property!).

Three operators allows to simplify testing whenever a variable holds `null` and behave accordingly, we detail them below.

null-Conditional Operator

The null-conditional operator `?` allows to test if a variable holds `null` and to avoid some `NullReferenceException`.

For example,

```
Console.WriteLine($"Length of a is: {a?.Length}.");
```

will display "Length of a is: 1." if a holds a reference to an array of size 1, and "Length of a is: ." if a holds a `null`. Stated differently, `a?.Length` evaluates to the size of the array referenced by a if it exists, to `null` otherwise.

One can similarly write `a?[0]` to either get a `null` (if a itself is `null`) or the value at the first index of the array referenced by a.

null-Coalescing Operator

The null-coalescing operator `??` allows to assign a reference *if it is not null*, and to assign a default value otherwise.

For example,

```
string s1 = null;
string s2 = s1 ?? "nothing";
Console.WriteLine($"s1 is {s1}, s2 is {s2}.");
```

will display "s1 is , s2 is nothing.": the assignment `s2 = s1 ?? "nothing"` "skipped" the value `s1` since it was `null` and used `"nothing"` instead.

null-Coalescing Assignment Operator

The null-coalescing assignment operator `??=` allows to re-assign a variable if it is **`null`**.

For example,

```
s1 ??= "default";
```

will assign `"default"` to `s1` if it is **`null`**, leave its value unchanged otherwise. Note that this operator is available only starting with C# 8.0.

Nullable value types

It is also possible to make a value type *nullable*, so that it can contains the **`null`** value. For example,

```
int[] a = null;
int aLength = a?.Length;
```

is not valid since `a?.Length` will evaluate to **`null`**, and an `int` variable cannot contain a reference!

It is possible, however, to make `aLength` *nullable*, using the `?` operator:

```
int[] a = null;
int? aLength = a?.Length;
```

This way, `aLength` can contain either an integer value, or the **`null`** reference.

To "convert" a nullable value type back into a "non-nullable" value type can be done using the null-coalescing operator `??`. For example,

```
int d = aLength ?? -1;
```

will assign `aLength` to `d` if it is not **`null`**, and `-1` otherwise: note that either way, `d` will end up containing a non-**`null`** value.

Testing for Equality

Motivation

A great care is required when comparing *references*, since one need to make sure that

- **`null`** is accounted for,
- the comparison is "shallow" only if we want it to.

A "shallow" comparison compares only the "surface" of reference variables, as follows:

```
int[] a = { 10 };
int[] b = a;
int[] c = { 10 };

if (a == b){ Console.WriteLine("a and b refers the same
 ↪  array."); }
if (a != c){ Console.WriteLine("a and c refers different
 ↪  arrays."); }
```

Both tests would evaluate to **`true`**, since a and b do indeed refer to the same array, while a and c refer to different arrays. In general, this is not what is intended when comparing objects or arrays: we want to know if *what they refer to* is identical.

Comparing Arrays

To compare arrays while accounting for possible **`null`** values, a great care is needed. One can write a method as follows:

```
public static bool SameArray<T>(T[] arP1, T[] arP2)
{
    if (arP1 == null && arP2 == null) { return true; }
    else if (arP1 == null || arP2 == null) { return false;
     ↪  }
    else if (arP1.Length != arP2.Length) return false;
    else {
        for (int i = 0; i < arP1.Length; i++)
        {
            if (!Equals(arP1[i], arP2[i])) return false;
        }
    }
    return true;
}
```

So that, if `SameArray` is passed…

- … two **`null`** references, it will return **`true`** since, indeed, the arguments refers to "the same" array, which does not exist,
- … a **`null`** reference and a reference that is not **`null`**, it will return **`false`**, as a non-existent array is not the same as an existing array,
- … two arrays of different size, it will return **`false`**,
- … two arrays of the same size, where every single value is the same, it will return **`true`**.

Note that

- for the first two cases, one may decide to use `throw new ArgumentNullException()` instead, because it could be argued comparing **`null`** references is, precisely, shallow.

- it is ok to use `arP1.Length` and `arP2.Length` in our code, since we know at that point that neither `arP1` nor `arP2` is **`null`**.
- we cannot use `if (arP1[i] != arP2[i])` as C# doesn't "know" by default that what we use for T will accept this operator. Instead, we have to use the "generic" `Equals` method.

Passing Arguments

Motivation

Consider the following "swapping" method and a `Main` method calling it:

```
using System;

class Program
{
  static void Main()
  {
    int a = 10;
    int b = 20;
    Console.WriteLine(
      $"Before swap: a holds {a}, b holds {b}."
    );
    Swap(a, b);
    Console.WriteLine(
      $"After swap:  a holds {a}, b holds {b}."
    );
  }

  static void Swap(int a, int b)
  {
    int temp = a;
    a = b;
    b = temp;
    Console.WriteLine(
      $"Inside swap: a holds {a}, b holds {b}."
    );
  }
}
```

This program would display:

```
Before swap: a holds 10, b holds 20.
Inside swap: a holds 20, b holds 10.
After swap:  a holds 10, b holds 20.
```

As we can see, the values held by the variables a and b are correctly swapped by the Swap method, but this change is not "permanent": once the Swap method completed, a and b still have their "old" values inside Main.

Since a method cannot return two values, making that change permanent is difficult. A solution could be designed using arrays for example, but it would require additional manipulation in the Main method. Instead, one can use references to pass *the reference to the variables instead of their values*.

ref Keyword

The **ref** keyword can be used to pass the reference to a variable, as follows:

```
using System;

class Program
{
  static void Main()
  {
    int a = 10;
    int b = 20;
    Console.WriteLine(
      $"Before swap: a holds {a}, b holds {b}."
    );
    Swap(ref a, ref b);
    Console.WriteLine(
      $"After swap:  a holds {a}, b holds {b}."
    );
  }

  static void Swap(ref int a, ref int b)
  {
    int temp = a;
    a = b;
    b = temp;
    Console.WriteLine(
      $"Inside swap: a holds {a}, b holds {b}."
    );
  }
}
```

Note that the change with the previous code is minimal: only the keyword **ref** is added:

- In front of the datatype of the arguments in Swap's header,
- In front of the name of the variables when the Swap method is called.

Note that *both* edits are required: the first one stipulates that the Swap method expects *references*, and the second one stipulates that the *references* are passed.

This program would display:

```
Before swap: a holds 10, b holds 20.
Inside swap: a holds 20, b holds 10.
After swap:  a holds 20, b holds 10.
```

Indeed, since *the reference* was passed, Swap stored the new values *in the same variables a and b*, making the swapping "permanent".

out Keyword

In some cases, one may want to pass a reference to a method simply as an address where a value must be stored. The benefit is that this reference does not need to contain a value before being passed to a method.

For example, consider:

```
static void SetToRandom(ref int a)
{
    Random gen = new Random();
    a = gen.Next(10);
}
```

that sets the value of a reference to a random number between 0 and 9 (both included).

It can**not** be called as follows:

```
int a; // This code will not compile
SetToRandom(ref a);
```

Because C#'s compilation will return the error message "Use of unassigned local variable 'c' ". Indeed, SetToRandom expects the argument to already holds a reference to a value, even if it has no use for it.

A better alternative is to use the **out** keyword:

```
using System;

class Program
{
  static void Main()
```

```
  {
    int a;
    SetToRandom(out a);
    Console.WriteLine(a);
  }

  static void SetToRandom(out int a)
  {
    Random gen = new Random();
    a = gen.Next(10);
  }
}
```

Note that:

- The keyword **out** is similarly added in the header of the method and when the argument is passed,
- The variable a is *not* given a value before being passed to the method.

Summing up, the difference between **ref** and **out** is that **out** does not require the reference to point to an actual value *entering into the method* but *it must hold a value by the time we exit the method*.

To illustrate this last point, observe that

```
static void Dummy(out int a)
{
    Console.WriteLine("Hi!");
}
```

would not compile, as C# would give back a message "The out parameter 'a' must be assigned to before controls leaves the current method": an argument passed using the keyword **out must** be initialized in the body of the method.