Functional programming in C#

A quick approach to another paradigm

Nacho Iborra

IES San Vicente

Table of Contents

1. Introduction

Functional programming is a programming paradigm (this is, a way of implementing programs) that focuses on the evaluation of mathematical functions. It is based on lambda calculus, a system born in 1930 to evaluate function definition, application and recursion.

Functional paradigm is ***declarative***, this is, code is made with declarations instead of sentences or statements. In other words, we tell the program how things are, instead of how to solve a problem. On the opposite side we have ***imperative*** languages, which are based on commands in the source code, such as assignments, that tell the program how to solve the problem.

1.1. Some functional programming features

Functional programming relies on some fundamental concepts:

- **Referential transparency**: the output of a function depends only on its arguments, so that if we call it many times with the same arguments, we will always get the same result. So, functional programming has no side effects that can modify something out of the function
- **Data immutability**: in order to make sure that no side effects will be produced when calling functions, one of the most important rules of functional programming paradigm is that data can't be mutable.
- **Function composition**: functional languages treat functions as data, so we can apply *function composition*, this is, we can chain the output of a function with the input of next function.
- **First order functions**: these functions are higher level functions, which allow other functions as parameters. They are very popular in languages such as Javascript, when we can define *callbacks* to respond to some events or asynchronous tasks, but they are also an important aspect of functional languages.

Imperative languages can produce side effects in external elements. They also have functions, just like functional programming has, but in this case functions are not mathematical definitions, but a group of sentences that can be called from different parts of the program. Taking into account possible side effects, we might have different results when calling a function many times with the same arguments. Let's have a look at the following code written in C:

```
int externalValue = 1;
int aFunction(int param)
{
      externalValue++;
      return externalValue + param;
}
```

If we call the function as `aFunction(1)`, it will return 3. But if we call it again with the same arguments (`aFunction(1)`) it will return 4... and so on. This kind of situation is forbidden in a functional paradigm, although some functional languages let us commit this "crime".

1.2. Functional languages

If we are looking for a functional language, we must distinguish between:

- **Pure** functional languages, this is, languages that were created to follow this paradigm, and no other. In this group we can find languages like Haskell or Miranda. The first one is used by Facebook to deal with big data and analysis.
- **Hybrid** languages, this is, languages that accept several paradigms, although they do well with functional approaches. In this group we can talk about Scala, Clojure, or Lisp.
- Other languages, that were initially imperative, but have modified their syntax and structure to accept some functional features. This is the case of some of the most important languages of these years, such as Java, C#, Javascript...

1.3. An introductory example

Just to have a first experience with the functional paradigm, let's see how to solve a typical problem. We are going to use a popular language, such as Java, to illustrate it. We have a list of *Person* objects, each one with its own name and age. If we want get just the people that are adults (i.e. their age is greater or equal than 18), we would do it this way with traditional, imperative programming:

```
List<Person> adultPeople = new ArrayList<>();
for (int i = 0; i < people.size(); i++)
{
    if (people.get(i).getAge() >= 18)
        adultPeople.add(people.get(i));
}
```

What we do is explore the original list, and add to a new one every person older than 18. But, if we take advantage of some of the new features provided with Java 8 regarding functional programming, we can solve the same problem like this:

```
List<Persona> adultPeople = people.stream()
                                  .filter(p -> p.getAge() >= 18)
                                  .collect(Collectors.toList());
```

As we can see, code is more compact, less error-prone and (once we get used to the syntax), more understandable. You can also see how function composition works: the output of `stream` method is the input for `filter` method, and the output of this method is the input for `collect`.

2. Functional programming in C#

C# is one of the currently popular programming languages that has adopted some of the most important features of functional paradigm. From version 2.0 (.NET 2.0 framework) we can define anonymous methods, and with version 3.0 (.NET 3.0 or 3.5 framework) lambda expressions were also available. We can also use some additional elements, such as LINQ library, from version 3.0. Let's see some of these aspects in this session.

2.1. Adaptation to functional paradigm

If we pay attention to some of the main features of functional paradigm, we can see how C# has adapted his structure to them:

- **Data immutability:** C# is still an imperative language, so we can't expect that data is immutable. Global variables can be modified from particular methods or functions.
- **Referential transparency**: because of data mutability, referential transparency is not guaranteed in C#. So the programmer must make an effort to make this possible.
- **Function composition:** you can chain function calls easily with C#
- **First order functions:** you can also define functions that allow other functions as parameters, as we will see in some examples.

2.2. Anonymous functions in C#

An anonymous function is a function without name, that is immediatly used (called) at the same place where it is defined, or assigned to a variable for later use. Let's see how this feature has evolved in C# from version 2.0.

2.2.1. The origin: delegates

A **delegate** is some kind of signature of a function. It determines the return type and parameters, but not the code, so any function with the same return type and parameters can be used instead of the delegate.

For instance, we can define a delegate that returns an integer and takes a string as a parameter:

```
public delegate int MyDelegate(string param);
```

Every method that has this specification can be used as delegate. So if we define this method:

```
public static int MyMethod(string param)
{
    return param.Length;
}
```

We can define a delegate variable and assign it to this method. This way, if we call the delegate, we will be running the method, actually.

```
MyDelegate del1 = MyMethod;
Console.WriteLine(del1("Hello"));         // Would show 5 (length of "Hello")
```

We can also use delegates as parameters of other functions (first order functions). Here we can see a function whose last parameter is a delegate:

```
public static void MyFunctionWithDelegate(string text, MyDelegate del)
{
    Console.WriteLine("The result is {0}", del(text));
}
```

If we call this function, we can use the delegate created in previous example:

```
MyFunctionWithDelegate("Hello world", del1); // Will show "The result is 11"
```

2.2.2. Anonymous functions

With C# 2.0 we could also define anonymous functions or methods. This way, we can assign a block of code to a delegate, without creating a conventional function. Regarding previous example:

```
public delegate int MyDelegate(string param);
```

Instead of defining *MyMethod* method and associating it to the delegate, we can just implement the code of the delegate and assign it:

```
MyDelegate del2 = delegate(string param)
{
   return param.length;
};
```

and then we can use the delegate wherever we want:

```
Console.WriteLine(del2("Hello"));         // Will also show 5 as result
```

This way, our code is more compact and we don't need to define conventional methods to be used as delegates.

2.2.3. Lambda expressions

Lambda expressions are a compact way of defining functions. They were introduced in C# 3.0 and it is the preferred way of adding in-line code currently (this is, adding the code of a function just where it is actually needed).

Lambda expressions are usually defined to act as delegates. Regarding our delegate from previous examples:

```
public delegate int MyDelegate(string param);
```

We can define a lambda expression that takes a string as parameter and returns an integer (although data types are not specified in the definition of the lambda expression):

```
MyDelegate del3 = (param) => param.length;

Console.WriteLine(del3("Hello")); // Will show 5
```

Take a look at the syntax: First of all, we define the parameters within parentheses, and separated by commas if there are more than one parameter. We can also put empty parentheses if the expression does not need any parameter, and we can omit the parentheses if there is only one parameter. Then, there is an "arrow" and the code of the function. If the code just returns a value, we can put the value to be returned. Otherwise, we must use brackets as we do with a traditional function:

```
MyDelegate del4 = param => {
    Console.WriteLine("Param: " + param);
    Console.WriteLine("Result: " + param.Length);
```

```
    return param.length;
};
```

2.3. Usefulness of lambda expressions and delegates

After all these concepts that we have learnt, we could think that delegates or lambda expressions are quite useless. We can do the same things with traditional functions. However, delegates are used by C# to pass functions as parameters of other functions (called *first order functions*), and this is particularly useful in some scopes:

- When working with Windows applications (Windows Forms, for instance), you will find a lot of event handlers that need a delegate to determine the function to be run. For instance, if we want to call a given function when clicking a button, the final code may look like this:

  ```
  button.Click += new EventHandler(delegateMethod);
  ```

  where `EventHandler` is a delegate that defines the events of the applications. So we just need to define a method that follows the correct specification, and use it as a parameter in this piece of code.

- Also, when we are working with threads in Windows applications, we will not be able to access the graphical contents from a thread. In other words, we can't update a list or a label from a thread. To solve this problem, we can call a delegate defined in the main program from the thread, so that this delegate can modify the graphical contents that we want to change.

But let's have a look at some other scopes where delegates can be also useful, and may be more "familiar".

2.3.1. Example: sorting objects

Let's create a class called `Person`, with a name and age:

```
class Person
{
    string name;
    int age;

    // ... constructors getters, setters and other methods
}
```

If we want to sort a list of *Person* objects by their ages (in ascending order, for instance), we can do it in two ways:

- Manually applying some sorting algorithm (bubble sort, quick sort...)
- Defining a sort criteria, and letting the list to sort itself by calling `Sort` method.

The first approach is a typical *imperative* approach (we tell the program how to sort each element), whereas the second one is a *declarative* approach (we only tell what criteria must be accomplished by all the people in the list in order to be sorted). To do this, there is a delegate in C# called `Comparison`, that can be implemented to define the comparison criteria. This delegate takes two arguments (of the data type that we want to compare), and returns an integer indicating if the first argument is lower (<0) equal (0) or greater (>0) than the second one.

Regarding our `Person` list, we could define an implementation of this delegate this way, to sort people by age in ascending order:

```
public static int ComparePeople(Person p1, Person p2)
{
  return p1.Age - p2.Age;
}
```

And we could sort a list of `Person` objects like this:

```
List<Person> people = new List<Person>();
…
people.Sort(ComparePeople);
```

But we can also define the delegate "on the fly" while we are sorting the list, by using a lambda expression instead of the previous function:

```
people.Sort((p1, p2) => p1.Age - p2.Age);
```

2.3.2. Example: checking with predicates

There are some more scopes in which delegates and lambda expressions are really welcome. For instance, we can easily check if any (or some) elements of a list match a given criteria. To do this, there is another delegate called `Predicate` in C#. It gets an object as a parameter and returns a boolean indicating whether the object meets some conditions or not.

We can check if there is any adult in the list of people used above, this way:

```
public static bool CheckAdult(Person p)
{
    return p.Age >= 18;
}
…
Console.WriteLine("Is there any adult? {0}", people.Exists(CheckAdult));
```

and also (preferably) with a lambda expression:

```
Console.WriteLine("Is there any adult? {0}",
                  people.Exists(p => p.Age >= 18));
```

There are some methods in `List` class that use predicates as parameters. For instance, we can get all the adult people from the list with `FindAll` method. It returns a list with all the elements that meet the condition(s) of the predicate.

```
List<Person> adults = people.FindAll(p => p.Age >= 18);
```

2.4. Using LINQ to deal with collections

LINQ (*Language-INtegrated Query*) is a powerful library to manipulate collections and filter, sort or map some data on them. It has a syntax similar to SQL, but it also has some methods based on predicates and other features to filter and transform the information contained in those collections.

Let's go back to the *Person* list shown above. If we want to filter adult people using LINQ, we can have a code like this:

```
IEnumerable<Person> adultPeople = from person in people
                                  where person.Age >= 18
                                  select person;
```

> ***NOTE**: in order to use LINQ syntax and methods, we must add the corresponding `using` sentence to our source code (`System.Linq`).*

This result can be explored with a `foreach`:

```
foreach(Person p in adultPeople)
    Console.WriteLine(p.Name);
```

Besides, `IEnumerable` interface provides some useful methods, such as `Where`, that lets us perform more checkings and filters over the resulting subset of data. These methods usually get a predicate as parameter, that can be defined as a lambda expression. This way we can filter adult people whose name starts with an "A":

```
IEnumerable<Person> adultPeopleStartingWithA =
    adultPeople.Where(p => p.Name.StartsWith("A"));

foreach(Person p in adultPeopleStartingWithA)
    Console.WriteLine(p.Name);
```

We can also map some information of the result. For instance, here we only get the age of each chosen person, through the *select* clause:

```
IEnumerable<Int32> adultAges = from person in people
                               where person.Age >= 18
                               select person.Age;
```

Finally, there are also some reduction methods, such as `Aggregate`, `Average`, `Min` or `Max` that take the whole list and calculate a given value (average, maximum value, minimum value...). This way we calculate the age average for all the adult people

```
Console.WriteLine("Average age for adult people is {0}",
                  adultAges.Average());
```

3. Exercises

These exercises must be implemented in a project called "FunctionalCSharp".

Define a class called *Hotel*, with three attributes: the hotel name (*string*), the hotel location (a city name) and the hotel rating (a floating point value between 0 and 5, both included). Define the constructor to set those values, and the corresponding get/set properties.

Then, in the main program, create a list of hotels (at least with 5 hotels) and do the following:

- Sort the hotels by rating in descending order using a lambda expression, and show the result in the console.
- Get all the hotels whose rating is greater than 3 using `FindAll` method with a lambda expression. Show the corresponding sublist in the console.
- Use LINQ to get the same sublist (hotels whose rating is greater than 3). Show the result in the console.
- Use LINQ to get the rating average for all the hotels in Alicante. Show the result in the console