The Joy of Kotlin

Pierre-Yves Saumont

MEAP

MANNING

MEAP Edition
Manning Early Access Program
The Joy of Kotlin
Version 8

For more information on this and other Manning titles go to
www.manning.com

welcome

Welcome to *The Joy of Kotlin*. The goal of this book is not simply to help you learn the Kotlin language, but also to teach you how you can write much safer programs using Kotlin. This does not mean that you *should* use Kotlin if you want to write safer programs, and even less that you can only write safer programs in Kotlin. This book uses Kotlin for all examples because Kotlin is one of the friendliest languages for writing safe programs in the Java ecosystem. Some even think that Kotlin is what Java could or should have become, if backward compatibility had not been so tightly enforced.

This book teaches techniques that were developed long ago in many different environments, although many of them come from functional programming. But this is not a book about fundamentalist functional programming. This is a book about pragmatic safe programming. All the techniques described have been put in production in the Java ecosystem for years, and have proven to be very effective in producing programs with much fewer implementation bugs than traditional programming techniques. These techniques can be implemented in any language, and they have been used by some for many years in Java. But often, this has been done through struggling to overcome Java limitations. Kotlin frees the programmer from many of these limitations.

This book is not about learning programming from level zero. It is aimed primarily at Java programmers in professional environments who are looking for an easier and safer way to write bug-free programs. It might not be a book for "good" programmers, since good programmers are able to write bug-free programs without effort and generally prefer freedom (including the freedom to write buggy programs) over safety. This is like good drivers being against speed limits, since speed limits are only for bad drivers. Instead, this book is for average programmers who would like to learn techniques for systematically writing bug-free programs. Of course, I am mainly talking about implementation bugs, but not just those.

One of the most important techniques you will learn is pushing abstraction much further (although traditional programmers consider premature abstraction to be as evil as premature optimization). Pushing abstraction to the limit results in a much better understanding of the problem to solve, which in turn results in more often solving the right problem instead of a different one.

Other techniques you will learn are immutability, referential transparency, clear separation between functions and effects, using laziness, encapsulating shared state mutation, safe handling of errors and exceptions, using data sharing immutable structures, replacing control structures with functions, completely avoiding control flow, and more.

One very important point to note is that this book is primarily an exercise book. You are not supposed to fully understand the text without doing the exercises (meaning finding your own solutions without cheating). If you understand the concepts just from reading the text, good for you, but you probably don't need this book. This book in not about information, but about training. In the same way you won't become a good musician by just reading books about music, you will mostly benefit from this book through working through the exercises. Remember that you will always learn more from trial and error than from successes.

Furthermore, all concepts are presented in progression. Each new concept is most often based upon previous ones, so you should use this book from start to end, and not by randomly picking subjects of interest.

I suspect that many readers will be Java programmers looking for new solutions to their daily problems. If this is your case, you may be wondering why you should use Kotlin. Aren't there already other languages in the Java ecosystem that make it possible for you to easily apply safe programming techniques? Sure there are. One of the most well-known is Scala. Scala is a very good alternative to Java, but Kotlin has something more. Scala can interact with Java at the library level, meaning that Java programs can use Scala libraries (objects and functions) and Scala libraries can use Java libraries (objects and methods). But Scala and Java programs must be built as separate projects, with distinct build chains.

Kotlin is different because it integrates with Java programs at the source level. You can mix Java and Kotlin source files in the same projects with a single build chain. This is a game changer, especially regarding team programming, where using Kotlin in a Java environment is no more hassle that using any third-party library. This make for a smoother transition from Java to a new language allowing writing programs that are safer, easier to write, to test, and to maintain, and more scalable.

If you have any questions, comments, or suggestions, please share them in Manning's Author Online forum for my book: https://forums.manning.com/forums/the-joy-of-kotlin

—Pierre-Yves Saumont

brief contents

about this book

The goal of this book is not simply to help you learn the Kotlin language, but also to teach you how you can write much safer programs using Kotlin. This does not mean that you should only use Kotlin if you want to write safer programs, and even less that only Kotlin allows writing safer programs. This book uses Kotlin for all examples because Kotlin is one of the friendliest languages for writing safe programs in the JVM (Java Virtual Machine) ecosystem. Some even think that Kotlin is what Java should have become, if backward compatibility had not been so tightly enforced.

This book teaches techniques that were developed long ago in many different environments, although much of them come from functional programming. But this is not a book about fundamentalist functional programming. This is a book about pragmatic safe programming. All the techniques described have been put in production in the Java ecosystem for years, and have proven to be very effective in producing programs with much fewer implementation bugs than traditional programming techniques. These safe techniques can be implemented in any language, and they have been used by some for many years in Java. But often, this has been done through struggling to overcome Java limitations.

This book is not about learning programming from level zero. It is aimed mainly at programmers in professional environments who are looking for an easier and safer way to write bug-free programs.

WHAT YOU WILL LEARN

In this book, you will learn specific techniques that might differ from what you have learned if you are a Java programmer. Most of these techniques will sound unfamiliar or even in contradiction with what programmers usually recognize as "best practices." But many (though not all) "best practices" are from when computers had 640K of memory, 5MB of disk storage, and a single-core processor. Things have changed. Nowadays, a simple smartphone is a computer with as much as 3GB RAM memory, 256GB of solid state disk storage, and an 8-core processor; likewise, computers have many gigabytes of memory, terabytes of storage, and multi-core processors.

The techniques I cover in this book include

- pushing abstraction further,

- immutability,
- referential transparency,
- encapsulated state mutation sharing,
- abstracting control flow and control structures,
- using the right types, and
- laziness

Pushing abstraction further

One of the most important techniques you will learn is pushing abstraction much further (although traditional programmers consider premature abstraction to be as evil as premature optimization). Pushing abstraction further results in a much better understanding of the problem to solve, which in turn results in more often solving the right problem instead of a different one.

Immutability

Immutability is a technique consisting in using only non-modifiable data. Many traditional programmers have trouble imagining how it can be possible to write useful programs using only immutable data. Isn't programming primarily based upon modifying data? Well, this is like believing that accounting is primarily modifying values in an accounting book. The transition from "mutable" to "immutable" accounting was made in the 15th century, and the principle of immutability has been recognized since then as the main element of safety for accounting. This principle also applies to programming, as you will see in this book.

Referential transparency

Referential transparency is a technique allowing you to write deterministic programs, meaning programs whose results you can predict and reason about. These programs always produce the same result when given the same input. This doesn't mean that they always produce the same result, but that variation in result only depends upon variation in input and not upon external conditions. Not only are such programs safer (since you always know how they will behave) but they are much easier to compose, to maintain, to update, and to test. And programs that are easier to test are of course generally much better tested and hence more reliable.

Encapsulated state mutation sharing

Immutable data is automatically protected against accidental sharing of state mutation, which causes many problem in concurrent and parallel processing, such as deadlock, livelock, thread starvation, and stale data. But making state mutation sharing impossible (since there is no state mutation) is a problem when state must be shared, which is the case in concurrent and parallel programming. In this book, you'll learn how to abstract and encapsulate state mutation sharing so that you will only have to write it once; then you can reuse it everywhere you need it.

Abstracting control flow and control structures

The second common source of bugs in programs (after sharing mutable state) is messing with control structures. Traditional programs are composed of control structures such as loops and conditional testing. It is so easy to mess with these structures that language designers have tried to abstract the details as much as possible. One of the best examplee is the `for each` loop that is now present in most languages (although in Java it is still simply called `for`).

Another common problem is the correct use of `while` and `do while` (or `repeat until`), and particularly determining where to test the condition.

An additional problem is concurrent modification while looping on collections, where you encounter the problem of sharing mutable state although you're using a single thread!

Abstracting control structures makes it possible to completely eliminate these kinds of problems.

Using the right types

Even the simplest control structure, call the *sequence*, where instructions are supposed to be executed in sequence in a predefined order, can be a source of bugs if instructions are out of order. This is only made possible because generic types such a `int` and `String` are used to represent quantities without taking units in account. Using value types can completely eliminate this kind of problem at a very low cost, even if the language you are using doesn't offer true value types.

Laziness

Most of the common languages are said to be strict, meaning that arguments passed to a method or function are evaluated first before being processed. This seems to make sense, although it often doesn't. On contrary, laziness is a

technique consisting in evaluating elements only if and when they are to be used. Programming is essentially based upon laziness. For example, in an `if..else` structure, the condition is strictly evaluated, meaning that it is evaluated before being tested, but the branches are lazily evaluated, which means only the branch corresponding to the condition is executed. This laziness is totally implicit and the programmer doesn't control it. Making explicit use of laziness will help you write much more efficient programs.

WHY KOTLIN

Each language has its own way, determined by some fundamental concepts. Java was created with several strong concepts in mind. It's supposed to run everywhere, meaning in any environment for which a JVM is available. The promise was, "Write once, run anywhere." Although some may argue otherwise, this promise was fulfilled. Not only can you run Java programs nearly everywhere, but you can also run programs written in other languages and compiled for the JVM. Kotlin is one of those languages.

Another of Java's promises was that no evolution would ever break existing code. Although this hasn't always been true, it has most often been respected. But this might not have been a good thing. The main consequence is that many improvements in other languages couldn't be brought into Java because those improvements would have destroyed compatibility. Any program compiled with a previous version of Java must be able to run in the newer versions without being recompiled. Whether this is useful or not is a matter of opinion, but the result is that backward compatibility has constantly played against Java's evolution.

Java was also supposed to make programs safer by using checked exceptions, thus forcing programmers to take these exceptions into consideration. For many programmers, this has proven to be a burden, leading to the practice of constantly wrapping checked exceptions into unchecked ones.

Although Java is an object-oriented language, it was supposed to be as fast as most languages for crunching numbers. So, the language designers decided that besides objects representing numbers and Boolean values, Java would benefit from having corresponding non-object primitives, allowing for much faster computations. The consequence was that you couldn't (and still can't) put primitives into collections such as lists, sets, or maps. And when streams were added, the language designers decided to create specific versions for

primitives—but not all primitives, only those most commonly used. If you're using some of the unsupported primitives, you're out of luck.

The same thing happened with functions. Generic functions were added to Java 8, but they'd only allow manipulating objects, not primitives. So specialized functions were designed to handle integers, longs, doubles, and booleans. (Again, unfortunately, not all primitives. There are no functions for byte, short, and float primitive types.) To make things even worst, additional functions were needed to convert from one primitive type to another, or from primitives to objects and back.

Java was designed more that 20 years ago. Many things have changed since that time, but most of these changes couldn't be brought into Java because it would have broken compatibility, or they were brought in in such a way that compatibility was preserved at the expense of usability.

Many new languages, such as Groovy, Scala, and Clojure, have since been released to address these limitations. These languages are compatible with Java to a certain extent, meaning one can use existing Java libraries in projects written in these languages, and Java programmers can use libraries developed in these languages.

Kotlin is different. Kotlin is much more strongly integrated with Java, to the point that you can mix Kotlin source code and Java source code in the same project without any difficulties. Unlike other JVM languages, Kotlin doesn't look like a different language (although it's somewhat different!). Instead, it looks like what Java should have become. Some even say that Kotlin is Java made right, meaning that it fixes most of the problems with the Java language. (Kotlin has, however, to deal with the limitations of the JVM.)

But more importantly, Kotlin was designed to be much friendlier to all the techniques I described in the previous section. Kotlin has both mutable and immutable data structures and references, but it heavily promotes preferring immutable ones. Kotlin also offers in standard a great part of the functional abstractions that allow avoiding control structures (although it also have traditional control structures in order to smooth transition from traditional languages).

Another great benefit of using Kotlin is that it reduces the need for boilerplate code to the bare minimum. With Kotlin, you can create a class with two optional properties, plus `equals`, `hashcode`, `toString`, and `copy` methods, in a single line

of code, where the same class written in Java would need about thirty lines (including getters, setters, and overloaded constructors).

Although other languages exist that were designed to overcome Java's limitations in the JVM environment, Kotlin is different because it integrates with Java programs at the project source level. You can mix Java and Kotlin source files in the same projects with a single build chain. This is a game changer, especially regarding team programming, where using Kotlin in a Java environment is no more hassle than using any third-party library. This makes for the smoothest possible transition from Java to a new language that makes it possible for you to write programs that are safer; easier to write, test, and maintain; and more scalable.

I suspect that many readers will be Java programmers looking for new solutions to their day-to-day problems. If this is you, you may be asking why you should use Kotlin. Aren't there already other languages in the Java ecosystem with which you can easily apply safe programming techniques? Sure there are, and one of the most well-known is Scala. Scala is a very good alternative to Java, but Kotlin has something more. Scala can interact with Java at the library level, meaning that Java programs can use Scala libraries (objects and functions) and Scala libraries can use Java libraries (objects and methods). But Scala and Java programs have to be built as separate projects, or at least separate modules, whereas Kotlin and Java classes can be mixed inside the same module.

AUDIENCE

This book is for readers with some previous programming experience in Java. Some understanding of parameterized types (generics) is assumed. The book makes heavy use of such techniques, including parameterized method calls, or variance, which are not often used in Java (although it is a powerful technique). Don't be afraid if you don't know these techniques already: I'll explain what they mean and why they're needed.

HOW TO USE THIS BOOK

This book is intended to be read sequentially, because each chapter builds upon the concepts learned in the previous ones. I use the word "read," but this book isn't intended to just be read. Very few sections are theory only. To get the most out of this book, read it at your computer, solving the exercises as you go. Each chapter includes a number of exercises with the necessary instructions and hints to help you arrive at the solution. All the code is available as a separate free download from GitHub (github.com/pysaumont/fpinkotlin). Each exercise comes

with a proposed solution and test that you can use to verify that your solution is correct.

The code comes with all the necessary elements for the project to be imported into IntelliJ (recommended), or to be compiled and run using Gradle 4. If you use Gradle, you can edit the code with any text editor. Note that Kotlin is supposed to be usable with Eclipse, but I can't guarantee this. But IntelliJ is a far superior IDE and is downloadable for free from the Jetbrains site (www.jetbrains.com/idea/download).

Please note that you're not expected to understand most of the concepts presented in this book just by reading the text. Doing the exercises is probably the most important part of the learning process, so I encourage you not to skip any exercises. Some might seem quite difficult, and you might be tempted to look at the proposed solutions. It's perfectly okay to do so, but you should then come back to the exercise and do it without looking at the solution. If you only read the solution, you'll probably have trouble later trying to solve more advanced exercises.

This approach doesn't require much tedious typing, because you have nearly nothing to copy. Most exercises consist of writing implementations for methods, for which you are given the environment and the method signature. No exercise solution is longer than a dozen lines of code; the majority are around four or five lines long.

Once you finish an exercise (which means when your implementation compiles), just run the corresponding test to verify that it's correct.

One important thing to note is that each exercise is self-contained in regard to the rest of the chapter, so code created inside a chapter is duplicated from one exercise to the next. This is necessary because each exercise is often built upon the preceding one, so although the same class might be used, implementations differ. As a consequence, you shouldn't look at a later exercise before you complete the previous ones, because you'd see the solutions to yet-unsolved exercises.

You can download the code as an archive, or you can clone it using Git. I highly recommend cloning, especially during the MEAP process, since the code is subject to many changes while the book is being written. It's much more efficient to update your code with a simple pull command than to re-download the complete archive.

The code for exercises is organized in modules with names that reflect the chapter titles, rather than the chapter numbers. As a result, IntelliJ will sort them alphabetically, rather than in the order in which they appear in the book. To help you figure out which module corresponds to each chapter, I've provided a list of the chapters with the corresponding module names in the README file accompanying the code (github.com/pysaumont/fpinkotlin).

AUTHOR ONLINE

Purchase of *The Joy of Kotlin* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and other users, or even provide help to other users. To access the forum and subscribe to it, point your web browser to forums.manning.com/forums/the-joy-of-kotlin. This Author Online page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

- Manning's commitment to our readers is to provide a venue where a meaningful dialog among individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary. I, as the author of this book, will be monitoring this forum and will answer questions as promptly as possible. It might however happen that I do not receive a notification when a question is posted on the forum. If you have a doubt, you can post a message on Github (github.com/pysaumont/fpinkotlin).
- The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Preface

Although Kotlin appeared in 2011, it is one of the newest languages in the Java ecosystem. Since then, a version of Kotlin running on the JavaScript virtual machine as been released, as well as a version compiling to native code. This makes Kotlin a much more universal language than Java, although there are great differences between these versions since the Java version relies upon the Java standard library, which is not available in the two others. JetBrains, the creator or Kotlin, is working hard to bring each version to an equivalent level, but this will need some time.

The JVM (Java Virtual Machine) version is by far the most used version, and this has seen a great boost when Google decided to adopt Kotlin as an official language for developing Android applications. One of the primary reasons for this adoption is the fact that the version of Java available under Android is Java 6, whether Kotlin offers most of the features of Java 11, and much more. Kotlin was also adopted by Gradle as the official language for writing build scripts, in replacement of Groovy, which allows using the same language for the build and for what is built.

So, Kotlin is primarily targeted at Java programmers. There might come the time when programmers will learn Kotlin as their primary language, but for now, most programmers will only be transitioning from Java to Kotlin.

Each language has its own way, determined by some fundamental concepts. Java was created with several strong concepts in mind. It’s supposed to run everywhere, meaning in any environment for which a JVM is available. The promise was, "Write once, run anywhere." Although some may argue otherwise, this promise was fulfilled. Not only can you run Java programs nearly everywhere, but you can also run programs written in other languages and compiled for the JVM. Kotlin is one of those languages.

Another of Java’s promises was that no evolution would ever break existing code. Although this hasn’t always been true, it has most often been respected. But this might not have been a good thing. The main consequence is that many improvements in other languages couldn’t be brought into Java because these improvements would have destroyed compatibility. Any program compiled with a previous version of Java must be able to run in the newer versions without being recompiled. Whether this is useful or not is a matter of opinion, but the

result is that backward compatibility has constantly played against Java's evolution.

Java was also supposed to make programs safer by using checked exceptions, thus forcing programmers to take these exceptions into consideration. For many programmers, this has proven to be a burden, leading to the practice of constantly wrapping checked exceptions into unchecked ones.

Although Java is an object-oriented language, it was supposed to be as fast as most languages for crunching numbers. So, the language designers decided that besides objects representing numbers and Boolean values, Java would benefit from having corresponding non-object primitives, allowing for much faster computations. The consequence was that you couldn't (and still can't) put primitives into collections such as lists, sets, or maps. And when streams were added, the language designers decided to create specific versions for primitives—but not all primitives, only those most commonly used. If you're using some of the unsupported primitives, you're out of luck.

The same thing happened with functions. Generic functions were added to Java 8, but they'd only allow manipulating objects, not primitives. So specialized functions were designed to handle integers, longs, doubles, and booleans. (Again, unfortunately, not all primitives. There are no functions for byte, short, and float primitive types.) To make things even worst, additional functions were needed to convert from one primitive type to another, or from primitives to objects and back.

Java was designed more than 20 years ago. Many things have changed since that time, but most of these changes couldn't be brought into Java because it would have broken compatibility, or they were brought into Java in such a way that compatibility was preserved at the expense of usability.

Many new languages, such as Groovy, Scala, and Clojure, have since been released to address these limitations. These languages are compatible with Java to a certain extent, meaning you can use existing Java libraries in projects written in these languages, and Java programmers can use libraries developed in these languages.

Kotlin is different. Kotlin is much more strongly integrated with Java, to the point that you can mix Kotlin source code and Java source code in the same project without any difficulties. Unlike other JVM languages, Kotlin doesn't look like a different language (although it's somewhat different!). Instead, it looks like what Java should have become. Some even say that Kotlin is Java made right,

meaning that it fixes most of the problems with the Java language. (Kotlin has, however, to deal with the limitations of the JVM.)

But more importantly, Kotlin was designed to be much friendlier to many the techniques coming from functional programming. Kotlin has both mutable and immutable references, but it heavily promotes preferring immutable ones. Kotlin also has a great part of the functional abstractions that allow avoiding control structures (although it also have traditional control structures in order to smooth transition from traditional languages).

Another great benefit of using Kotlin is that it reduces the need for boilerplate code to the bare minimum. With Kotlin, you can create a class with two optional properties, plus `equals`, `hashCode`, `toString`, and `copy` functions, in a single line of code, where the same class written in Java would need about thirty lines (including getters, setters, and overloaded constructors).

Although other languages exist that were designed to overcome Java's limitations in the JVM environment, Kotlin is different because it integrates with Java programs at the project source level. You can mix Java and Kotlin source files in the same projects with a single build chain. This is a game changer, especially regarding team programming, where using Kotlin in a Java environment is no more hassle than using any third-party library. This makes for the smoothest possible transition from Java to a new language that makes it possible for you to write programs that are:

- safer
- easier to write, test, and maintain
- more scalable.

I suspect that many readers will be Java programmers looking for new solutions to their day-to-day problems. If this is you, you may be asking why you should use Kotlin. Aren't there already other languages in the Java ecosystem with which you can easily apply safe programming techniques?

Sure there are, and one of the most well-known is Scala. Scala is a very good alternative to Java, but Kotlin has something more. Scala can interact with Java at the library level, meaning that Java programs can use Scala libraries (objects and functions) and Scala libraries can use Java libraries (objects and methods). But Scala and Java programs have to be built as separate projects, or at least separate modules, whereas Kotlin and Java classes can be mixed inside the same module.

1 Making programs safer

This chapter covers:

- Identifying programming traps
- Problems with side effects
- How referential transparency makes programs safer
- Using a substitution model to reason about programs
- Making the most of abstraction

Programming is a dangerous activity. If you're a hobbyist programmer, you may be surprised to read this. You probably thought you were safe sitting in front of your screen and keyboard. You might think that you don't risk much more than some back pain for sitting too long, some vision problems from reading tiny characters onscreen, or even some wrist tendonitis if you happen to type too furiously. But if you're (or want to be) a professional programmer, the reality is much worse than this.

The main danger is the bugs that are lurking in your programs. Bugs can cost a lot if they manifest at the wrong time. Remember the Y2K bug? Many programs written between 1960 and 1990 used only two digits to represent the year in dates because the programmers didn't expect their programs would last until the next century. Many of these programs were still in use in the 1990s and would have handled year 2000 as 1900. The estimated cost of that bug, actualized in 2017 US dollars, was $417 billion.[1]

But for bugs occurring in a single program, the cost can be much higher. On June 4, 1996, the first flight of the French Ariane 5 rocket ended after 36 seconds with a

[1] Federal Reserve Bank of Minneapolis Community Development Project. "Consumer Price Index (estimate) 1800–" https://www.minneapolisfed.org/community/teaching-aids/cpi-calculator-information/consumer-price-index-1800.

crash. It appears that the crash was due to a single bug in the navigation system. A single integer arithmetic overflow caused a $370 million loss.[2]

How would you feel if you were held responsible for such a disaster? How would you feel if you were writing this kind of program on a day-to-day basis, never sure that a program working today will still be working tomorrow? This is what most programmers do: writing undeterministic programs that don't produce the same result each time they are run with the same input data. Users are aware of this, and when a program doesn't work as expected, they try again, as if the same cause could produce a different effect the next time. And it sometimes does because nobody knows what these programs depend on for their output.

With the development of artificial intelligence (AI), the problem of software reliability becomes more crucial. If programs are meant to make decisions that can jeopardize human life, such as flying planes or driving autonomous cars, we'd better be sure they work as intended.

What do we need to make safer programs? Some will answer that we need better programmers. But good programmers are like good drivers. Of the programmers, 90% agree that only 10% of all programmers are good enough, but at the same time, 90% of the programmers think they are part of the 10%!

The most needed quality for programmers is to acknowledge their own limitations. Let's face it: we are only, at best, average programmers. We spend 20% of our time writing buggy programs, and then we spend 40% of our time refactoring our code to obtain programs with no apparent bugs. And later, we spend another 40% debugging code that's already in production because bugs come in two categories: apparent and non-apparent. Rest assured, non-apparent bugs will become apparent—it's just a matter of time. The question remains, how long and how much damage will be done before the bugs become apparent¸ .

What can we do about this problem? No programming tool, technique, or discipline will ever guarantee that our programs are completely bug-free. But there exist many programming practices that can eliminate some categories of bugs and guarantee that the remaining bugs only appear in isolated (unsafe) areas of our programs. This makes a huge difference because it makes bug hunting much easier and more efficient. Among such practices are writing programs that are so simple that they have no bugs rather than writing programs that are so complex that they have no obvious bugs.[3]

In the rest of this chapter, I briefly present concepts like immutability, referential transparency, and the substitution model, as well as other suggestions, which

[2] Rapport de la commission d'enquête Ariane 501 Echec du vol Ariane 501 http://www.astrosurf.com/luxorion/astronautique-accident-ariane-v501.htm.

[3] there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. "The Emperor's Old Clothes,"*Communications of the ACM*

together you can use to make your programs much safer. You'll apply these concepts over and over in the upcoming chapters.

1.1 Identifying programming traps

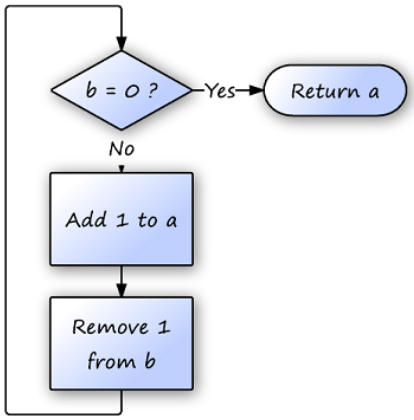
Programming is often seen as a way of describing how some process is to be carried out. Such a description generally includes actions that mutate a state in a program's model to solve a problem and decisions about the result of such mutations. This is something everyone understands and practices, even if they aren't programmers.

If you've some complex task to achieve, you divide it into steps. You then execute the first step and examine the result. Following the result of this examination, you continue with the next step or an alternate one. For example, a program for adding two positive values `a` and `b` might be represented by the following pseudocode:

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

In this pseudocode, you can recognize the traditional instructions of most languages: testing conditions, mutating variables, branching, and returning a value. This code can be represented graphically by a flow chart like that shown in figure 1.1.

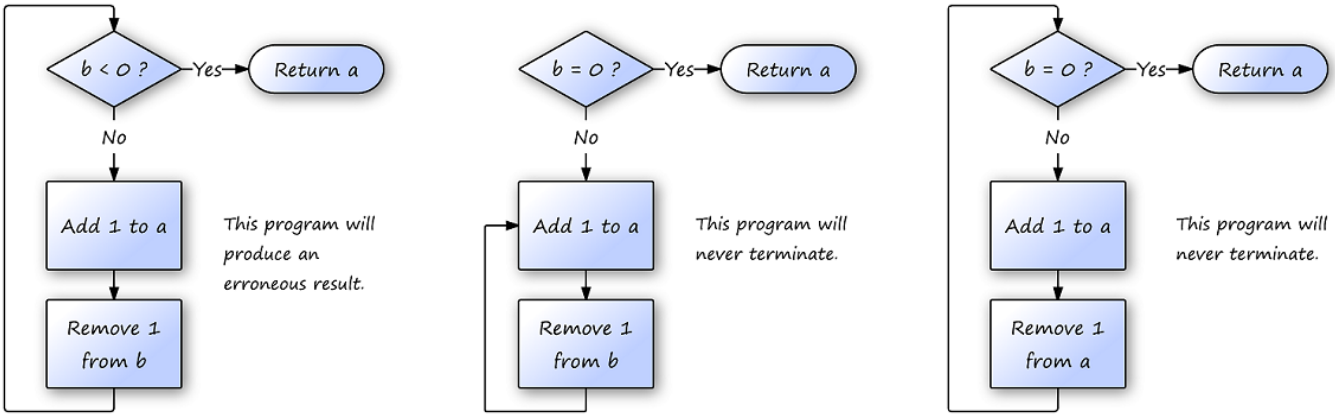
Figure 1.1. A flow chart representing a program as a process that occurs in time. Various things are transformed and states are mutated until the result is obtained.



You can easily see how such a program could go wrong. Change any data on the flowchart, or change the origin or the destination of any arrow, and you get a potentially buggy program. Note that if you're lucky, you could get a program that doesn't run at all, or that runs forever and never stops. This could be considered as

good luck because it would immediately allow you to see that there's a problem that needs fixing. Figure 1.2 shows three examples of such problems:

Figure 1.2. Three buggy versions of the same program



The first example produces an erroneous result, and the second and the third never terminate. Note, however, that your programming language may not allow you to write some of these examples. None of these could be written in a language that doesn't allow mutating references, and none of them could be written in a language that doesn't allow branching or looping. You might think all you have to do is to use such a language. And, in fact, you could. But you'd be restricted to a small number of languages and probably none of them would be allowed in your professional environment.

Is there a solution? Yes, there is. What you can do is to avoid using mutable references, branching (if your language allows it), and looping. All you need to do is to program with discipline. Don't use dangerous features like mutations and loops. It's as simple as that! And if you do find that you eventually need mutable references or loops, abstract them. Write some component that abstracts state mutation once and for all, and you'll never again have to deal with the problem. (Some more or less exotic languages offer this out of the box, but these too are probably not languages you can use in your environment.)

The same applies to looping. In this case, most modern languages offer abstractions of looping alongside a more traditional usage of loops. Again, it's a question of discipline. Only use the good parts! More on this in chapters 4 and 5.

Another common source of bugs is the `null` reference. As you'll see in chapter 6, with Kotlin, you can clearly separate code that allows null references from code that forbids these. But ultimately, it's up to you to completely eradicate the use of null references from your programs.

A huge category of different types of bugs is caused by programs depending on the

outside world to execute correctly. But, depending on the outside world is generally necessary in some way in all programs. Restricting this dependency to specific areas of your programs will make problems easier to spot and deal with, although it won't completely remove the possibility of these types of bugs.

In this book, you'll learn several techniques to make your programs incredibly much safer. Here's a list of these practices:

- Avoiding mutable references (*variables*) and abstracting the single case where mutation can't be avoided.
- Avoiding control structures.
- Restricting effects (interaction with the outside world) to specific areas in your code. This means no printing to the console or to any device, no writing to files, databases, networks, or whatever else that can happen outside of these restricted areas.
- No exception throwing. Throwing exceptions is the modern form of branching (`GOTO`), which leads to what is called *spaghetti code*, meaning that you know where it starts, but you can't follow where it goes. In chapter 7, you'll learn how to completely avoid throwing exceptions.

1.2 Safely handling effects

As I said, the word *effects* means all interactions with the outside world, such as writing to the console, to a file, to a database, or to a network, and also mutating any element outside the component's *scope*. Programs are generally written in small blocks that have a scope. Some languages call these blocks *procedures*, others (like Java) call them *methods*. Koltin calls them *functions*, although this doesn't have the same meaning as the mathematical concept of a function.

Kotlin functions are basically methods, as in Java and many other modern languages. These blocks of code have a *scope*, meaning an area of the program that's visible only by those blocks. Blocks not only have visibility of the enclosing scope, but this itself also provides visibility of the outer scopes and, by transitivity, to the outside world. Any mutation of the outside world caused by a function or method (be it mutating the enclosing scope, like the class in which the method is defined) is, therefore, an effect.

Some methods (functions) return a value. Some mutate the outer world, and some do both. When a method or function returns a value that has an effect, this is called a *side effect*. Programming with side effects is wrong in all cases. In medicine, the term *side effects* is primarily used to describe unwanted, adverse secondary outcomes. In programming, a side effect is something that's observable outside of the program and comes *in addition* to the result returned by the program. If the program doesn't return a result, you can't call its observable effect a side effect; it's the primary effect. It can still have side (secondary) effects, although this is also generally considered bad practice, following what's called the **"single**

responsibility" principle.

Safe programs are built by composing functions that take an argument and return a value, and that's it. We don't care about what's happening *inside* the functions because, in theory, nothing ever happens there. Some languages only offer such effect-free functions: programs written in these languages don't have any observable effects beside returning a value. But this value can, in fact, be a new program that you can run to evaluate the effect. Such a technique can be used in any language, but it's often considered inefficient (which is arguable). A safe alternative is to clearly separate effects evaluation from the rest of the program and even, as much as possible, to abstract effect evaluation. You'll learn many techniques allowing this in chapters 7, 11, and 12.

1.3 Making programs safer with referential transparency

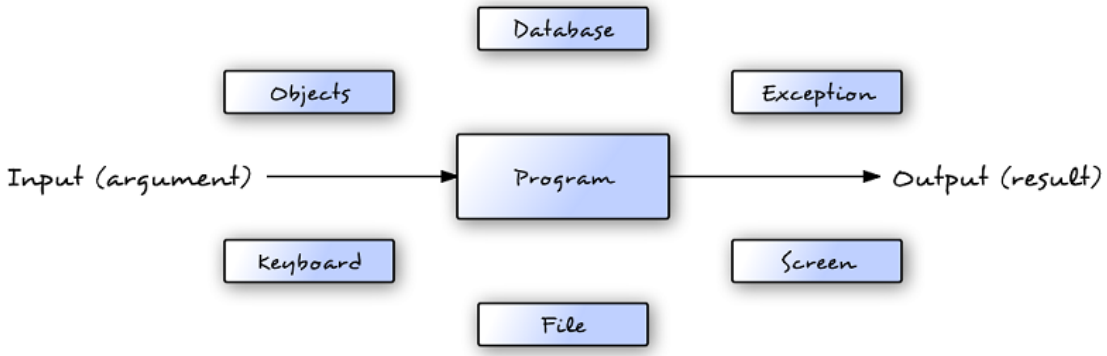
Having no side effects (not mutating anything in the external world) isn't enough to make a program safe and deterministic. Programs also mustn't be affected by the external world; the output of a program should depend only on its argument. This means that programs shouldn't read data from the console, a file, a remote URL, a database, or even from the system.

Code that neither mutates nor depends on the external world is said to be *referentially transparent*. Referentially transparent code has several interesting attributes that might be of interest:

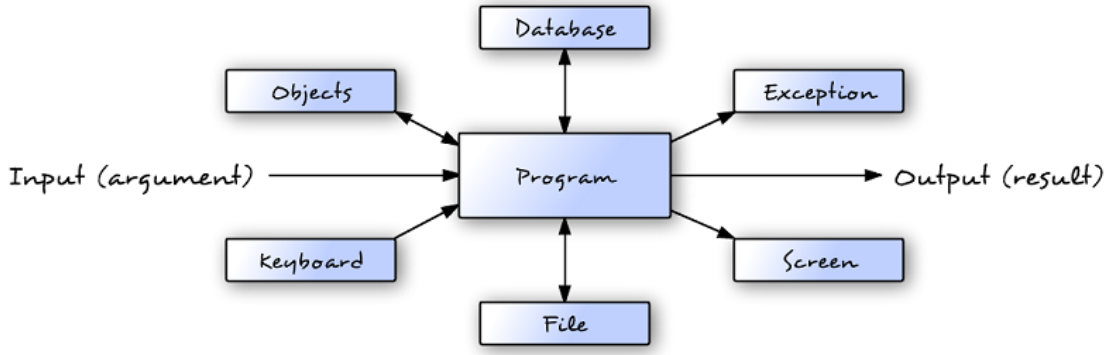
- *It's self-contained.* You can use it in any context—all you have to do is to provide a valid argument.
- *It's deterministic.* It always returns the same value for the same argument so you won't be surprised. It might, however, return a wrong result, but at least for the same argument, the result never changes.
- *It never throws any kind of exception.* It might throw errors, such as out-of-memory errors (OOMEs) or stack-overflow errors (SOEs), but these errors mean that the code has a bug. This isn't a situation you, as a programmer, nor the users of your API are supposed to handle (besides crashing the application, which often will not happen automatically, and eventually fixing the bug).
- *It doesn't create conditions causing other code to unexpectedly fail.* It won't mutate arguments or some other external data, for example, causing the caller to find itself with stale data or concurrent access exceptions.
- *It doesn't depend on any external device to work.* It won't hang because some external device (whether database, filesystem, or network) is unavailable, too slow, or broken.

Figure 1.3 illustrates the difference between a referentially transparent program and one that's not referentially transparent.

Figure 1.3. Comparing a program that's referentially transparent to one that's not



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.

1.4 The benefits of safe programming

From what I've described, you can likely guess the many benefits you can expect by using referential transparency:

- *Your programs will be easier to reason about because they'll be deterministic.* A specific input will always give the same output. In many cases, you might be able to prove a program correct rather than to extensively test it and still remain uncertain about whether it'll break under unexpected conditions.

```
//not sure what isolating the programs under test from the outside means
exactly; change OK?
* _Your programs will be easier to test._ Because there are no side effects,
```

```
you won't need mocks, which are generally required when testing to isolate
program components from the outside.
```

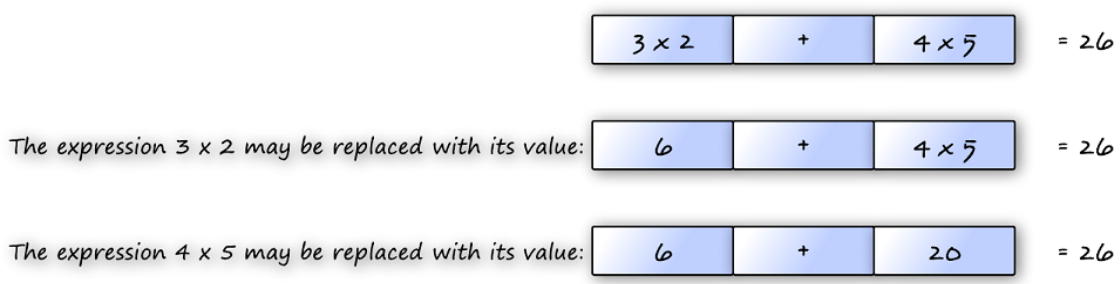
- *Your programs will be more modular.* That's because they'll be built from functions that only have input and output; there are no side effects to handle, no exceptions to catch, no context mutation to deal with, no shared mutable state, and no concurrent modifications.
- *Composition and recombination of programs is much easier.* To write a program, you start by writing the various base functions you'll need and then combine these functions into higher-level ones, repeating the process until you have a single function corresponding to the program you want to build. And, because all these functions are referentially transparent, they can then be reused to build other programs without any modifications.
- *Your programs will be inherently thread-safe because they avoid mutation of shared states.* This doesn't mean that all data has to be immutable, only shared data must be. But programmers applying these techniques soon realize that immutable data is always safer, even if the mutation is not visible externally. One reason is that data that's not shared at one point can become shared accidentally after refactoring. Always using immutable data ensures that this kind of problem never happens.

In the rest of this chapter, I will present some examples of using referential transparency to write safer programs.

1.4.1 Using the substitution model to reason about programs

The main benefit from using functions that return a value without any other observable effect is that they're equivalent to their return value. Such a function doesn't do anything. It has a value, which is dependent only on its arguments. As a consequence, it's always possible to replace a function call, or any referentially transparent expression, with its value as figure 1.4 shows.

Figure 1.4. Replacing referentially transparent expressions with their values doesn't change the overall meaning.



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

```
fun main(args: Array<String>) {
    val x = add(mult(2, 3), mult(4, 5))
    println(x)
}

fun add(a: Int, b: Int): Int {
    log(String.format("Returning ${a + b} as the result of $a + $b"))
    return a + b
}

fun mult(a: Int, b: Int) = a * b

fun log(m: String) {
    println(m)
}
```

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

```
val x = add(6, 20)
```

In contrast, replacing the call to the `add` function with its return value will change the signification of the program because the call to `log` will no longer be made, so no logging takes place. This might be important or not; in any case, it changes the outcome of the program.

1.4.2 Applying safe principles to a simple example

To convert an unsafe program into a safer one, let's consider a simple example representing the purchase of a donut with a credit card.

Listing 1.1. A Kotlin program with side effects

```
fun buyDonut(creditCard: CreditCard): Donut {
    val donut = Donut()
    creditCard.charge(Donut.price)      (1)
    return donut                        (2)
}
```

(1) Charges the credit card as a side effect
(2) Returns the donut

In this code, charging the credit card is a side effect. 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.

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` function 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. But 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` function will have to return both the donut and this representation of the payment. To represent the payment, you'll use a `Payment` class.

Listing 1.2. The `Payment` class

```
class Payment(val creditCard: CreditCard, val amount: Int)
```

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

```
class Purchase(val donut: Donut, val payment: Payment)
```

You'll often need such a class to hold two (or more) values of different types because, for making programs safer, you have to replace side effects with returning a representation of these effects.

Rather than creating a specific `Purchase` class, you can use a generic one, `Pair`. This class is parameterized by the two types it contains (in this case, `Donut` and `Payment`). Kotlin provides this class, as well as `Triple`, which allows the representation of three values. Such a class would be useful in a language like Java because defining the `Purchase` class would imply writing a constructor, getters, and probably `equals` and `hashcode` methods, as well as `toString`. That's much less useful in Kotlin because the same result can be obtained with a single line of code:

```
data class Purchase(val donut: Donut, val payment: Payment)
```

The `Purchase` class already doesn't need an explicit constructor and getters. By adding the `data` keyword in front of the class definition, Kotlin additionally provides implementations of `equals`, `hashCode`, `toString`, and `copy`; however, you must accept the default implementations. Two instances of a data class will be equal if all properties are equal. If this isn't what you need, you can override any of these functions with your own implementations.

```
fun buyDonut(creditCard: CreditCard): Purchase {
    val donut = Donut()
    val payment = Payment(creditCard, Donut.price)
    return Purchase(donut, payment)
}
```

Note that you're no longer concerned at this stage with how the credit card will be charged. This adds some freedom to the way you build your application. You could process the payment immediately, or you could store it for later processing. You can even combine stored payments for the same card and process them in a single operation. This would save you some money by minimizing the bank fees for the credit card service.

The `combine` function in the following listing is used to combine payments. Note that if the credit cards don't match, an exception is thrown. This doesn't contradict what I said about safe 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 realistic. You'll have to wait until chapter 7 to learn how to deal with such situations without throwing exceptions.)

Listing 1.3. Composing multiple payments into a single one

```
package com.fpinkotlin.introduction.listing03

class Payment(val creditCard: CreditCard, val amount: Int) {

    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")
}
```

In this scenario, the `combine` function wouldn't be efficient when buying several donuts at once. For this you could replace the `buyDonut` function with `buyDonuts(n: Int, creditCard: CreditCard)` as shown in the following listing. Note that you have to define a new `Purchase` class. Alternatively, if you had chosen to use a `Pair<Donut, Payment>`, you'd have to replace it with `Pair<List<Donut>, Payment>`.

Listing 1.4. Buying multiple donuts at once

```
package com.fpinkotlin.introduction.listing05

data class Purchase(val donuts: List<Donut>, val payment: Payment)

fun buyDonuts(quantity: Int = 1, creditCard: CreditCard): Purchase =
        Purchase(List(quantity) {
            Donut()
        }, Payment(creditCard, Donut.price * quantity))
```

Note that `List(quantity) { Donut() }` creates a list of `quantity` elements successively applying the function `` `{ Donut() }` `` to values `0` to `quantity - 1`. The `{ Donut() }` function is equivalent to:

```
{ index -> Donut{} }
```

or

```
{ _ -> Donut{} }
```

When there's a single parameter, you can omit the `parameter ->` part and use the parameter as `it`. Because it's not used, the code is reduced to `{ Donut() }`. If this is not clear, don't worry: we'll cover this more in the next chapter

Also note that the `quantity` parameter receives a default value of `1`. This lets you call the `buyDonuts` function with the following syntax without specifying the quantity:

```
buyDonuts(creditCard = cc)
```

In Java, you'd have to overload the method with a second implementation, such as

```
public static Purchase buyDonuts(CreditCard creditCard) {
    return buyDonuts(1, creditCard);
}

public static Purchase buyDonuts(int quantity,
                                 CreditCard creditCard) {
    return new Purchase(Collections.nCopies(quantity, new Donut()),
                        new Payment(creditCard, Donut.price * quantity));
}
```

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

```
import org.junit.Assert.assertEquals
import org.junit.Test

class DonutShopKtTest {

    @Test
    fun testBuyDonuts() {
        val creditCard = CreditCard()
        val purchase = buyDonuts(5, creditCard)
        assertEquals(Donut.price * 5, purchase.payment.amount)
        assertEquals(creditCard, purchase.payment.creditCard)
    }
}
```

Another benefit of having refactored your code is that your program is more easily composable. If the same person makes several purchases with your initial program, you'd have to contact the bank (and pay the corresponding fee) each time the person bought something. With the new version, however, 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 functions from the Kotlin `List` class:

- `groupBy(f: (A) → B): Map<B, List<A>>`:: Takes as its parameter a function from `A` to `B` and returns a map of keys and value pairs, with keys being of type `B` and values of type `List<A>`. You'll use it to group payments by credit cards.
- `values: List<A>`:: An instance function of `Map` that returns a list of all the values in the map.
- `map(f: (A) → B): List<B>`:: An instance function 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`.
- `reduce(f: (A, A) → A): A`:: A function of `List` that uses an operation to reduce the list to a single value. This operation is represented by a function `f: (A, A) → A`. It could be, for example, an addition. In such a case, it would mean a function such as `f(a, b) = a + b`.

Using these functions, you can now create a new function that groups payments by credit card as shown in the next listing.

Listing 1.5. Grouping payments by credit card

```
package com.fpinkotlin.introduction.listing05;

class Payment(val creditCard: CreditCard, val amount: Int) {
    fun combine(payment: Payment): Payment =
        if (creditCard == payment.creditCard)
            Payment(creditCard, amount + payment.amount)
        else
            throw IllegalStateException("Cards don't match.")

    companion object {
        fun groupByCard(payments: List<Payment>): List<Payment> =
            payments.groupBy { it.creditCard }              (1)
                    .values                                 (2)
                    .map { it.reduce(Payment::combine) }    (3)
    }
}
```

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

(2) Changes the Map<CreditCard, List<Payment>> into a List<List<Payment>>

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

Note the use of a function reference in the last line of the `groupByCard` function. Function references are similar to method references in Java.

If this example isn't clear, well, that's what this book is for! By the end, you'll be an expert in composing such code.

1.4.3 Pushing abstraction to the limit

As you've seen, you can write safer programs that are easier to test by composing *pure functions*, which means functions without side effects. You can declare these functions using the `fun`keyword or as value functions, such as the arguments of methods `groupBy`, `map`, or `reduce` in the previous listing. *Value functions* are functions represented in such a way that, unlike `fun` functions, they can be manipulated by the program. In most cases, you can use them as arguments to other functions or as values returned by other functions. You'll learn how this is done in the following chapters.

But the most important concept here is abstraction. Look at the `reduce` function. It takes as its argument an operation, and uses that operation 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` function to compute the sum of the elements; then you could write a `product` function to compute the product of the elements, or you could write a `min` or a `max` function to compute the minimum or the maximum of the list. Alternatively, you could also use the `reduce` function for all these computations. This is *abstraction*. You abstract the part that's common to all operations in the `reduce` function, and you pass the variable part (the operation) as an argument.

You could go further. The `reduce` function is a particular case of a more general function 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 use this function, called `fold`. Also note that the `reduce` function 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? You know that the result should be 0, but this only works for a sum. It won't work for a product.

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

Pushing abstraction to the limits allows making programs safer because the abstracted part will only be written once. As a consequence, once it's fully tested, there'll be no risk of producing new bugs by reimplementing it.

In the rest of this book, you'll learn how to abstract many things so you'll only have to define them once. 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 let you switch from serial to parallel processing by selecting a function in the `List` class.

1.5 Summary

- You can make programs safer by clearly separating functions, which return values, from effects, which interact with the outside world.
- Functions are easier to reason about and easier to test because their outcome is deterministic and doesn't depend on an external state.
- Pushing abstraction to a higher level improves safety, maintainability, testability, and reusability.
- Applying safe principles like immutability and referential transparency protects programs against accidental sharing of a mutable state, which is a huge source of bugs in multithreaded environments.