Functional Programming in Java

How functional techniques improve your Java programs

Pierre-Yves Saumont

MANNING

SAMPLE CHAPTER

Functional Programming in Java

by Pierre-Yves Saumont

Chapter 1

brief contents

1 What is functional programming?

This chapter covers

- The benefits of functional programming
- Problems with side effects
- How referential transparency makes programs safer
- Reasoning about programs with the substitution model
- Making the most of abstraction

Not everybody agrees on a definition for functional programming (FP). In general terms, functional programming is a programming paradigm, and it's about programming with functions. But this doesn't explain the most important aspect: how FP is different from other paradigms, and what makes it a (potentially) better way to write programs. In his article "Why Functional Programming Matters," published in 1990, John Hughes writes the following:

> Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant—since no side effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa—that is, programs are "referentially transparent." This freedom helps make functional programs more tractable mathematically than their conventional counterparts.[1]

In the rest of this chapter, I'll briefly present concepts such as referential transparency and the substitution model, as well as other concepts that together are the essence of functional programming. You'll apply these concepts over and over in the coming chapters.

1.1 What is functional programming?

It's often as important to understand what something is not, as to agree about what it is. If functional programming is a programming paradigm, there clearly must be other programming paradigms that FP differs from. Contrary to what some might think, functional programming isn't the opposite of object-oriented programming (OOP). Some functional programming languages are object-oriented; some are not.

Functional programming is sometimes considered to be a set of techniques that supplement or replace techniques found in other programming paradigms, such as

- First-class functions
- Anonymous functions
- Closures
- Currying
- Lazy evaluation
- Parametric polymorphism
- Algebraic data types

Although it is true that most functional languages do use a number of these techniques, you may find, for each of them, examples of functional programming languages that don't, as well as non-functional languages that do. As you'll see when studying each of these techniques in this book, it's not the language that makes programming functional. It's the way you write the code. But some languages are more *functional-friendly* than others.

What functional programming may be opposed to is the imperative programming paradigm. In imperative programming style, programs are composed from elements that "do" something. "Doing" something generally implies an initial state, a transition,

[1] John Hughes, "Why Functional Programming Matters," from D. Turner, ed., *Research Topics in Functional Programming* (Addison-Wesley, 1990), 17–42, www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

and an end state. This is sometimes called *state mutation*. Traditional imperative-style programs are often described as a series of mutations, separated with condition testing. For example, an addition program for adding two positive values `a` and `b` might be represented by the following pseudo code:

- `if b == 0, return a`
- `else increment a and decrement b`
- `start again with the new a and b`

In this pseudo code, you can recognize the traditional instructions of most imperative languages: testing conditions, mutating variables, branching, and returning a value. This code may be represented graphically by a flow chart, such as figure 1.1.

On the other hand, functional programs are composed of elements that "are" something—they don't "do" something. The addition of `a` and `b` doesn't "make" a result. The addition of 2 and 3, for example, doesn't *make* 5. It *is* 5.

The difference might not seem important, but it is. The main consequence is that each time you encounter 2 + 3, you can replace it with 5. Can you do the same thing in an imperative program? Well, sometimes you can. But sometimes you can't without changing the program's outcome. If the expression you want to replace has no other effect than returning the result, you can safely replace it with its result. But how can you be sure that it has no other effect? In the addition example, you clearly see that the two variables `a` and `b` have been destroyed by the program. This is an effect of the program, besides returning the result, so it's called a *side effect*. (This would be different if the computation were occurring inside a Java method, because the variables `a` and `b` would be passed by value, and the change would then be local and not visible from outside the method.)

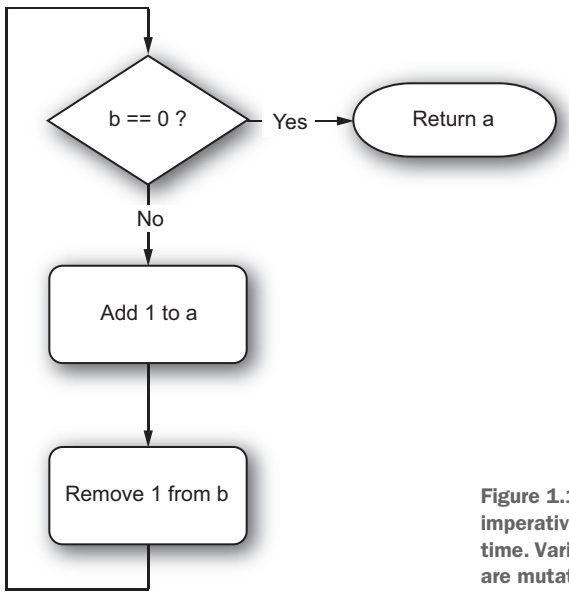


Figure 1.1 A flow chart representing an imperative program as a process that occurs in time. Various things are transformed and states are mutated until the result is obtained.

One major difference between imperative programming and FP is that in FP there are no side effects. This means, among other things,

- No mutation of variables
- No printing to the console or to any device
- No writing to files, databases, networks, or whatever
- No exception throwing

When I say "no side effects," I mean no observable side effects. Functional programs are built by composing *functions* that take an argument and return a value, and that's it. You don't care about what's happening *inside* the functions, because, in theory, nothing is happening ever. But in practice, programs are written for computers that aren't functional at all. All computers are based on the same imperative paradigm; so functions are black boxes that

- Take an argument (a *single* one, as you'll see later)
- Do mysterious things inside, such as mutating variables and a lot of imperative-style stuff, but with no effect observable from outside
- Return a (single) value

This is theory. In practice, it's impossible for a function to have no side effects at all. A function will return a value at some time, and this time may vary. This is a side effect. It might create an out-of-memory error, or a stack-overflow error, and crash the application, which is a somewhat observable side effect. And it will cause writing to memory, registering mutations, thread launching, context switching, and other sorts of things that are indeed effects observable from outside.

So functional programming is writing programs with no *intentional side effects*, by which I mean side effects that are part of the expected outcome of the program. There should also be as few non-intentional side effects as possible.

1.2 Writing useful programs with no side effects

You may wonder how you can possibly write useful programs if they have no side effects. Obviously, you can't. Functional programming is not about writing programs that have no observable results. It's about writing programs that have no observable results other than returning a value. But if this is all the program does, it won't be very useful. In the end, functional programs have to have an observable effect, such as displaying the result on a screen, writing it to a file or database, or sending it over a network. This interaction with the outside world won't occur in the middle of a computation, but only when you finish the computation. In other words, side effects will be delayed and applied separately.

Take the example of the addition in figure 1.1. Although it's described in imperative style, it might yet be functional, depending on how it's implemented. Imagine this program is implemented in Java as follows:

```
public static int add(int a, int b) {
  while (b > 0) {
    a++;
    b--;
  }
  return a;
}
```

This program is fully functional. It takes an argument, which is the pair of integers `a` and `b`, it returns a value, and it has absolutely no other observable effect. That it mutates variables doesn't contradict the requirements, because arguments in Java are passed by value, so the mutations of the arguments aren't visible from outside. You can then choose to apply an effect, such as displaying the result or using the result for another computation.

Note that although the result might not be correct (in case of an arithmetic overflow), that's not in contradiction with having no side effects. If values `a` and `b` are too big, the program will silently overflow and return an erroneous result, but this is still functional. On the other hand, the following program is not functional:

```
public static int div(int a, int b) {
  return a / b;
}
```

Although this program doesn't mutate any variables, it throws an exception if `b` is equal to `0`. Throwing an exception is a side effect. In contrast, the following implementation, although a bit stupid, is functional:

```
public static int div(int a, int b) {
  return (int) (a / (float) b);
}
```

This implementation won't throw an exception if `b` is equal to `0`, but it will return a special result. It's up to you to decide whether it's OK or not for your function to return this specific result to mean that the divisor was `0`. (It's probably not!)

Throwing an exception might be an intentional or unintentional side effect, but it's always a side effect. Often, though, in imperative programming, side effects are wanted. The simplest form might look like this:

```
public static void add(int a, int b) {
  while (b > 0) {
    a++;
    b--;
  }
  System.out.println(a);
}
```

This program doesn't return a value, but it prints the result to the console. This is a desired side effect.

Note that the program could alternatively both return a value and have some intentional side effects, as in the following example:

```
public static int add(int a, int b) {
  log(String.format("Adding %s and %s", a, b));
  while (b > 0) {
    a++;
    b--;
  }
  log(String.format("Returning %s", a));
  return a;
}
```

This program isn't functional because it uses side effects for logging.

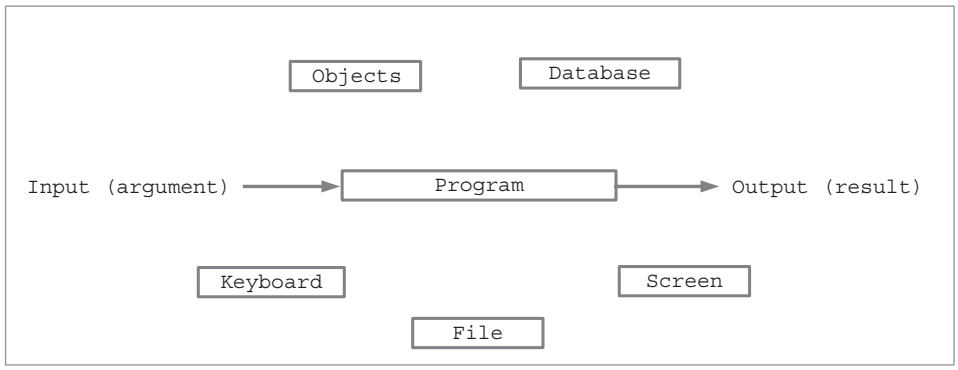
1.3 How referential transparency makes programs safer

Having no side effects (and thus not mutating anything in the external world) isn't enough for a program to be functional. Functional programs must also not be affected by the external world. In other words, the output of a functional program must depend only on its argument. This means functional code may not read data from the console, a file, a remote URL, a database, or even from the system. Code that doesn't mutate or depend on the external world is said to be referentially transparent.

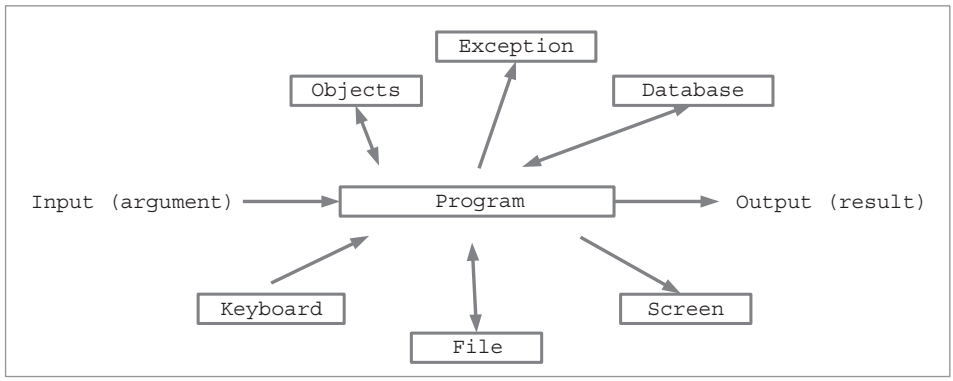
Referentially transparent code has several properties that might be of some interest to programmers:

- It's self-contained. It doesn't depend on any external device to work. You can use it in any context—all you have to do is provide a valid argument.
- It's deterministic, which means it will always return the same value for the same argument. With referentially transparent code, you won't be surprised. It might return a wrong result, but at least, for the same argument, this result will never change.
- It will never throw any kind of `Exception`. It might throw errors, such as OOME (out-of-memory error) or SOE (stack-overflow error), but these errors mean that the code has a bug, which is not a situation you, as a programmer, or the users of your API, are supposed to handle (besides crashing the application and eventually fixing the bug).
- It won't create conditions causing other code to unexpectedly fail. For example, it won't mutate arguments or some other external data, causing the caller to find itself with stale data or concurrent access exceptions.
- It won't hang because some external device (whether database, file system, or network) is unavailable, too slow, or simply broken.

Figure 1.2 illustrates the difference between a referentially transparent program and one that's not referentially transparent.



A referentially transparent program doesn't interfere with the outside world apart from taking an argument as input and outputting a result. Its result only depends on its argument.



A program that isn't referentially transparent may read data from or write it to elements in the outside world, log to file, mutate external objects, read from keyboard, print to screen, and so on. Its result is unpredictable.

Figure 1.2 Comparing a program that's referentially transparent to one that's not

1.4 The benefits of functional programming

From what I've just said, you can likely guess the many benefits of functional programming:

- Functional programs are easier to reason about because they're deterministic. One specific input will always give the same output. In many cases, you might be able to prove your program correct rather than extensively testing it and still being uncertain whether it will break under unexpected conditions.
- Functional programs are easier to test. Because there are no side effects, you don't need mocks, which are generally required to isolate the programs under test from the outside.

- Functional programs are more modular because they're built from functions that have only input and output; there are no side effects to handle, no exceptions to catch, no context mutation to deal with, no shared mutable state, and no concurrent modifications.
- Functional programming makes composition and recombination much easier. To write a functional program, you have to start by writing the various base functions you need and then combine these base functions into higher-level ones, repeating the process until you have a single function corresponding to the program you want to build. As all these functions are referentially transparent, they can then be reused to build other programs without any modifications.

Functional programs are inherently thread-safe because they avoid mutation of shared state. Once again, this doesn't mean that all data has to be immutable. Only shared data must be. But functional programmers will soon realize that immutable data is always safer, even if the mutation is not visible externally.

1.5 Using the substitution model to reason about programs

Remember that a function doesn't *do* anything. It only has a value, which is only dependent on its argument. As a consequence, it's always possible to replace a function call, or any referentially transparent expression, with its value, as shown in figure 1.3.

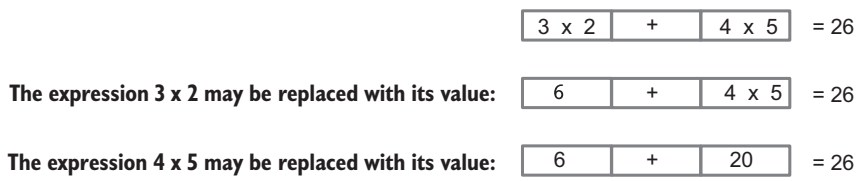


Figure 1.3 Replacing referentially transparent expressions with their values doesn't change the overall meaning.

When applied to functions, the substitution model allows you to replace any function call with its return value. Consider the following code:

```
public static void main(String[] args) {
  int x = add(mult(2, 3), mult(4, 5));
}
public static int add(int a, int b) {
  log(String.format("Returning %s as the result of %s + %s", a + b, a, b));
  return a + b;
}
public static int mult(int a, int b) {
  return a * b;
}
```

```
public static void log(String m) {
  System.out.println(m);
}
```

Replacing `mult(2, 3)` and `mult(4, 5)` with their respective return values doesn't change the signification of the program:

```
int x = add(6, 20);
```

In contrast, replacing the call to the `add` function with its return value changes the signification of the program, because the `log` method will no longer be called, and no logging will happen. This might be important or not; in any case, it changes the result of the program.

1.6 Applying functional principles to a simple example

As an example of converting an imperative program into a functional one, we'll consider a very simple program representing the purchase of a donut with a credit card.

Listing 1.1 A Java program with side effects

```
public class DonutShop {

  public static Donut buyDonut(CreditCard creditCard) {
    Donut donut = new Donut();
    creditCard.charge(Donut.price);      <-- 1 Charges the credit card as a side effect
    return donut;                        <-- 2 Returns the donut
  }
}
```

In this code, the charging of the credit card is a side effect ❶. Charging a credit card probably consists of calling the bank, verifying that the credit card is valid and authorized, and registering the transaction. The function returns the donut ❷.

The problem with this kind of code is that it's difficult to test. Running the program for testing would involve contacting the bank and registering the transaction using some sort of mock account. Or you'd need to create a mock credit card to register the effect of calling the `charge` method and to verify the state of the mock after the test.

If you want to be able to test your program without contacting the bank or using a mock, you should remove the side effect. Because you still want to charge the credit card, the only solution is to add a representation of this operation to the return value. Your `buyDonut` method will have to return both the donut and this representation of the payment.

To represent the payment, you can use a `Payment` class.

Listing 1.2 The `Payment` class

```
public class Payment {

  public final CreditCard creditCard;
  public final int amount;
```

```
  public Payment(CreditCard creditCard, int amount) {
    this.creditCard = creditCard;
    this.amount = amount;
  }
}
```

This class contains the necessary data to represent the payment, which consists of a credit card and the amount to charge. Because the `buyDonut` method must return both a `Donut` and a `Payment`, you could create a specific class for this, such as `Purchase`:

```
public class Purchase {

  public Donut donut;
  public Payment payment;

  public Purchase(Donut donut, Payment payment) {
    this.donut = donut;
    this.payment = payment;
  }
}
```

You'll often need such a class to hold two (or more) values, because functional programming replaces side effects with returning a representation of these effects.

Rather than creating a specific `Purchase` class, you'll use a generic one that you'll call `Tuple`. This class will be parameterized by the two types it will contain (`Donut` and `Payment`). The following listing shows its implementation, as well as the way it's used in the `DonutShop` class.

Listing 1.3 The `Tuple` class

```
public class Tuple<T, U> {

  public final T _1;
  public final U _2;

  public Tuple(T t, U u) {
    this._1 = t;
    this._2 = u;
  }
}
public class DonutShop {

  public static Tuple<Donut, Payment> buyDonut(CreditCard creditCard) {
    Donut donut = new Donut();
    Payment payment = new Payment(creditCard, Donut.price);
    return new Tuple<>(donut, payment);
  }
}
```

Note that you're no longer concerned (at this stage) with how the credit card will actually be charged. This adds some freedom to the way you build your application. You could still process the payment immediately, or you could store it for later processing. You could even combine stored payments for the same card and process them

in a single operation. This would allow you to save money by minimizing the bank fees for the credit card service.

The `combine` method in the following listing allows you to combine payments. Note that if the credit cards don't match, an exception is thrown. This doesn't contradict what I said about functional programs not throwing exceptions. Here, trying to combine two payments with two different credit cards is considered a bug, so it should crash the application. (This isn't very realistic. You'll have to wait until chapter 7 to learn how to deal with such situations without throwing exceptions.)

Listing 1.4 Composing multiple payments into a single one

```
package com.fpinjava.introduction.listing01_04;

public class Payment {

  public final CreditCard creditCard;
  public final int amount;

  public Payment(CreditCard creditCard, int amount) {
    this.creditCard = creditCard;
    this.amount = amount;
  }

  public Payment combine(Payment payment) {
    if (creditCard.equals(payment.creditCard)) {
      return new Payment(creditCard, amount + payment.amount);
    } else {
      throw new IllegalStateException("Cards don't match.");
    }
  }
}
```

Of course, the `combine` method wouldn't be very efficient for buying several donuts at once. For this use case, you could simply replace the `buyDonut` method with `buyDonuts(int n, CreditCard creditCard)`, as shown in the following listing. This method returns a `Tuple<List<Donut>, Payment>`.

Listing 1.5 Buying multiple donuts at once

```
package com.fpinjava.introduction.listing01_05;

import static com.fpinjava.common.List.fill;
import com.fpinjava.common.List;
import com.fpinjava.common.Tuple;

public class DonutShop {

  public static Tuple<Donut, Payment> buyDonut(final CreditCard cCard) {
    return new Tuple<>(new Donut(), new Payment(cCard, Donut.price));
  }

  public static Tuple<List<Donut>, Payment> buyDonuts(final int quantity,
                                                      final CreditCard cCard) {
    return new Tuple<>(fill(quantity, () -> new Donut()),
                                      new Payment(cCard, Donut.price * quantity));
```

```
  }
}
```

Note that this method doesn't use the standard `java.util.List` class because that class doesn't offer some of the functional methods you'll need. In chapter 3, you'll see how to use the `java.util.List` class in a functional way by writing a small functional library. Then, in chapter 5, you'll develop a completely new functional `List`. It's this list that's used here. This `combine` method is somewhat equivalent to the following, which uses the standard Java list:

```
public static Tuple<List<Donut>, Payment> buyDonuts(final int quantity,
                                                    final CreditCard cCard) {
    return new Tuple<>(Collections.nCopies(quantity, new Donut()),
                       new Payment(cCard, Donut.price * quantity));
  }
```

As you'll soon need additional functional methods, you won't be using the Java list. For the time being, you just need to know that the `static List<A> fill(int n, Supplier<A> s)` method creates a list of `n` instances of `A` by using a special object, `Supplier<A>`. As its name indicates, a `Supplier<A>` is an object that supplies an `A` when its `get()` method is called. Using a `Supplier<A>` instead of an `A` allows for *lazy evaluation*, which you'll learn about in the next chapters. For now, you may think of it as a way to manipulate an `A` without effectively creating it until it's needed.

Now, your program can be tested without using a mock. For example, here's a test for the method `buyDonuts`:

```
@Test
public void testBuyDonuts() {
  CreditCard creditCard = new CreditCard();
  Tuple<List<Donut>, Payment> purchase = DonutShop.buyDonuts(5, creditCard);
  assertEquals(Donut.price * 5, purchase._2.amount);
  assertEquals(creditCard, purchase._2.creditCard);
}
```

Another benefit of making your program functional is that it's more easily composable. If the same person made several purchases with your initial program, you'd have to contact the bank (and pay the corresponding fee) each time. With the new functional version, you can choose to charge the card immediately for each purchase or to group all payments made with the same card and charge it only once for the total.

To group payments, you'll need to use additional methods from your functional `List` class (you don't need to understand how these methods work for now; you'll study them in detail in chapters 5 and 8):

```
public <B> Map<B, List<A>> groupBy(Function<A, B> f)
```

This instance method of the `List` class takes a function from `A` to `B` and returns a map of key and value pairs, with keys being of type `B` and values of type `List<A>`. In other words, it groups payments by credit cards:

```
List<A> values()
```

This is an instance method of `Map` that returns a list of all the values in the map:

```
<B> List<B> map(Function<A, B> f)
```

This is an instance method of `List` that takes a function from `A` to `B` and applies it to all elements of a list of `A`, giving a list of `B`:

```
Tuple<List<A1>, List<A2>> unzip(Function<A, Tuple<A1, A2>> f)
```

This is a method of the `List` class that takes as its argument a function from `A` to a tuple of values. For example, it might be a function that takes an email address and returns the name and the domain as a tuple. The `unzip` method, in that case, would return a tuple of a list of names and a list of domains.

```
A reduce(Function<A, Function<A, A>> f)
```

This method of `List` uses an operation to reduce the list to a single value. This operation is represented by `Function<A, Function<A, A>> f`. This notation may look a bit weird, but you'll learn what it means in chapter 2. It could be, for example, an addition. In such a case, it would simply mean a function such as `f(a, b) = a + b`.

Using these methods, you can now create a new method that groups payments by credit card.

Listing 1.6 Grouping payments by credit card

```
package com.fpinjava.introduction.listing01_06;

import com.fpinjava.common.List;

public class Payment {

  public final CreditCard creditCard;
  public final int amount;

  public Payment(CreditCard creditCard, int amount) {
    this.creditCard = creditCard;
    this.amount = amount;
  }

  public Payment combine(Payment payment) {
    if (creditCard.equals(payment.creditCard)) {
      return new Payment(creditCard, amount + payment.amount);
    } else {
      throw new IllegalStateException("Cards don't match.");
    }
  }
```

```
public static List<Payment> groupByCard(List<Payment> payments) {
  return payments
      .groupBy(x -> x.creditCard)
      .values()
      .map(x -> x.reduce(c1 -> c2 -> c1.combine(c2)));
}
}
```

Changes a List<Payment> into a Map<CreditCard, List<Payment>> where each list contains all payments for a particular credit card

Reduces each List<Payment> into a single Payment, leading to the overall result of a List<Payment>

Changes the Map<CreditCard, List<Payment>> into a List<List<Payment>>

Note that you could use a method reference in the last line of the `groupByCard` method, but I chose the lambda notation because it's probably (much) easier to read. If you prefer method references, you can replace this line with the following one:

```
.map(x -> x.reduce(c1 -> c1::combine));
```

In listing 1.6, the portion after `c1 ->` is a function taking a single parameter and passing that parameter to `c1.combine()`. And that's exactly what `c1::combine` is—it's a function taking a single parameter. Method references are often easier to read than lambdas, but not always!

1.7 Pushing abstraction to the limit

As you've seen, functional programming consists in writing programs by composing pure functions, which means functions without side effects. These functions may be represented by methods, or they may be *first-class functions*, such as the arguments of methods `groupBy`, `map`, or `reduce`, in the previous example. First-class functions are simply functions represented in such a way that, unlike methods, they can be manipulated by the program. In most cases, they're used as arguments to other functions, or to methods. You'll learn in chapter 2 how this is done.

But the most important notion here is abstraction. Look at the `reduce` method. It takes as its argument an operation, and uses it to reduce a list to a single value. Here, the operation has two operands of the same type. Except for this, it could be any operation. Consider a list of integers. You could write a `sum` method to compute the sum of the elements; you could write a `product` method to compute the product of the elements; or you could write a `min` or a `max` method to compute the minimum or the maximum of the list. But you could also use the `reduce` method for all these computations. This is abstraction. You abstract the part that is common to all operations in the `reduce` method, and you pass the variable part (the operation) as an argument.

But you could go further. The `reduce` method is a particular case of a more general method that might produce a result of a different type than the elements of the list. For example, it could be applied to a list of characters to produce a `String`. You'd need to start from a given value (probably an empty string). In chapters 3 and 5, you'll learn how to develop this method (called `fold`). Also note that the `reduce` method

won't work on an empty list. Think of a list of integers—if you want to compute the sum, you need to have an element to start with. If the list is empty, what should you return? Of course, you know that the result should be 0, but this only works for a sum. It doesn't work for a product.

Also consider the `groupByCard` method. It looks like a business method that can only be used to group payments by credit cards. But it's not! You could use this method to group the elements of any list by any of their properties, so this method should be abstracted and put inside the `List` class in such a way that it could be reused easily.

A very important part of functional programming consists in pushing abstraction to the limit. In the rest of this book, you'll learn how to abstract many things so you never have to define them again. You will, for example, learn how to abstract loops so you won't have to write loops ever again. And you'll learn how to abstract parallelization in a way that will allow you to switch from serial to parallel processing just by selecting a method in the `List` class.

1.8 Summary

- Functional programming is programming with functions, returning values, and having no side effects.
- Functional programs are easy to reason about and easy to test.
- Functional programming offers a high level of abstraction and reusability.
- Functional programs are more robust than their imperative counterparts.
- Functional programs are safer in multithreading environments because they avoid shared mutable state.

JAVA

Functional Programming in Java

How functional techniques improve your Java programs

Pierre-Yves Saumont

Here's a bold statement: learn functional programming and you'll be a better Java developer. Fortunately, you don't have to master every aspect of FP to get a big payoff. If you take in a few core principles, you'll see an immediate boost in the scalability, readability, and maintainability of your code. And did we mention that you'll have fewer bugs? Let's get started!

Functional Programming in Java teaches you how to incorporate the powerful benefits of functional programming into new and existing Java code. This book uses easy-to-grasp examples, exercises, and illustrations to teach core FP principles such as referential transparency, immutability, persistence, and laziness. Along the way, you'll discover which of the new functionally inspired features of Java 8 will help you most.

What's Inside

- Writing code that's easier to read and reason about
- Safer concurrent and parallel programming
- Handling errors without exceptions
- Java 8 features like lambdas, method references, and functional interfaces

Written for Java developers with no previous FP experience.

Pierre-Yves Saumont is a Java developer with three decades of experience designing and building enterprise software. He is an R&D software engineer at Alcatel-Lucent Submarine Networks.

"An excellent introduction to functional programming for Java programmers."
—Piotr Bzdyl, SmartRecruiters

"Helped me understand the basic concepts of functional programming."
—Philippe Charrière, GitHub

"The fundamental concepts and mindset of functional programming explained with the help of a non-esoteric programming language like Java."
—Sebastian Metzger, snapADDY

"Shows you how to write modern Java code."
—Alessandro Campeis, Vimar

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/functional-programming-in-java

ISBN-13: 978-1-617-29273-6
ISBN-10: 1-617-29273-7
54999
9 781617 292736

MANNING US $ 49.99 | Can $ 57.99