Contents

Generic Type Parameter

Introduction

Imagine that you want to write a method that takes as an argument an array and returns an array of the same type, but with the values reversed. You may write the following code:

```
public class Helper{
    public static int[] Reverse(int[] arrayP)
    {
        int[] result = new int[arrayP.Length];
        int j = 0;
        for (int i = arrayP.Length - 1; i >= 0; i--)
        {
            result[j] = arrayP[i];
            j++;
        }
        return result;
    }
}
```

Then, this method could be used as follows:

```
int[] array1 = {0, 2, 3, 6};
int[] array1reversed = Helper.Reverse(array1);
```

And then `array1reversed` would contain 6, 3, 2, 0.

This method works as intended, but you can use it only with arrays of *integers*. If you want to use a similar method with arrays of, say, `char`, then you need to copy-and-paste the code above and to replace every occurrence of `int` by `char`. This is not very efficient, and it is error-prone.

Generic Types

There is a tool in C# to avoid having to be *too* specific, and to be able to tell the compiler that the method will work "with some type", called

generic type parameter[1], using the keyword T. In essence, <T> is affixed after the name of the method to signal that the method will additionally require to instantiate T with a particular type.

The previous method would become:

```
public class Helper{
    public static T[] Reverse<T>(T[] arrayP)
    {
        T[] result = new T[arrayP.Length];
        int j = 0;
        for (int i = arrayP.Length - 1; i >= 0; i--)
        {
            result[j] = arrayP[i];
            j++;
        }
        return result;
    }
}
```

where three occurrences of `int[]` were replaced by `T[]`, and <T> was additionally added between the name of the method and its parameters. This method is used as follows:

```
int[] array1 = {0, 2, 3, 6};
int[] array1reversed = Helper.Reverse<int>(array1);

char[] array2 = {'a', 'b', 'c'};
char[] array2reversed = Helper.Reverse<char>(array2);
```

In essence, `Reverse<int>` tells C# that `Reverse` will be used with T being `int` (not `int[]`, as the method uses `T[]` for its argument and return type). Note that to use *the same method* with `char`, we simply use `Reverse<char>`, and then we provide an array of `char` as parameters, and obtain an array of `char` in return.

Implicitly Typed Local Variables

Sometimes, the body of the method needs to declare variable with the same type as T. Indeed, imagine, for example, that we want to add to our `Helper` class a method that returns a `string` description of an array. We can write the following:

```
public static string Description(int[] arrayP)
{
    string returned = "";
```

[1] https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters

```
    foreach (int element in arrayP)
    {
        returned += element + " ";
    }
    return returned;
}
```

but this method is specific to arrays of `int`, and we would have to write another one for `char`, for example. Making the header generic is "easy", as we can use, as before:

```
public static string Description<T>(T[] arrayP)
```

but the body is problematic: what should be the type of the `element` variable in the header of the **`foreach`**? We cannot simply use T, but we can use *implicitly typed variable*. This technique, that uses the keyword `var` essentially tells C# to … figure out the type of the variable. In that case, since C# knows the type of the array you are passing, it can easily infer the type of its elements.

We can then rewrite the previous method as follows:

```
public static string Description<T>(T[] arrayP)
{
    string returned = "";
    foreach (var element in arrayP)
    {
        returned += element + " ";
    }
    return returned;
}
```

and use it with

```
Console.WriteLine(Helper.Display<char>(array2);
```

for example.