The Functional Style

We'll cover the following

- What's the functional style?
- Why and when to use functional style

The functional style of programming has been around for a very long time, although it hasn't been the mainstream. In recent years, it's been gaining in popularity. Let's first look at what it is, and then why that's significant.

What's the functional style?

"Computers are stupid; you have to tell them every detail," were the words of my teenager learning to program. Sadly, feeling that pain every single day is the job of most professional programmers. Anything that can alleviate those troubles is a step in the right direction. Functional programming is one such solution.

In the imperative style of programming, we have to key in every step. Create a variable named `i`, set the initial value to `0`, check if the value is less than `k`—wait, is it less than or less than or equal to? Hmm.

The declarative style is where you say what you want and let the functions you call figure out how to do it. For example, to find if our friend Nemo exists in a list of names, the imperative style will demand iterating over each element to compare the values at each position in the collection. In the declarative style, we'll call a `contains()` method and be done quickly.

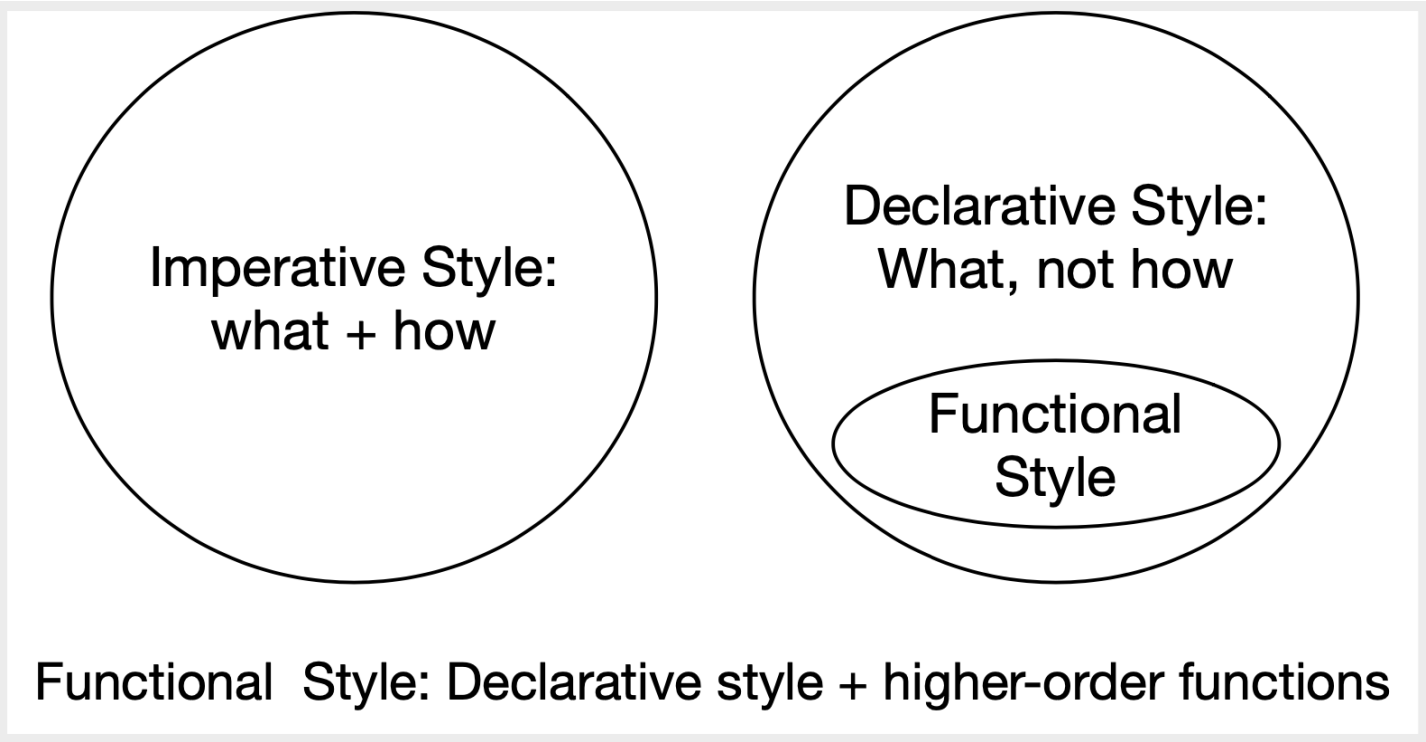
"How does that method work?" is a reasonable question. The short answer is "We don't care," but that sounds rather rude—let's rephrase: "It's encapsulated," meaning we shouldn't care. But as programmers, we know that such details are important.

The difference is that in the imperative style, the details are in your face all the time, whether you need them or not. In the declarative style, they are encapsulated

in a layer below. You can seek the details, at your will, anytime you deem it important and not be bothered at other times.

Functional style builds on declarative style. It mixes the expressive nature of the declarative style with higher-order functions. We're used to passing objects to functions and returning objects from functions. Higher-order functions may receive functions as parameters and/or may return functions as a result. We can nicely compose a chain of function calls—known as functional composition—to accomplish tasks, and that leads to fluent and easier-to-understand code.

The imperative style is familiar to most programmers, but it can get complex and hard to read. Part of the reason is that it contains too many moving parts. The following small example illustrates that point.




```
var doubleOfEven = mutableListOf<Int>()

for (i in 1..10) {
    if (i % 2 == 0) {
        doubleOfEven.add(i * 2)
    }
}

println(doubleOfEven) //[4, 8, 12, 16, 20]
```

doubleofeven.kts

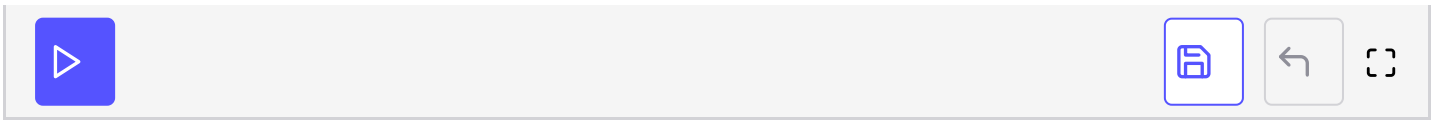
To compute the double of all even numbers in the range `1` to `10`, we first created

an empty mutable list. Then we picked each value in the collection, checked if the value is even, and if so, added the double of the value to the list.

The functional style is less familiar to a lot of programmers, but it's simple. We can use higher-order functions, `filter()` and `map()`, on the `IntRange` class to achieve the same thing in the functional style. We can pass lambdas to these functions, as you can see in this next example. Ignore the syntax for now and focus on the concept; we'll explore the syntax shortly.

```
val doubleOfEven = (1..10)
    .filter { e -> e % 2 == 0}
    .map { e -> e * 2 }

println(doubleOfEven) //[4, 8, 12, 16, 20]
```



doubleofeven.kts

Instead of creating a mutable list and repeatedly adding values to it, we have a better flow of logic in this code. Given a range of values from 1 to 10, we filter (or pick) only even values, and then map (or transform) the ones picked to double their values. The result is an immutable list of values. And during the execution of code, we didn't use any explicit mutable variables—that's fewer moving parts.

Before getting deep into the functional style, let's discuss the reasons, why and when, this is a better approach when compared to the imperative style.

Why and when to use functional style

Declarative style is taking root beyond programming; the world is embracing this style.

For example, look at cars—when rental car agencies ask me what kind I'd like to rent, I say, "Give me one with four wheels, please." Driving a stick-shift car is the equivalent of the imperative style of programming—though you'll see me use imperative style from time to time, you'll never catch me driving a stick-shift. Auto-transmission is like adding fluency to imperative style. I'm more of an Uber or Lyft kind of person—code away in the back seat while the driver takes care of getting me to the destination. As a bonus, the streets are safer with me not behind the wheel.

wheel.

As the world is heading toward autonomous vehicles so we can focus on where to go, not how to get there, the programming world is likewise heading in the direction of being more declarative. We see this with frameworks and libraries that remove the drudgery of programming using low-level APIs.

As you ease into the functional style, which is declarative in nature, you may realize:

- Imperative style is familiar, but complex; it's easier to write, due to our familiarity, but is hard to read.
- Functional style is less familiar, but simpler; it's harder to write, due to our unfamiliarity, but is easier to read.

Once you get comfortable with the functional style, you'll see that it has less complexity baked in compared to imperative-style code. And it's much easier to make functional code concurrent than to make imperative code concurrent. It's less complex, easier to change, and safer to run concurrently.

The functional style is less complex, but that doesn't make it better than imperative style all the time. Use the functional style when the code is mostly focused on computations—you can avoid mutability and side effects—and for the problems that can be expressed in terms of series of transformations. But if the problem at hand involves a lot of IO, has unavoidable state mutation or side effects, or if code will have to deal with many levels of exceptions, then imperative style may be a better option. We'll see this aspect further in Chapter 17, Asynchronous Programming.

In the next lesson, let's look at the structure of lambdas, including how to create and use them.