

**Marcin Moskała**

# **Effective Kotlin**

## **BEST PRACTICES**

**Second Edition**



# **Effective Kotlin**

Best practices

**Marcin Moskała**

This book is available at <http://leanpub.com/effectivekotlin>

This version was published on 2024-10-10



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2024 Marcin Moskała

## **Tweet This Book!**

Please help Marcin Moskała by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

[I just bought @EffectiveKotlin!](#)

The suggested hashtag for this book is [#effectivekotlin](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#effectivekotlin](#)

*For my parents*

# Contents

|  |            |
|--|------------|
| <b>Introduction: Be pragmatic</b>  | <b>1</b>   |
| Who is this book for?  | 4          |
| Code Sources   | 7          |
| <b>Part 1: Good code</b>   | <b>11</b>  |
| <b>Chapter 1: Safety</b>   | <b>12</b>  |
| Item 1: Limit mutability   | 13         |
| Item 2: Eliminate critical sections  | 25         |
| Item 3: Eliminate platform types as soon as possible   | 34         |
| Item 4: Minimize the scope of variables  | 40         |
| Item 5: Specify your expectations for arguments and state  | 46         |
| Item 6: Prefer standard errors to custom ones  | 56         |
| Item 7: Prefer a nullable or <code>Result</code> result type when the lack of a result is possible | 58         |
| Item 8: Close resources with <code>use</code>  | 63         |
| Item 9: Write unit tests   | 66         |
| <b>Chapter 2: Readability</b>  | <b>69</b>  |
| Item 10: Design for readability  | 70         |
| Item 11: An operator's meaning should be consistent with its function name                         | 75         |
| Item 12: Use operators to increase readability   | 80         |
| Item 13: Consider making types explicit  | 84         |
| Item 14: Consider referencing receivers explicitly   | 87         |
| Item 15: Properties should represent a state, not a behavior                                       | 93         |
| Item 16: Avoid returning or operating on <code>Unit?</code>  | 97         |
| Item 17: Consider naming arguments   | 99         |
| Item 18: Respect coding conventions  | 104        |
| <b>Part 2: Code design</b>   | <b>107</b> |
| <b>Chapter 3: Reusability</b>  | <b>108</b> |
| Item 19: Do not repeat knowledge   | 109        |
| Item 20: Do not repeat common algorithms   | 116        |

## CONTENTS

|   |            |
|---|------------|
| Item 21: Use generics when implementing common algorithms                           | 120        |
| Item 22: Avoid shadowing type parameters  | 123        |
| Item 23: Consider using variance modifiers for generic types                        | 125        |
| Item 24: Reuse between different platforms by extracting common modules             | 127        |
| <b>Chapter 4: Abstraction design</b>  | <b>132</b> |
| Item 25: Each function should be written in terms of a single level of abstraction  | 136        |
| Item 26: Use abstraction to protect code against changes                            | 142        |
| Item 27: Specify API stability  | 155        |
| Item 28: Consider wrapping external APIs  | 158        |
| Item 29: Minimize elements' visibility  | 160        |
| Item 30: Define contracts with documentation  | 164        |
| Item 31: Respect abstraction contracts  | 174        |
| <b>Chapter 5: Object creation</b>   | <b>176</b> |
| Item 32: Consider factory functions instead of constructors                         | 177        |
| Item 33: Consider a primary constructor with named optional arguments               | 192        |
| Item 34: Consider defining a DSL for complex object creation                        | 202        |
| Item 35: Consider using dependency injection  | 213        |
| <b>Chapter 6: Class design</b>  | <b>216</b> |
| Item 36: Prefer composition over inheritance  | 217        |
| Item 37: Use the data modifier to represent a bundle of data                        | 228        |
| Item 38: Use function types or functional interfaces to pass operations and actions | 235        |
| Item 39: Use sealed classes and interfaces to express restricted hierarchies        | 241        |
| Item 40: Prefer class hierarchies instead of tagged classes                         | 244        |
| Item 41: Use enum to represent a list of values                                     | 249        |
| Item 42: Respect the contract of equals   | 256        |
| Item 43: Respect the contract of hashCode   | 268        |
| Item 44: Respect the contract of compareTo  | 276        |
| Item 45: Consider extracting non-essential parts of your API into extensions        | 279        |
| Item 46: Avoid member extensions  | 284        |
| <b>Part 3: Efficiency</b>   | <b>289</b> |
| <b>Chapter 7: Make it cheap</b>   | <b>290</b> |
| Item 47: Avoid unnecessary object creation  | 291        |
| Item 48: Consider using object declarations   | 296        |
| Item 49: Use caching when possible  | 298        |
| Item 50: Extract objects that can be reused   | 301        |

## CONTENTS

|  |            |
|--|------------|
| Item 51: Use the inline modifier for functions with parameters of functional types | 304        |
| Item 52: Consider using inline value classes                                       | 318        |
| Item 53: Eliminate obsolete object references                                      | 326        |
| <b>Chapter 8: Efficient collection processing</b>                                  | <b>333</b> |
| Item 54: Prefer Sequences for big collections with more than one processing step   | 337        |
| Item 55: Consider associating elements to a map                                    | 350        |
| Item 56: Consider using groupingBy instead of groupBy                              | 355        |
| Item 57: Limit the number of operations  | 359        |
| Item 58: Consider Arrays with primitives for performance-critical processing       | 362        |
| Item 59: Consider using mutable collections  | 365        |
| Item 60: Use appropriate collection types  | 367        |
| <b>Dictionary</b>  | <b>374</b> |

# Introduction: Be pragmatic

Stories of the origins of popular programming languages are often fascinating.

The prototype of JavaScript (named Mocha back then) was created in just 10 days. Its creators first considered using Java but wanted to make a simpler language for Web designers<sup>1</sup>.

Scala was created at a university by a scientist, Martin Odersky. He wanted to transfer some concepts from functional programming into the Java object-oriented programming world. It turned out that these worlds are connectable<sup>2</sup>.

Java was originally designed in the early ‘90s for interactive television and set-top boxes by a team working at Sun called “The Green Team”. Eventually, this language was too advanced for the digital cable television industry at the time. It ended up revolutionizing general programming instead<sup>3</sup>.

Many languages were designed for a totally different purpose than what they are used for today. Many originated as experiments, as can still be seen in them today. The differences in the Kotlin story are that:

1. Since the beginning, it has been created and designed to be a general-purpose programming language that is suitable for large applications.
2. The creators of Kotlin are taking their time. The development of Kotlin started in 2010, and the first officially stable version was released in February of 2016. During this period, a lot changed. If you find some code from the first proposals, it looks almost nothing like Kotlin today.

**Kotlin was created as a pragmatic language for practical applications**, and this is reflected in its design. For instance, as opposed to academic or hobbyist languages, it has never had ambitions to experiment with new concepts. Kotlin has introduced a few new concepts (like property delegation), but the Kotlin team is very conscientious and prefers to analyze how existing concepts have cooperated and worked for other languages<sup>4</sup>. They are always trying to understand the strengths and weaknesses of other languages and build on them. This is a good strategy for JetBrains (the creators of Kotlin) since it is also the creator of IntelliJ, an Integrated Development Environment (IDE). They have a lot of data

---

<sup>1</sup>Read about it here: [kt.academy/l/wiki-js](https://kt.academy/l/wiki-js) and [kt.academy/l/js-origins](https://kt.academy/l/js-origins)

<sup>2</sup>Read about it here: [kt.academy/l/scala-origins](https://kt.academy/l/scala-origins)

<sup>3</sup>Read about it here: [kt.academy/l/java-origins](https://kt.academy/l/java-origins)

<sup>4</sup>I am talking about language itself. Some libraries provided by Kotlin team are full of experimental ideas. Kotlin Coroutines library is a good example.

and knowledge about how various languages are used, and they have specialists who understand each of these languages.

For that same reason, Kotlin is also different because it has reached a new level of cooperation between the IDE and the language. Code analysis in IntelliJ or Eclipse is done using the same compiler that is used to compile the code. Thanks to that, Kotlin can freely introduce more advanced smart casting and type inference without requiring IDE changes. The Kotlin team also supports developers by constantly improving IntelliJ warnings, hints, and lints. Because of this mechanism, most of the classic optimization hints don't need to be collected in books or articles because they can just be provided via discrete warnings exactly where they are needed.

## The philosophy of Kotlin

Every language has its own philosophy that determines design decisions. The central point of Kotlin's philosophy is pragmatism. This means that, in the end, all choices need to serve business needs, like:

- Productivity - application production is fast.
- Scalability - as an application grows, its development does not become more expensive. It may even get cheaper.
- Maintainability - maintenance is easy.
- Reliability - applications behave as expected, and there are fewer errors.
- Efficiency - applications run fast and need fewer resources (memory, processor, etc.).

As a programming community, we have tried to satisfy these needs for quite some time. Based on our experiences, we have developed different tools and techniques. For instance, we have learned that automatic testing is very important to prevent errors that are accidentally added to one feature when someone modifies another. There are also rules to follow. For instance, the Single Responsibility Principle from SOLID<sup>5</sup> helps us with the same problem. Throughout this book, we will mention many such rules.

The programming community also found that some less-abstract values (from the programmers' point of view) support higher-level business needs. The Kotlin team collected values that are important in terms of language design and used them as a point of reference for all design decisions. These values are:

- Safety

---

<sup>5</sup>SOLID is a popular set of principles for OOP, introduced and popularized by Robert C. Martin.

- Readability
- Powerful code reusability
- Tool friendliness
- Interoperability with other languages

I would add another point that is normally not included, but that can be seen in many decisions:

- Efficiency

These requirements were not only present at the inception of Kotlin: they are still with us today, and each change is considered with them in mind. I will also show that they are all very strongly reflected in Kotlin's design. This was possible thanks to the fact that Kotlin was intentionally kept in beta for nearly 6 years. During this time, it was changing at all levels. It takes a lot of time to shape the design of a programming language to reflect high-level values. The Kotlin creators did a good job of that.

## The purpose of this book

To really unleash the advantages of Kotlin, we need to use it properly. Knowing the features and the standard library (stdlib) is not enough. The main goal of this book is to explain how to use different Kotlin features to achieve safe, readable, scalable, and efficient code. Since this book is written to help developers improve their code, it also touches on many general rules for programmers. You can find the influence of programming classics like Clean Code, Effective Java, Structure and Implementation of Computer Programs, Code Complete, and many more. You can also feel the influence of suggestions from influential presentations and Kotlin forums. This book tries to offer as much knowledge about best practices in Kotlin as possible, no matter from where it originated.

You can call it a collection of best practices, but it differs from classic “Effective X” books because of Kotlin’s characteristics. The Effective Java book contains many warnings about internal Java problems. In Kotlin, such problems are mostly eliminated by the Kotlin team. In contrast to Java, Kotlin is not worried about deprecating something and fixing it in the future<sup>6</sup>. In the worst case, the Kotlin team controls a powerful IDE that can do nearly any migration to a better alternative. Most “Effective X” books also hint at preferring to use some functions or constructs over others. These kinds of suggestions are rarely useful in Kotlin as most of them already have a warning or hint in IntelliJ. I have left only a few such items in this book. This book is different: it concentrates on higher-level good practices that come from authorities, the Kotlin creators, and from my experience as a developer, consultant, and trainer for international companies worldwide.

---

<sup>6</sup>KotlinConf 2018 keynote by Andrey Breslav.

## Who is this book for?

This book does not teach the basics. It assumes that you have enough knowledge and skills to do Kotlin development. If you don't, I recommend starting with some resources designed for beginners, like my books *Kotlin Essentials* and *Functional Kotlin*. *Effective Kotlin* is for experienced Kotlin developers.

I will assume that even experienced developers might not know some features. This is why I explain concepts such as:

- Properties
- Platform types
- Named arguments
- DSL creation
- Inline classes and functions

I want this book to be a complete guide for Kotlin developers on how to become amazing Kotlin developers.

## Book design

Concepts in the book are grouped into three parts:

- Good code - more general rules about writing good-quality code. This part is for every Kotlin developer, no matter how big their project is. It starts from items about safety and later talks about readability. It is not a coincidence that the first chapter is dedicated to safety. I believe that program correctness is generally of the highest priority. Another chapter concerns readability because code is not only for a compiler but also for programmers. Even when we work alone, we want readable and self-explanatory code.
- Code design - this section is for developers creating projects or libraries in cooperation with other developers. This part is about conventions and declaring contracts. The *Code design* chapter will, in the end, reflect on readability and safety in terms of code design. This part is a bit more abstract at the beginning, but this lets it explore topics that are often omitted in books about code quality. This section is also about preparing our code for growth. Many items are about being ready for changes in the future and are therefore especially important for developers creating large projects.
- Efficiency - this section is for developers who care about code efficiency. Most of the rules presented here do not come at the cost of development time or readability, so they are suitable for everyone. However, they are particularly important for developers implementing high-performance applications, libraries, or applications for millions of users.

Each part is divided into chapters, which are subdivided into items. Here are the chapters in each part:

#### Part 1: Good code

- Chapter 1: Safety
- Chapter 2: Readability

#### Part 2: Code design

- Chapter 3: Reusability
- Chapter 4: Abstraction design
- Chapter 5: Objects creation
- Chapter 6: Class design

#### Part 3: Efficiency

- Chapter 7: Make it cheap
- Chapter 8: Efficient collection processing

Each chapter contains items that are suggestions. The concept is that items are rules that, in most cases, need an explanation, but once the idea is clear, it can be triggered just by this title. In the end, suggestions designed this way, with their explanations, should clearly show readers how to write good and idiomatic Kotlin code.

## Chapter organization

Chapters often start with the most important concept that is used in other items. A very visible example is *Chapter 2: Readability*, which starts with Item 10: *Design for readability*, but this is also true for:

- *Chapter 7: Make it cheap*'s first item is Item 47: *Avoid unnecessary object creation*
- *Chapter 3: Reusability*'s first item is Item 19: *Do not repeat knowledge*
- *Chapter 1: Safety*'s first item is Item 1: *Limit mutability*

Chapters can also end with an item that is generally less connected with the rest but presents an important concept that needs to be included, for instance:

- *Chapter 1: Safety*'s last item is Item 9: *Write unit tests*
- *Chapter 2: Readability*'s last item is Item 18: *Respect coding conventions*
- *Chapter 3: Reusability*'s last item is Item 24: *Reuse between different platforms by extracting common modules*

## How should this book be read?

How should this book be read? The way you like it. Don't bother jumping between chapters. To some degree, one builds on another, but knowledge is presented so that the chapters should be understandable independently of the others. Having said that, you should read each chapter from the beginning as each is designed to form a flow.

Choose whatever chapter you want to start with, and you can return to the others later. If you feel bored with an item or chapter, skip it. **This book was written with pleasure and should be read the same way.**

## Labels

It is impossible to write a book for everyone. This book is written primarily for experienced Kotlin developers who are already familiar with general best programming practices and looking for Kotlin-specific suggestions. Nevertheless, there are some items I decided to include even though they are not Kotlin-specific or they might seem basic for experienced developers. To make it clear which these are, I added the following labels at the beginning of such items:

- Not Kotlin-specific - this item does not have Kotlin-specific suggestions, and similar arguments might be applied to other OOP languages like Java, C#, or Swift. If you are looking for Kotlin-specific content, skip such items.
- Basics - the presented suggestions might sound basic for experienced Kotlin developers as they are already covered in other best-practices books, and they seem to be understood by the community. If a title seems clear to you, skip this item.

## **Code Sources**

The source code of all the snippets is published in the following repository:

[https://github.com/MarcinMoskala/effectivekotlin\\_sources](https://github.com/MarcinMoskala/effectivekotlin_sources)

## **Suggestions**

If you have any suggestions or corrections regarding this book, send them to  
[contact@kt.academy](mailto:contact@kt.academy)

## Acknowledgments

This book would not be so good without the great reviewers who greatly influenced it with their suggestions and comments. I would like to thank all of them. Here is the list of reviewers, starting from the most active ones.



**Márton Braun** - A Kotlin enthusiast since the version 1.0 of the language. He is currently working at Google, making Kotlin great to use for Android development. University lecturer, StackOverflow contributor, and open source library author on the side.

Márton's useful comments, suggestions, and corrections strongly influenced chapters 1 to 6. He suggested changes in the names of chapters, supported the reorganization of the book, and contributed many important ideas.



**David Blanc** - After graduating in Computer Science from INSA (a French Engineering school), David worked as a Java engineer for 8 years at various French IT companies; in 2012, he moved to mobile application development on iOS and Android. In 2015, he decided to focus on Android and joined i-BP (an IT department of the banking group BPCE) as an Android expert. He is now passionate about Android, clean code, and, of course,

Kotlin programming since version 1.0.

David gave many on-point corrections and corrected the wording for nearly all chapters. He suggested some good examples and useful ideas.



**Jordan Hansen** - Jordan has been developing on and off since he was 10. He started developing full-time when he graduated from the University of Utah. He started evaluating Kotlin version 0.6 as a viable language and has been using it as his primary language since version 0.8. He was part of an influential team that brought Kotlin to his entire organization. Jordan loves playing tabletop games with his family.

Jordan strongly influenced most of the book and provided many corrections and suggestions for snippets and titles. He suggested a deeper explanation of DSL and how the item dedicated to unit testing could be shortened. He encouraged correct technical wording.

**Juan Ignacio Vimberg** - The most active reviewer in the toughest part of this book: Part 3: Efficiency. Also strongly influenced chapters 1 to 4. He suggested showing correct benchmarks and describing Semantic Versioning.

**Kirill Bubochkin** - Gave perfectly on-point and well-thought-out comments throughout the book.

**Fabio Collini** - Great review, especially for Chapters 4 and 5.

**Bill Best** - An important reviewer who influenced chapters 6 to 8, where he left important corrections.

**Geoff Falk** - Helped improve the language, grammar, and some code snippets, especially in chapters 2 and 5.

**Danilo Herrera** - Influenced chapter 3 and greatly influenced Chapter 4: Abstraction design.

**Allan Caine** - Strongly influenced Chapter 5: Object creation.

**Edward Smith** - Strongly influenced Chapter 6: Class design.

**Juan Manuel Rivero** - Reviewed Chapters 6, 7, and 8.

I would also like to thank:

- Nicola Corti, for great suggestions and wording improvements.
- Marta Raźniewska, who made the drawings at the start of each section.
- Most active alpha testers: Pablo Guardiola, Hubert Kosacki, Carmelo Iriti and Maria Antonietta Osso.
- Everyone who helped this book by sharing news about it or sharing feedback and feelings with me.

# Part 1: Good code



# Chapter 1: Safety

Why do we decide to use Kotlin in our projects instead of Java, JavaScript, or C++? For developers, the answer is often that Kotlin is a modern language with a lot of useful features. For business, the answer is usually that Kotlin is a safe language, meaning that it is less prone to errors due to its design. You don't need to have any experience with development to get upset when an application crashes or there is an error on a website that does not let you check out after you've spent an hour adding products to your basket. Fewer crashes improve the lives of both users and developers, thus providing significant business value.

Safety is important for us, and Kotlin is a really safe language, but it still needs developer support to be truly safe. In this chapter, we'll talk about the most important best practices for safety in Kotlin. We'll see how Kotlin features promote safety and how we can use them properly. The general purpose of every item in this chapter is to produce code that is less prone to errors.

## Item 1: Limit mutability

In Kotlin, we design programs in modules, each of which comprises different kinds of elements, such as classes, objects, functions, type aliases, and top-level properties. Some of these elements can hold a state, for instance, by having a read-write `var` property or by composing a mutable object:

```
var a = 10
val list: MutableList<Int> = mutableListOf()
```

When an element holds a state, the way it behaves depends not only on how you use it but also on its history. A typical example of a class with a state is a bank account (class) that has some money balance (state):

```
class BankAccount {
    var balance = 0.0
    private set

    fun deposit(depositAmount: Double) {
        balance += depositAmount
    }

    @Throws(InsufficientFunds::class)
    fun withdraw(withdrawAmount: Double) {
        if (balance < withdrawAmount) {
            throw InsufficientFunds()
        }
        balance -= withdrawAmount
    }
}

class InsufficientFunds : Exception()

val account = BankAccount()
println(account.balance) // 0.0
account.deposit(100.0)
println(account.balance) // 100.0
account.withdraw(50.0)
println(account.balance) // 50.0
```

Here `BankAccount` has a state that represents how much money is in this account. Keeping a state is a double-edged sword. On the one hand, it is very useful because

it makes it possible to represent elements that change over time. On the other hand, state management is hard because:

1. It is harder to understand and debug a program with many mutating points. The relationship between these mutations needs to be understood, and it is harder to track how they have changed when more of them occur. A class with many mutating points that depend on each other is often really hard to understand and modify. This is especially problematic in the case of unexpected situations or errors.
2. Mutability makes it harder to reason about code. The state of an immutable element is clear, but a mutable state is much harder to comprehend. It is harder to reason about what its value is as it might change at any point; therefore, even though we might have checked a moment ago, it might have already changed.
3. A mutable state requires proper synchronization in multithreaded programs. Every mutation is a potential conflict. We will discuss this in more detail later in the next item. For now, let's just say that it is hard to manage a shared state.
4. Mutable elements are harder to test. We need to test every possible state; the more mutability there is, the more states there are to check. Moreover, the number of states we need to test generally grows exponentially with the number of mutation points in the same object or file, as we need to consider all combinations of possible states.
5. When a state mutates, other classes often need to be notified about this change. For instance, when we add a mutable element to a sorted list, if this element changes, we need to sort this list again.

The drawbacks of mutability are so numerous that there are languages that do not allow state mutation at all. These are purely functional languages, a well-known example of which is Haskell. However, such languages are rarely used for mainstream development since it's very hard to do programming with such limited mutability. A mutating state is a very useful way to represent the state of real-world systems. I recommend using mutability, but only where it gives us some real value. When possible, it is better to limit it. The good news is that Kotlin has good support for limiting mutability.

## **Limiting mutability in Kotlin**

Kotlin is designed to support limiting mutability: it is easy to make immutable objects or to keep properties immutable. This is a result of many features and characteristics of this language, the most important of which are:

- Read-only properties `val`,

- Separation between mutable and read-only collections,
- copy in data classes.

Let's discuss these one by one.

## Read-only properties

In Kotlin, we can make each property a read-only `val` (like “value”) or a read-write `var` (like “variable”). Read-only (`val`) properties cannot be set to a new value:

```
val a = 10
a = 20 // ERROR
```

Notice though that read-only properties are not necessarily immutable or final. A read-only property can hold a mutable object:

```
val list = mutableListOf(1, 2, 3)
list.add(4)

print(list) // [1, 2, 3, 4]
```

A read-only property can also be defined using a custom getter that might depend on another property:

```
var name: String = "Marcin"
var surname: String = "Moskała"
val fullName
    get() = "$name $surname"

fun main() {
    println(fullName) // Marcin Moskała
    name = "Maja"
    println(fullName) // Maja Moskała
}
```

In the above example, the value returned by the `val` changes because when we define a custom getter, it will be called every time we ask for the value.

```
fun calculate(): Int {
    print("Calculating... ")
    return 42
}

val fizz = calculate() // Calculating...
val buzz
    get() = calculate()

fun main() {
    print(fizz) // 42
    print(fizz) // 42
    print(buzz) // Calculating... 42
    print(buzz) // Calculating... 42
}
```

This trait, namely that properties in Kotlin are encapsulated by default and can have custom accessors (getters and setters), is very important in Kotlin because it gives us flexibility when we change or define an API. This will be described in detail in Item 15: *Properties should represent state, not behavior*. The core idea though is that `val` does not offer mutation points because, under the hood, it is only a getter. `var` is both a getter and a setter. That's why we can override `val` with `var`:

```
interface Element {
    val active: Boolean
}

class ActualElement : Element {
    override var active: Boolean = false
}
```

Values of read-only `val` properties can change, but such properties do not offer a mutation point, and this is the main source of problems when we need to synchronize or reason about a program. This is why we generally prefer `val` over `var`.

Remember that `val` doesn't mean immutable. It can be defined by a getter or a delegate. This fact gives us more freedom to change a final property into a property represented by a getter. However, when we don't need to use anything more complicated, we should define final properties, which are easier to reason about as their value is stated next to their definition. They are also better supported in Kotlin. For instance, they can be smart-casted:

```
val name: String? = "Márton"
val surname: String = "Braun"

val fullName: String?
    get() = name?.let { "$it $surname" }

val fullName2: String? = name?.let { "$it $surname" }

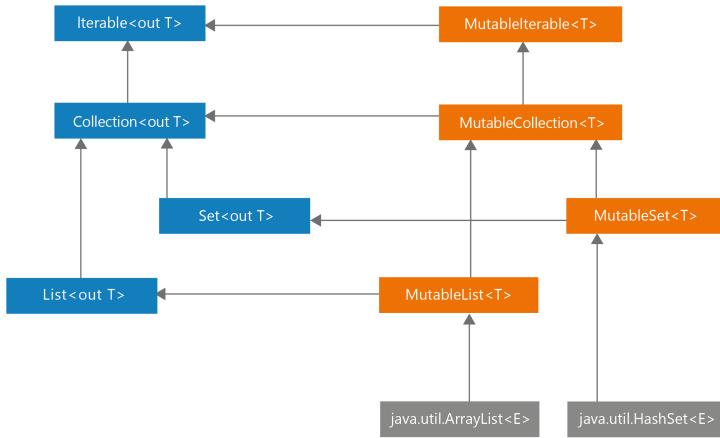
fun main() {
    if (fullName != null) {
        println(fullName.length) // ERROR
    }

    if (fullName2 != null) {
        println(fullName2.length) // 12
    }
}
```

Smart casting is impossible for `fullName` because it is defined using a getter; so, when checked it might give a different value than it does during use (for instance, if some other thread sets `name`). Non-local properties can be smart-casted only when they are final and do not have a custom getter.

## Separation between mutable and read-only collections

Similarly, just as Kotlin separates read-write and read-only properties, Kotlin also separates read-write and read-only collections. This is achieved thanks to how the hierarchy of collections was designed. Take a look at the diagram presenting the hierarchy of collections in Kotlin. On the left side, you can see the `Iterable`, `Collection`, `Set`, and `List` interfaces, all of which are read-only. This means that they do not have any methods that would allow modification. On the right side, you can see the `MutableIterable`, `MutableCollection`, `MutableSet`, and `MutableList` interfaces, all of which represent mutable collections. Notice that each mutable interface extends the corresponding read-only interface and adds methods that allow mutation. This is similar to how properties work. A read-only property means just a getter, while a read-write property means both a getter and a setter.



The hierarchy of collection interfaces in Kotlin and the actual objects that can be used in Kotlin/JVM. On the left side, the interfaces are read-only. On the right side, the collections and interfaces are mutable.

Read-only collections are not necessarily immutable. They are often mutable, but they cannot be mutated because they are hidden behind read-only interfaces. For instance, the `Iterable<T>.map` and `Iterable<T>.filter` functions return `ArrayList` (which is a mutable list) as a `List`, which is a read-only interface. In the snippet below, you can see a simplified implementation of `Iterable<T>.map` from stdlib.

```

inline fun <T, R> Iterable<T>.map(
    transformation: (T) -> R
): List<R> {
    val list = ArrayList<R>()
    for (elem in this) {
        list.add(transformation(elem))
    }
    return list
}
    
```

The design choice to make these collection interfaces read-only instead of truly immutable is very important because it gives us much more freedom. Under the hood, any actual collection can be returned as long as it satisfies the interface; therefore, we can use platform-specific collections.

The safety of this approach is close to what is achieved by having immutable collections. The only risk is when a developer tries to “hack the system” by

performing down-casting. This is something that should never be allowed in Kotlin projects. We should be able to trust that when we return a list as read-only, it is only used to read it. This is part of the contract. More about this in Part 2.

Down-casting collections not only breaks their contract and depends on implementation instead of abstraction (as we should), but it is also insecure and can lead to surprising consequences. Take a look at this code:

```
val list = listOf(1, 2, 3)

// DON'T DO THIS!
if (list is MutableList) {
    list.add(4)
}
```

The result of this operation depends on our compilation target. On JVM, `listOf` returns an instance of `Arrays.ArrayList` that implements the Java `List` interface, which has methods like `add` and `set`, so it translates to the Kotlin `MutableList` interface. However, `Arrays.ArrayList` does not implement `add` and some other operations that mutate objects. This is why the result of this code is `UnsupportedOperationException`. On different platforms, the same code could give us different results.

What is more, there is no guarantee how this will behave a year from now. The underlying collections might change; they might be replaced with truly immutable collections implemented in Kotlin that do not implement `MutableList` at all. Nothing is guaranteed. This is why **down-casting read-only collections to mutable ones should never happen in Kotlin**. If you need to transform from read-only to mutable, you should use the `List.toMutableList` function, which creates a copy that you can then modify:

```
val list = listOf(1, 2, 3)

val mutableList = list.toMutableList()
mutableList.add(4)
```

This way does not break any contract, and it is safer for us as we can feel safe that when we expose something as `List` it won't be modified from outside.

## Copy in data classes

There are many reasons to prefer immutable objects – objects that do not change their internal state, like `String` or `Int`. In addition to the previously given reasons why we generally prefer less mutability, immutable objects have their own advantages:

1. They are easier to reason about since their state stays the same once they have been created.
2. Immutability makes it easier to parallelize a program as there are no conflicts among shared objects.
3. References to immutable objects can be cached as they will not change.
4. We do not need to make defensive copies of immutable objects. When we do copy immutable objects, we do not need to make a deep copy.
5. Immutable objects are the perfect material to construct other objects, both mutable and immutable. We can still decide where mutability is allowed, and it is easier to operate on immutable objects.
6. We can add them to sets or use them as keys in maps, unlike mutable objects, which shouldn't be used this way. This is because both these collections use hash tables under the hood in Kotlin/JVM. When we modify an element that is already classified in a hash table, its classification might not be correct anymore, therefore we won't be able to find it. This problem will be described in detail in Item 43: Respect the contract of `hashCode`. We have a similar issue when a collection is sorted.

```
val names: SortedSet<FullName> = TreeSet()
val person = FullName("AAA", "AAA")
names.add(person)
names.add(FullName("Jordan", "Hansen"))
names.add(FullName("David", "Blanc"))

print(s) // [AAA AAA, David Blanc, Jordan Hansen]
print(person in names) // true

person.name = "ZZZ"
print(names) // [ZZZ AAA, David Blanc, Jordan Hansen]
print(person in names) // false
```

At the last check, the collection returned false even though that person is in this set. It couldn't be found because it is at an incorrect position.

As you can see, mutable objects are more dangerous and less predictable. On the other hand, the biggest problem of immutable objects is that data sometimes needs to change. The solution is that immutable objects should have methods that produce a copy of this object with the desired changes applied. For instance, `Int` is immutable, and it has many methods like `plus` or `minus` that do not modify it but instead return a new `Int`, which is the result of the operation. `Iterable` is read-only, and collection processing functions like `map` or `filter` do not modify it but instead return a new collection. The same can be applied to our immutable objects. For instance, let's say that we have an immutable class `User`, and we need to allow

its surname to change. We can support it with the `withSurname` method, which produces a copy with a particular property changed:

```
class User(  
    val name: String,  
    val surname: String  
) {  
    fun withSurname(surname: String) = User(name, surname)  
}  
  
var user = User("Maja", "Markiewicz")  
user = user.withSurname("Moskała")  
print(user) // User(name=Maja, surname=Moskała)
```

Writing such functions is possible but it's also tedious if we need one for every property. So, here comes the `data` modifier to the rescue. One of the methods it generates is `copy`. The method `copy` creates a new instance in which all primary constructor properties are, by default, the same as in the previous one. New values can be specified as well. `copy` and other methods generated by the `data` modifier are described in detail in Item 37: *Use the `data` modifier to represent a bundle of data.* Here is a simple example showing how it works:

```
data class User(  
    val name: String,  
    val surname: String  
)  
  
var user = User("Maja", "Markiewicz")  
user = user.copy(surname = "Moskała")  
print(user) // User(name=Maja, surname=Moskała)
```

This elegant and universal solution supports making data model classes immutable. This way is less efficient than just using a mutable object instead, but it is safer and has all the other advantages of immutable objects. Therefore it should be preferred by default.

## Different kinds of mutation points

Let's say that we need to represent a mutating list. There are two ways we can achieve this: either by using a mutable collection or by using the read-write `var` property:

```
val list1: MutableList<Int> = mutableListOf()
var list2: List<Int> = listOf()
```

Both properties can be modified, but in different ways:

```
list1.add(1)
list2 = list2 + 1
```

Both of these ways can be replaced with the plus-assign operator, but each of them is translated into a different behavior:

```
list1 += 1 // Translates to list1.plusAssign(1)
list2 += 1 // Translates to list2 = list2.plus(1)
```

Both these ways are correct, and both have their pros and cons. They both have a single mutating point, but each is located in a different place. In the first one, the mutation takes place on the concrete list implementation. We might depend on the fact that the collection has proper synchronization in the case of multithreading, if we used a collection with support for concurrency<sup>7</sup>. In the second one, we need to implement the synchronization ourselves, but the overall safety is better because the mutating point is only a single property. However, in the case of a lack of synchronization, remember that we might still lose some elements:

```
var list = listOf<Int>()
for (i in 1..1000) {
    thread {
        list = list + i
    }
}
Thread.sleep(1000)
print(list.size) // Very unlikely to be 1000,
// every time a different number, like for instance 911
```

Using a mutable property instead of a mutable list allows us to track how this property changes when we define a custom setter or use a delegate (which uses a custom setter). For instance, when we use an observable delegate, we can log every change of a list:

---

<sup>7</sup>We will discuss such collections in the next item.

```
var names by observable(listOf<String>()) { _, old, new ->
    println("Names changed from $old to $new")
}

names += "Fabio"
// Names changed from [] to [Fabio]
names += "Bill"
// Names changed from [Fabio] to [Fabio, Bill]
```

To make this possible for a mutable collection, we would need a special observable implementation of the collection. For read-only collections in mutable properties, it is also easier to control how they change as there is only a setter instead of multiple methods mutating this object, and we can make it private:

```
var announcements = listOf<Announcement>()
private set
```

In short, using mutable collections is a slightly faster option, but using a mutable property instead gives us more control over how the object changes.

Notice that the worst solution is to have both a mutating property and a mutable collection:

```
// Don't do that
var list3 = mutableListOf<Int>()
```

The general rule is that **one should not create unnecessary ways to mutate a state**. Every way to mutate a state is a cost. Every mutation point needs to be understood and maintained. We prefer to limit mutability.

## Summary

In this chapter, we've learned why it is important to limit mutability and to prefer immutable objects. We've seen that Kotlin gives us many tools that support limiting mutability. We should use them to limit mutation points. The simple rules are:

- Prefer `val` over `var`.
- Prefer an immutable property over a mutable one.
- Prefer objects and classes that are immutable over mutable ones.
- If you need immutable objects to change, consider making them data classes and using `copy`.

- When you hold a state, prefer read-only over mutable collections.
- Design your mutation points wisely and do not produce unnecessary ones.

There are some exceptions to these rules. Sometimes we prefer mutable objects because they are more efficient. Such optimizations should be preferred only in performance-critical parts of our code (Part 3: Efficiency); when we use them, we need to remember that mutability requires more attention when we prepare it for multithreading. The baseline is that we should limit mutability.

## Item 2: Eliminate critical sections

When multiple threads modify a shared state, it can lead to unexpected results. This problem was already discussed in the previous item, but now I want to explain it in more detail and show how to deal with it in Kotlin/JVM.

### The problem with threads and shared state

While I'm writing these words, many things are happening concurrently on my computer. Music is playing, IntelliJ displays the text of this chapter, Slack is displaying messages, and my browser is downloading data. All this is possible because operating systems introduced the concept of threads. The operating system schedules the execution of threads, each of which is a separate flow. Even if I had a single-core CPU, the operating system would still be able to run multiple threads concurrently by running one thread for a short period of time, then switching to another thread, and so on. This is called *time slicing*. What is more, in modern computers we have multiple cores, so operating systems can actually run many operations on different threads at the same time.

The biggest problem with this process is that we cannot be sure when the operating system will switch from executing one thread to executing another. Consider the following example. We start 1000 threads, each of which increments a mutable variable; the problem is that incrementing a value has multiple steps: getting the current value, creating the new incremented value, and assigning it to the variable. If the operating system switches threads between these steps, we might lose some increments. This is why the code below is unlikely to print 1000. I just tested it, and it printed 981.

```
var num = 0
for (i in 1..1000) {
    thread {
        Thread.sleep(10)
        num += 1
    }
}
Thread.sleep(5000)
print(num) // Very unlikely to be 1000
// Every time a different number
```

To better understand this problem, just consider the following situation that might occur if we had started two threads. One thread gets value 0, then the CPU switches execution to the other thread, which gets the same value, increments it,

and sets the variable to 1. The operating system switches to the previous thread, which then sets the variable to 1 again. In this case, we've lost one incrementation.

Losing some operations can be a serious problem in real-life applications, but this problem can have much more serious consequences. When we don't know the order in which operations will be executed, we risk our objects having incorrect states. This often leads to bugs that are hard to reproduce and fix, as is well visualized by adding an element to a list while another thread iterates over its elements. The default collections do not support their elements being modified when they are iterated over, so we get a `ConcurrentModificationException` exception.

```
var numbers = mutableListOf<Int>()
for (i in 1..1000) {
    thread {
        Thread.sleep(1)
        numbers.add(i)
    }
    thread {
        Thread.sleep(1)
        print(numbers.sum()) // sum iterates over the list
        // often ConcurrentModificationException
    }
}
```

We encounter the same problem when we start multiple coroutines on a dispatcher that uses multiple threads. To deal with this problem when using coroutines, we can use the same techniques as for threads. However, coroutines also have dedicated tools, as I described in detail in the book *Kotlin Coroutines: Deep Dive*.

As I explained in the previous chapter, we don't encounter all these problems if we don't use mutability. However, in real-life applications we often cannot avoid mutability, so we need to learn how to deal with shared state<sup>8</sup>. **Whenever you have a shared state that might be modified by multiple threads, you need to ensure that all the operations on this state are executed correctly.** Each platform offers different tools for this, so let's learn about the most important tools for Kotlin/JVM<sup>9</sup>.

---

<sup>8</sup>By shared state, I mean a state used by multiple threads.

<sup>9</sup>In Kotlin/JS, we don't need to worry about synchronization because JavaScript execution is single-threaded: if you start a process on a different thread (for instance, using workers), it operates in a different memory space.

## Synchronization in Kotlin/JVM

The most important tool for dealing with shared state in the Kotlin/JVM platform is synchronization. This is a mechanism that allows us to ensure that only one thread can execute a given block of code at the same time. It is based on the `synchronized` function, which requires a lock object and a lambda expression with the code that should be synchronized. This mechanism guarantees that only one thread can enter a synchronization block with the same lock at the same time. If a thread reaches a synchronization block but a different thread is already executing a synchronization block with the same lock, this thread will wait until the other thread finishes its execution. The following example shows how to use synchronization to ensure that the `num` variable is incremented correctly.

```
val lock = Any()
var num = 0
for (i in 1..1000) {
    thread {
        Thread.sleep(10)
        synchronized(lock) {
            num += 1
        }
    }
}
Thread.sleep(1000)
print(num) // 1000
```

In real-life cases, we often wrap all the functions in a class that need to be synchronized with a synchronization block. The example below shows how to synchronize all the operations in the `Counter` class.

```
class Counter {
    private val lock = Any()
    private var num = 0

    fun inc() = synchronized(lock) {
        num += 1
    }

    fun dec() = synchronized(lock) {
        num -= 1
    }
}
```

```
// Synchronization is not necessary; however,  
// without it, getter might serve stale value  
fun get(): Int = num  
}
```

In some classes, we have multiple locks for different parts of a state, but this is much harder to implement correctly, so it's much less common.

When we use Kotlin Coroutines, instead of using `synchronized`, we rather use a dispatcher limited to a single thread or Mutex, as I described that in the book *Kotlin Coroutines: Deep Dive*. Remember that thread-switching is not free, and in some classes it is more efficient to use a single thread instead of using multiple threads and synchronizing their execution.

## Atomic objects

We started our discussion with the problem of incrementing a variable, which can produce incorrect results because regular integer incrementation has multiple steps, but the operating system can switch between threads in the middle of these. Some operations, such as a simple value assignment, are only a single processor step, so they are always executed correctly, but only very simple operations are atomic by nature. However, Java provides a set of atomic classes that represent popular Java classes but with atomic operations. You can find `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference`, and many more. Each of these offers methods that are guaranteed to be executed atomically. For example, `AtomicInteger` offers an `incrementAndGet` method that increments a value and returns the new value. The example below shows how to use `AtomicInteger` to increment a variable correctly.

```
val num = AtomicInteger(0)  
for (i in 1..1000) {  
    thread {  
        Thread.sleep(10)  
        num.incrementAndGet()  
    }  
}  
Thread.sleep(5000)  
print(num.get()) // 1000
```

Atomic objects are fast and can help us with simple cases where a state is a simple value or a couple of independent values, but these are not enough for more complex cases. For example, we cannot use atomic objects to synchronize multiple operations on multiple objects. For that, we need to use a synchronization block.

## Concurrent collections

Java also provides some collections that have support for concurrency. The most important one is `ConcurrentHashMap`, which is a thread-safe version of `HashMap`. We can safely use all its operations without worrying about conflicts. When we iterate over it, we get a snapshot of the state at the moment of iteration, therefore we'll never get a `ConcurrentModificationException` exception, but this doesn't mean that we'll get the most recent state.

```
val map = ConcurrentHashMap<Int, String>()
for (i in 1..1000) {
    thread {
        Thread.sleep(1)
        map.put(i, "E$i")
    }
    thread {
        Thread.sleep(1)
        print(map.toList().sumOf { it.first })
    }
}
```

When we need a concurrent set, a popular choice is to use `newKeySet` from `ConcurrentHashMap`, which is a wrapper over `ConcurrentHashMap` that uses `Unit` as a value. It implements the `MutableSet` interface, so we can use it like a regular set.

```
val set = ConcurrentHashMap.newKeySet<Int>()
for (i in 1..1000) {
    thread {
        Thread.sleep(1)
        set += i
    }
}
Thread.sleep(5000)
println(set.size)
```

Instead of lists, I typically use `ConcurrentLinkedQueue` when I need a concurrent collection that allows duplicates. These are the essential tools that we can use on JVM to deal with the problem of mutable states.

Of course, there are also libraries that offer other tools that support code synchronization. There are even Kotlin multiplatform libraries, such as `AtomicFU`, which provides multiplatform atomic objects<sup>10</sup>.

---

<sup>10</sup>At the time of writing these words, `AtomicFU` is still in beta version, but it is already well-developed and seems rather stable.

```
// Using AtomicFU
val num = atomic(0)
for (i in 1..1000) {
    thread {
        Thread.sleep(10)
        num.incrementAndGet()
    }
}
Thread.sleep(5000)
print(num.value) // 1000
```

Let's change our perspective back to the more general problem with mutable states and explain how to deal with it in typical situations.

## Do not leak mutation points

Exposing a mutable object that is used to represent a public state, like in the following examples, is an especially dangerous situation. Take a look at this example:

```
data class User(val name: String)

class UserRepository {
    private val users: MutableList<User> = mutableListOf()

    fun loadAll(): MutableList<User> = users

    //...
}
```

One could use `loadAll` to modify the `UserRepository` private state:

```
val userRepository = UserRepository()

val users = userRepository.loadAll()
users.add(User("Kirill"))
//...

print(userRepository.loadAll()) // [User(name=Kirill)]
```

This situation is especially dangerous when such modifications are accidental. The first thing we should do is upcast the mutable objects to read-only types; in this case it means upcasting from `MutableList` to `List`.

```
data class User(val name: String)

class UserRepository {
    private val users: MutableList<User> = mutableListOf()

    fun loadAll(): List<User> = users

    // ...
}
```

But beware, because the implementation above is not enough to make this class safe. First, we receive what looks like a read-only list, but it is actually a reference to a mutable list, so its values might change. This might cause developers to make serious mistakes:

```
data class User(val name: String)

class UserRepository {
    private val users: MutableList<User> = mutableListOf()

    fun loadAll(): List<User> = users

    fun add(user: User) {
        users += user
    }
}

class UserRepositoryTest {
    fun `should add elements`() {
        val repo = UserRepository()
        val oldElements = repo.loadAll()
        repo.add(User("B"))
        val newElements = repo.loadAll()
        assert(oldElements != newElements)
        // This assertion will fail, because both references
        // point to the same object, and they are equal
    }
}
```

Second, consider a situation in which one thread reads the list returned using `loadAll`, but another thread modifies it at the same time. It is illegal to modify a mutable collection that another thread is iterating over. Such an operation leads to an unexpected exception.

```
val repo = UserRepository()
thread {
    for (i in 1..10000) repo.add(User("User$i"))
}
thread {
    for (i in 1..10000) {
        val list = repo.loadAll()
        for (e in list) {
            /* no-op */
        }
    }
}
// ConcurrentModificationException
```

There are two ways of dealing with this. The first is to return a copy of an object instead of a real reference. We call this technique *defensive copying*. Note that when we copy, we might have a conflict if another thread is adding a new element to the list while we are copying it; so, if we want to support multithreaded access to our object, this operation needs to be synchronized. Collections can be copied with transformation functions like `toList`, while data classes can be copied with the `copy` method.

```
class UserRepository {
    private val users: MutableList<User> = mutableListOf()
    private val lock = Any()

    fun loadAll(): List<User> = synchronized(lock) {
        users.toList()
    }

    fun add(user: User) = synchronized(lock) {
        users += user
    }
}
```

A simpler option is to use a read-only list as this is easier to secure and gives us more ways of tracking changes in objects.

```
class UserRepository {  
    private var users: List<User> = listOf()  
  
    fun loadAll(): List<User> = users  
  
    fun add(user: User) {  
        users = users + user  
    }  
}
```

When we use this option, and we want to introduce proper support for multi-threaded access, we only need to synchronize the operations that modify our list. This makes adding elements slower, but accessing the list is faster. This is a good trade-off when we have more reads than writes.

```
class UserRepository {  
    private var users: List<User> = listOf()  
    private val lock = Any()  
  
    fun loadAll(): List<User> = users  
  
    fun add(user: User) = synchronized(lock) {  
        users = users + user  
    }  
}
```

## Summary

- Multiple threads modifying the same state can lead to conflicts, thus causing lost data, exceptions, and other unexpected behavior.
- We can use synchronization to protect a state from concurrent modifications. The most popular tool in Kotlin/JVM is a synchronized block with a lock.
- To deal with concurrent modifications, Java also provides classes to represent atomic values and concurrent collections.
- There are also libraries that provide multiplatform atomic objects, such as AtomicFU.
- Classes should protect their internal state and not expose it to the outside world. We can operate on read-only objects or use defensive copying to protect a state from concurrent modifications.

## Item 3: Eliminate platform types as soon as possible

The null safety introduced by Kotlin is amazing. Java was known in the community for Null Pointer Exceptions (NPE), which Kotlin's safety mechanisms make rare or eliminate entirely. However, one thing that cannot be secured completely is a connection between Kotlin and a language that does not have solid null safety, such as Java or C. Imagine that we use a Java method that declares `String` as a return type. What type should it be in Kotlin?

If it is annotated with the `@Nullable` annotation, then we assume it is nullable and we interpret it as `String?`. If it is annotated with `@NotNull`, then we trust this annotation and we type it as `String`. However, what if this return type is not annotated with either of these annotations?

```
// Java
public class JavaTest {

    public String giveName() {
        // ...
    }
}
```

We might say that we should treat such a type as nullable. This would be a safe approach since everything is nullable in Java. However, we often know that something is not `null`, so we would end up using the non-null assertion `!!` in many places all around our code.

The real problem would arise when we need to take generic types from Java. Imagine that a Java API returns a `List<User>` that is not annotated at all. If Kotlin assumed nullable types by default and we did know that this list and those users are not `null`, we would need to not only assert the whole list but also filter the `null`s:

```
// Java
public class UserRepo {

    public List<User> getUsers() {
        /**
     }
}
```

```
// Kotlin
val users: List<User> = UserRepo().users!!.filterNotNull()
```

What if a function returned a `List<List<User>>` instead? This gets complicated:

```
val users: List<List<User>> = UserRepo().groupedUsers!!
    .map { it!!.filterNotNull() }
```

Lists at least have functions like `map` and `filterNotNull`. In other generic types, nullability would be an even bigger problem. This is why instead of being treated as nullable by default, a type that comes from Java and has unknown nullability is a special type in Kotlin: it is called a *platform type*.

Platform type - a type that comes from another language and has unknown nullability.

Platform types are notated with a single exclamation mark ! after the type name, such as `String!`. Platform types are non-denotable, meaning that one cannot write them explicitly in code. When a value with a platform type is assigned to a Kotlin variable or property, its type can be inferred, but it cannot be explicitly specified. Instead, we can specify either a nullable or a non-nullable type.

```
// Java
public class UserRepo {
    public User getUser() {
        //...
    }
}
```

```
// Kotlin
val repo = UserRepo()
val user1 = repo.user           // Type of user1 is User!
val user2: User = repo.user    // Type of user2 is User
val user3: User? = repo.user   // Type of user3 is User?
```

Thanks to this, getting generic types from Java is not problematic:

```
val users: List<User> = UserRepo().users
val users: List<List<User>> = UserRepo().groupedUsers
```

Casting platform types to non-nullable types is better than not specifying a type at all, but it is still dangerous because something we assumed to be non-null might be null. This is why, for safety reasons, I always suggest being very careful when we get platform types from Java. Remember that even if a function does not return `null` now, that doesn't mean that it won't change in the future. If its designer hasn't specified it with an annotation or by describing it in a comment, this behavior can still be introduced without changing a contract.

If you have some control over Java code that needs to interoperate with Kotlin, introduce `@Nullable` and `@NotNull` annotations wherever possible.

```
// Java

import org.jetbrains.annotations.NotNull;

public class UserRepo {
    public @NotNull User getUser() {
        //...
    }
}
```

This is one of the most important steps when we want to support Kotlin developers well (and it's also important information for Java developers). Annotating many exposed types was one of the most important changes that were introduced in the Android API after Kotlin became a first-class citizen. This made the Android API much more Kotlin-friendly.

Note that many different kinds of annotations are supported, including those by:

- JetBrains (`@Nullable` and `@NotNull` from `org.jetbrains.annotations`).
- Android (`@Nullable` and `@Nonnull` from `androidx.annotation` as well as from `com.android.annotations` and from the `android.support.annotations`) support library.
- JSR-305 (`@Nullable`, `@CheckForNull` and `@Nonnull` from `javax.annotation`).
- JavaX (`@Nullable`, `@CheckForNull`, `@Nonnull` from `javax.annotation`).
- FindBugs (`@Nullable`, `@CheckForNull`, `@PossiblyNull` and `@NonNull` from `edu.umd.cs.findbugs.annotations`).
- ReactiveX (`@Nullable` and `@NonNull` from `io.reactivex.annotations`).
- Eclipse (`@Nullable` and `@NonNull` from `org.eclipse.jdt.annotation`).
- Lombok (`@NonNull` from `lombok`).

Alternatively, in Java you can specify that all types should be non-nullable by default using JSR 305's `@ParametersAreNonnullByDefault` annotation.

There is something we can do in our Kotlin code as well. My recommendation is to eliminate these platform types as soon as possible for safety reasons. To understand why, think about the difference between how the `statedType` and `platformType` functions behave in this example:

```
// Java
public class JavaClass {
    public String getValue() {
        return null;
    }
}

// Kotlin
fun statedType() {
    val value: String = JavaClass().value
    //...
    println(value.length)
}

fun platformType() {
    val value = JavaClass().value
    //...
    println(value.length)
}
```

In both cases, the developer assumed that `getValue` would not return `null`, but this is wrong because it results in an NPE in both cases, but there's a difference in where this error happens.

In `statedType`, the NPE will be thrown in the same line where we get the value from Java. It would be absolutely clear that we wrongly assumed a non-null type and we got `null`. We would just need to change it and adjust the rest of our code to this change.

In `platformType`, the NPE will be thrown when we use this value as non-null (possibly from the middle of a more complex expression). A variable typed as a platform type can be treated as both nullable and non-nullable. Such a variable might be used a few times safely, but then it gets used unsafely and throws an NPE. When we use such properties, the type system does not protect us. This is a similar situation as in Java, but in Kotlin we do not expect to cause an NPE just by using an object. It is very likely that sooner or later someone will unsafely use a variable type as a platform type, and then we will end up with a runtime exception whose cause might not be easy to find.

```
// Java
public class JavaClass {
    public String getValue() {
        return null;
    }
}

// Kotlin
fun statedType() {
    val value: String = JavaClass().value // NPE
    //...
    println(value.length)
}

fun platformType() {
    val value = JavaClass().value
    //...
    println(value.length) // NPE
}
```

Even more dangerously, a platform type might be propagated further. For instance, we might expose a platform type as a part of our interface:

```
interface UserRepo {
    fun getUserName() = JavaClass().value
}
```

In this case, the method's inferred type is a platform type. This means that anyone can still decide if it is nullable or not. One might choose to treat it as nullable in a definition site, and as a non-nullable in the use site:

```
class RepoImpl : UserRepo {
    override fun getUserName(): String? {
        return null
    }
}

fun main() {
    val repo: UserRepo = RepoImpl()
    val text: String = repo.getUserName() // NPE in runtime
    print("User name length is ${text.length}")
}
```

Propagating a platform type is a recipe for disaster. They are problematic, so for safety reasons we should always eliminate them as soon as possible. In this case, IDEA IntelliJ helps us with a warning:



A screenshot of the IntelliJ IDEA code editor. The code shown is:

```
1 2 3 interface UserRepository {  
    fun.getUserName() = JavaClass().value  
}
```

A tooltip window is displayed at the bottom right of the code area, containing the text: "Declaration has type inferred from a platform call, which can lead to unchecked nullability issues. Specify type explicitly as nullable or non-nullable. more... (%F1)".

## Summary

Types that come from another language and have unknown nullability are known as platform types. Since they are dangerous, we should eliminate them as soon as possible and not let them propagate. It is also good to specify types using annotations that specify nullability for exposed Java constructors, methods and fields. This is precious information for Java and Kotlin developers who use these elements.

## Item 4: Minimize the scope of variables

When we define a state, we prefer to tighten the scope of variables and properties by:

- Using local variables instead of properties.
- Using variables in the narrowest scope possible, for instance by defining a variable inside a loop if it is used only inside this loop.

The scope of an element is the region of a computer program where this element is visible. In Kotlin, the scope is nearly always created by curly braces. The rule is: we can access elements from the same scope and the outer scopes. Take a look at this example:

```
val a = 1
fun fizz() {
    val b = 2
    print(a + b)
}
val buzz = {
    val c = 3
    print(a + c)
}
// Here we can see a, but not b nor c
```

In the above example, in the scope of the functions `fizz` and `buzz`, we can access variables from the outer scope. However, in the outer scope, we cannot access the variables defined in these functions. Here is an example of how limiting a variable's scope might look:

```
// Bad
var user: User
for (i in users.indices) {
    user = users[i]
    print("User at $i is $user")
}

// Better
for (i in users.indices) {
    val user = users[i]
    print("User at $i is $user")
```

```
}
```

```
// Same variables scope, nicer syntax
for ((i, user) in users.withIndex()) {
    print("User at $i is $user")
}
```

In the first example, the `user` variable is accessible not only in the scope of the for-loop but also outside of it. In the second and third examples, we limit the scope of the `user` variable concretely to the scope of the for-loop.

Similarly, we might have many scopes inside scopes (most likely created by lambda expressions inside lambda expressions), but it is better to define variables in as narrow a scope as possible.

There are many reasons why we prefer it this way, but the most important one is: **When we tighten a variable's scope, it is easier to keep our programs simple to track and manage.** When we analyze code, we need to think about what elements are there at this point. The more elements there are to deal with, the harder it is to do programming. The simpler your application is, the less likely it is to break. This is a similar reason to why we prefer immutable properties or objects over their mutable counterparts.

**It is easier to track how mutable properties change when they can only be modified in a small scope.** It is easier to reason about them and change their behavior.

Another problem is that **variables with a wide scope might be overused by another developer.** For instance, one could reason that if a variable is used to assign the next elements in an iteration, the last element in the list should remain in that variable once the loop is complete. Such reasoning could lead to terrible abuse, such as using this variable after the iteration to do something with the last element. This would be really bad because another developer who is trying to understand what value is there would need to understand the whole reasoning. This would be a needless complication.

**Whether a variable is read-only or read-write, we always prefer it to be initialized when it is defined.** Don't force another developer to look at where it was defined. This can be supported with control structures such as `if`, `when`, `try-catch`, or the Elvis operator used as an expression:

```
// Bad
val user: User
if (hasValue) {
    user = getValue()
} else {
    user = User()
}

// Better
val user: User = if (hasValue) {
    getValue()
} else {
    User()
}
```

If we need to set up multiple properties, destructuring declarations can help us:

```
// Bad
fun updateWeather(degrees: Int) {
    val description: String
    val color: Int
    if (degrees < 5) {
        description = "cold"
        color = Color.BLUE
    } else if (degrees < 23) {
        description = "mild"
        color = Color.YELLOW
    } else {
        description = "hot"
        color = Color.RED
    }
    // ...
}

// Better
fun updateWeather(degrees: Int) {
    val (description, color) = when {
        degrees < 5 -> "cold" to Color.BLUE
        degrees < 23 -> "mild" to Color.YELLOW
        else -> "hot" to Color.RED
    }
}
```

```
// ...
}
```

Finally, a variable scope that is too wide can be dangerous. Let's describe one common danger.

## Capturing

When I teach about Kotlin Coroutines, one of my exercises involves implementing the Sieve of Eratosthenes to find prime numbers using a sequence builder. The algorithm is conceptually simple:

1. We take a list of numbers starting from 2.
2. We take the first one. It is a prime number.
3. From the rest of the numbers, we remove the first one and filter out all the numbers that are divisible by this prime number.

A very simple implementation of this algorithm looks like this:

```
var numbers = (2..100).toList()
val primes = mutableListOf<Int>()
while (numbers.isNotEmpty()) {
    val prime = numbers.first()
    primes.add(prime)
    numbers = numbers.filter { it % prime != 0 }
}
print(primes) // [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
// 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

The challenge is to let it produce a potentially infinite sequence of prime numbers. If you want to challenge yourself, stop now and try to implement it.

This is what the solution could look like:

```

val primes: Sequence<Int> = sequence {
    var numbers = generateSequence(2) { it + 1 }

    while (true) {
        val prime = numbers.first()
        yield(prime)
        numbers = numbers.drop(1)
            .filter { it % prime != 0 }
    }
}

print(primes.take(10).toList())
// [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

```

In nearly every group, there is a person who tries to “optimize” this code by extracting `prime` as a mutable variable in order to avoid creating the variable in every loop:

```

val primes: Sequence<Int> = sequence {
    var numbers = generateSequence(2) { it + 1 }

    var prime: Int
    while (true) {
        prime = numbers.first()
        yield(prime)
        numbers = numbers.drop(1)
            .filter { it % prime != 0 }
    }
}

```

The problem is that this implementation no longer works correctly. These are the first 10 yielded numbers:

```

print(primes.take(10).toList())
// [2, 3, 5, 6, 7, 8, 9, 10, 11, 12]

```

Stop now and try to explain this result.

The reason why we have such a result is that we captured the variable `prime`. Filtering is done lazily because we’re using a sequence. In every step, we add more and more filters. In the “optimized” version, we always add the filter which references the mutable property `prime`. Therefore, we always filter the last value

of `prime`. This is why this filtering does not work properly. Only the `drop` step works, so we end up with consecutive numbers (except for 4, which was filtered out when `prime` was still set to 2).

We should be aware of problems with unintentional capturing. To prevent them we should avoid mutability and prefer a narrower scope for variables.

## Summary

For many reasons, we should prefer to define variables for the closest possible scope. Also, we should prefer `val` over `var` for local variables, and we should always be aware of the fact that variables are captured in lambdas. These simple rules can save us a lot of trouble.

## Item 5: Specify your expectations for arguments and state

When you have expectations, declare them as soon as possible. In Kotlin, we mainly do this using:

- require block - a universal way to specify expectations for arguments.
- check block - a universal way to specify expectations for state.
- error function - a universal way to signal that application reached an unexpected state.
- The Elvis operator with `return` or `throw`.

Here is an example that uses these mechanisms:

```
// Part of Stack<T>
fun pop(num: Int = 1): List<T> {
    require(num <= size) {
        "Cannot remove more elements than current size"
    }
    check(isOpen) { "Cannot pop from closed stack" }
    val ret = collection.take(num)
    collection = collection.drop(num)
    return ret
}
```

Specifying expectations this way does not free us from the necessity of specifying them in documentation, but it is really helpful anyway. Such declarative checks have many advantages:

- Expectations are visible even to programmers who have not read the documentation.
- If they are not satisfied, a function throws an exception instead of leading to unexpected behavior. It is important that these exceptions are thrown before the state is modified because this means we don't have a situation where only some modifications are applied and others are not. Such situations are dangerous and hard to manage<sup>11</sup>. Thanks to assertive checks,

---

<sup>11</sup>I remember how, in a Gameboy Pokemon game, one could copy a pokemon by making a transfer and disconnecting the cable at the right moment. After that, this pokemon was present on both Gameboys. Similar hacks worked in many games and they generally involved turning off the Gameboy at the correct moment. The general solution to all such problems is to make connected transactions atomic: either all occur or none occur. For instance, when we add money to one account and subtract it from another. Atomic transactions are supported by most databases.

errors are harder to miss and our state is more stable.

- Code is self-checking to some degree. There is less need for unit-testing when these conditions are checked in the code.
- All checks listed above work with smart casting, therefore less casting is required.

Let's talk about different kinds of checks and why we need them. Starting from the most popular one: the arguments check.

## Arguments

When you define a function with arguments, it is not uncommon for these arguments to have some expectations on them that cannot be expressed using the type system. Just take a look at a few examples:

- When you calculate the factorial of a number, you might require this number to be a positive integer.
- When you look for clusters, you might require a list of points to not be empty.
- When you send an email, you might require that the email address is valid.

The most universal and direct way to state these requirements in Kotlin is by using the `require` function, which checks this requirement and throws `IllegalArgumentException` if it is not satisfied:

```
fun factorial(n: Int): Long {
    require(n >= 0)
    return if (n <= 1) 1 else factorial(n - 1) * n
}

fun findClusters(points: List<Point>): List<Cluster> {
    require(points.isNotEmpty())
    //...
}

fun sendEmail(user: User, message: String) {
    requireNotNull(user.email)
    require(isValidEmail(user.email))
    //...
}
```

Notice that these requirements are highly visible because they are declared at the very beginning of functions, therefore they are clear for a user who is reading these functions (but requirements should also be stated in documentation because not everyone reads function bodies).

These expectations cannot be ignored because the `require` function throws an exception when the predicate is not satisfied. When such a block is placed at the beginning of a function, we know that if an argument is incorrect, the function will stop immediately and the user won't miss the fact that they are using this function incorrectly. An exception will be clear, unlike a potentially strange result that might propagate a long way until it fails. In other words, when we properly specify our expectations for arguments at the beginning of a function, we can then assume that these expectations will be satisfied.

We can also specify a lazy message for this exception in a lambda expression after the call:

```
fun factorial(n: Int): Long {
    require(n >= 0) {
        "Cannot calculate factorial of $n " +
        "because it is smaller than 0"
    }
    return if (n <= 1) 1 else factorial(n - 1) * n
}
```

I often see `require` inside `init` block of data classes. It is used to make sure that constructor arguments are correct, by making it impossible to create invalid instances according to the requirements.

```
data class User(
    val name: String,
    val email: String
) {
    init {
        require(name.isNotEmpty())
        require(isValidEmail(email))
    }
}
```

The `require` function is used when we specify requirements for arguments. Another common case is when we have expectations of the current state; in such a case, we can use the `check` function instead, which throws `IllegalStateException`.

## State

It is not uncommon that we only allow some functions to be used in concrete conditions. A few common examples:

- Some functions might need an object to be initialized first.
- Some actions might be allowed only if the user is logged in.
- Some functions might require an object to be open.

The standard way to check that these expectations of a state are satisfied is to use the `check` function:

```
fun speak(text: String) {
    check(isInitialized)
    // ...
}

fun getUserInfo(): UserInfo {
    checkNotNull(token)
    // ...
}

fun next(): T {
    check(isOpen)
    // ...
}
```

The `check` function works similarly to `require`, but it throws an `IllegalStateException` when the stated expectation is not met. It checks if a state is correct. An exception message can be customized using a lazy message, just like with `require`. When the expectation is on the whole function, we place it at the beginning, generally after the `require` blocks. However, some state expectations are local, and `check` can be used later.

We use such checks especially when we suspect that a user might break our contract and call a function when it should not be called. Instead of trusting that they won't do that, it is better to check and throw an appropriate exception. We might also use explicit checks when we do not trust that our own implementation handles the state correctly. However, for such cases, when we are checking mainly for the sake of testing our own implementation, we have another function called `assert`.

## Nullability and smart casting

Both `require` and `check` have Kotlin contracts that state that when a function returns, its predicate is true after this check.

```
public inline fun require(value: Boolean): Unit {  
    contract {  
        returns() implies value  
    }  
    require(value) { "Failed requirement." }  
}
```

Everything that is checked in these blocks will later be treated as true in the same function. This works well with smart casting because once we have checked that something is true, the compiler will treat it as something that is certain. In the example below, we require a person's outfit to be a `Dress`. After that, assuming that the `outfit` property is final, it will be smart casted to `Dress`.

```
fun changeDress(person: Person) {  
    require(person.outfit is Dress)  
    val dress: Dress = person.outfit  
    // ...  
}
```

This characteristic is especially useful when we check if something is null:

```
class Person(val email: String?)  
  
fun sendEmail(person: Person, message: String) {  
    require(person.email != null)  
    val email: String = person.email  
    // ...  
}
```

For such cases, we even have special functions: `requireNotNull` and `checkNotNull`. They both have the capability to smart cast variables, and they can also be used as expressions to “unpack” variables:

```

class Person(val email: String?)

fun validateEmail(email: String) { /*...*/ }

fun sendEmail(person: Person, text: String) {
    val email = requireNotNull(person.email)
    validateEmail(email)
    ...
}

fun sendEmail(person: Person, text: String) {
    requireNotNull(person.email)
    validateEmail(person.email)
    ...
}

```

## The problems with the non-null assertion !!

Instead of using `requireNotNull` or `checkNotNull` we can use the non-null assertion `!!` operator. This is conceptually similar to what happens in Java: we think something is not `null`, and an NPE is thrown if we are wrong. **The non-null assertion `!!` is a lazy option.** It throws a generic `NullPointerException` exception that explains nothing. It is also short and simple, which makes it easy to abuse or misuse. The non-null assertion `!!` is often used in situations where a type is nullable but `null` is not expected. The problem is that even if this is not currently expected, it almost always can be in the future, and this operator only quietly hides the nullability.

A very simple example is a function that looks for the largest of 4 arguments<sup>12</sup>. Let's say that we decided to implement it by packing all these arguments into a list and then using `maxOrNull` to find the biggest one. The problem is that this returns nullable because it returns `null` when the collection is empty. Only a developer who knows that this list cannot be empty will likely use a non-null assertion `!!`:

```

fun largestOf(a: Int, b: Int, c: Int, d: Int): Int =
    listOf(a, b, c, d).maxOrNull()!!

```

As was shown to me by Márton Braun, a reviewer of this book, a non-null assertion `!!` can even lead to an NPE in such a simple function. Someone might need to refactor this function to accept any number of arguments, but this person might forget that a collection cannot be empty if we want to use `maxOrNull` on it:

---

<sup>12</sup>In the Kotlin Standard Library, this function is called `maxOf`, but it accepts any number of arguments.

```
fun largestOf(vararg nums: Int): Int =  
    nums.maxOrNull()!!  
  
largestOf() // NPE
```

In the example above, the information about nullability was silenced and can easily be missed when it might be important. It's a similar situation with variables. Let's say that you have a variable that needs to be set later but will surely be set before its first use. Setting it to `null` and using a non-null assertion `!!` is not a good option. It is annoying that we need to unpack these properties every time, and we also block the possibility of these properties actually having a meaningful `null` in the future:

```
class UserControllerTest {  
    private var dao: UserDao? = null  
    private var controller: UserController? = null  
  
    @BeforeEach  
    fun init() {  
        dao = mock()  
        controller = UserController(dao!!)  
    }  
  
    @Test  
    fun test() {  
        controller!!.doSomething()  
    }  
}
```

Nobody can predict how code will evolve in the future; so, if you use a non-null assertion `!!` or an explicit error throw, you should assume that it will throw an error one day. Exceptions are thrown to indicate something unexpected and incorrect (Item 7: *Prefer a null or a sealed result class result when the lack of a result is possible*). However, **explicit errors say much more than generic NPEs and they should nearly always be preferred**.

The rare cases in which the non-null assertion `!!` does make sense are mainly a result of interoperability between our code and libraries in which nullability is not expressed correctly. When you interact with an API that is properly designed for Kotlin, this shouldn't be the norm.

In general, **we should avoid using the non-null assertion `!!`**. This suggestion is rather widely approved by our community; in fact, many teams have a policy to enforce it. Some set the Detekt static analyzer to throw an error whenever it is

used. I think such an approach is too extreme, but I do agree that it is often a code smell. It seems that this operator's appearance is no coincidence. !! seems to be screaming “Be careful” or “There is something wrong here”.

To avoid using !!, we should avoid meaningless nullability. In a case like the one presented above, we should use `lateinit` or `Delegates.notNull`. **lateinit is good practice when we are sure that a property will be initialized before the first use**. We deal with such a situation mainly when classes have a lifecycle, and we set properties in one of the first-invoked methods. For instance, when we set objects in `onCreate` in an Android Activity, `viewDidAppear` in an iOS `UIViewController`, or `componentDidMount` in a React `React.Component`.

```
class UserControllerTest {
    private lateinit var dao: UserDao
    private lateinit var controller: UserController

    @BeforeEach
    fun init() {
        dao = mockk()
        controller = UserController(dao!!)
    }

    @Test
    fun test() {
        controller.doSomething()
    }
}
```

Know that you can always check if a `lateinit` property is initialized by referencing it, and using `isInitialized` property, so in the above example, I could check if `dao` is initialized by using `::dao.isInitialized`.

## Using Elvis operator

For nullability, it is also popular to use the Elvis operator with `throw` or `return` on the right side. Such a structure is highly readable and it gives us more flexibility in deciding what behavior we want to achieve. First of all, we can easily stop a function using `return` instead of throwing an error:

```
fun sendEmail(person: Person, text: String) {  
    val email: String = person.email ?: return  
    //...  
}
```

If we need to take more than one action if a property is incorrectly `null`, we can always add these actions by wrapping `return` or `throw` into the `run` function. Such a capability might be useful if we need to log why a function was stopped:

```
fun sendEmail(person: Person, text: String) {  
    val email: String = person.email ?: run {  
        log("Email not sent, no email address")  
        return  
    }  
    //...  
}
```

The Elvis operator with `return` or `throw` is a popular and idiomatic way to specify what should happen in the case of variable nullability, and we should not hesitate to use it. Again, if possible, keep such checks at the beginning of the function to make them visible and clear.

## error function

In Kotlin stdlib you can find `error` function, that is used to throw an `IllegalStateException`. It is often used to handle a situation that we do not expect to ever take place, like an unexpected value type.

```
// error implementation from Kotlin stdlib  
public inline fun error(message: Any): Nothing =  
    throw IllegalStateException(message.toString())  
  
// example usage  
fun handleMessage(message: Message) = when(message) {  
    is TextMessage -> showText(message.text)  
    is ImageMessage -> showImage(message.image)  
    else -> error("Unknown message type")  
}
```

## Summary

Specify your expectations to:

- Make them more visible.
- Protect your application stability.
- Protect your code correctness.
- Smart cast variables.

The main mechanisms we use for this are:

- `require` block - a universal way to specify expectations for arguments.
- `check` block - a universal way to specify expectations for states.
- – `error` function - a universal way to signal that application reached an unexpected state.
- The Elvis operator with `return` or `throw`.

We prefer avoiding not-null assertion `!!`, however, it is sometimes useful when we are sure that a variable is not null, but the compiler cannot infer it. One feature that helps us avoid `!!` is `lateinit` property initialization.

## Item 6: Prefer standard errors to custom ones

We use exceptions to indicate an unexpected situation. For instance, when you implement a library to parse the JSON format<sup>13</sup>, it is reasonable to throw a `JsonParsingException` when the provided JSON file does not have the correct format:

```
inline fun <reified T> String.readObject(): T {  
    //...  
    if (incorrectSign) {  
        throw JsonParsingException()  
    }  
    //...  
    return result  
}
```

Here we have used a custom error because there is no suitable error in the standard library to indicate this situation. Whenever possible, you should use exceptions from the standard library instead of defining your own. Such exceptions are known by developers and they should be reused. Reusing known elements with well-established contracts makes your API easier to learn and understand. Here is the list of some of the most common exceptions you can use:

- `IllegalArgumentException` - Indicates that an argument passed to a method is invalid. We typically throw it using `require` or `requireNotNull`, as described in the previous item.
- `IllegalStateException` - Indicates that the state of our program is invalid. For instance, we might throw it when we try to use a variable that has not been initialized yet. We typically throw it using `check`, `checkNotNull` or `error`, as described in the previous item.
- `UnsupportedOperationException` - Indicates that the declared method is not supported by the object. Such a situation should be avoided; when a method is not supported, it should not be present in the class<sup>14</sup>. We typically throw it using `TODO` function from Kotlin stdlib, that is added by default by IntelliJ to auto-generated code.
- `IndexOutOfBoundsException` - Indicates that the index parameter value is out of range. Used especially by collections and arrays. It is thrown, for instance, by `ArrayList.get(Int)`.

---

<sup>13</sup>I am not suggesting implementing it by yourself when there is no good reason to. There are already great libraries for this that have been well tested, documented and optimized.

<sup>14</sup>One rule that is violated in such a case is the Interface Segregation Principle, which states that no client should be forced to depend on methods it does not use.

- `ConcurrentModificationException` - Indicates that concurrent modification is prohibited but has been detected.
- `NoSuchElementException` - Indicates that the element being requested does not exist. Used, for instance, when we implement `Iterable` and the client asks for `next` when there are no more elements.

## Item 7: Prefer a nullable or Result result type when the lack of a result is possible

Sometimes a function cannot produce its desired result. A few common examples are:

- We try to get data from a server, but there is a problem with the internet connection.
- We try to get the first element that matches some criteria, but there is no such element in our list.
- We try to parse an object from text, but this text is malformatted.

There are two main mechanisms to handle such situations:

- Return a `null` or `Result.failure`, thus indicating failure.
- Throw an exception.

There is an important difference between these two. Exceptions should not be used as a standard way to pass information. **All exceptions indicate incorrect, special situations and should be treated this way. We should use exceptions only for exceptional conditions** (*Effective Java* by Joshua Bloch). The main reasons for this are:

- The way exceptions propagate is not very readable for most programmers and might easily be missed in code.
- In Kotlin, all exceptions are unchecked. Users are not forced or even encouraged to handle them. They are often not well documented, and they are not very visible when we use an API.
- Because exceptions are designed for exceptional circumstances, there is little incentive for JVM implementers to make them as fast as explicit tests.
- Placing code inside a try-catch block inhibits certain optimizations that the compiler might otherwise perform.

It is worth mentioning that exceptions are used by some popular patterns, like on backend, exceptions are used to end request processing and respond to requester with a specific response code and message. Similarly on Android, exceptions are sometimes used to end a process and display a concrete dialog or toast to the user. In such cases, many of my arguments against exceptions do not apply, and using exceptions could be reasonable.

On the other hand, `null` or `Result.failure` are both perfect for indicating an expected error. They are explicit, efficient, and can be handled in idiomatic ways. This is why the rule is that **we should prefer to return `null` or `Result.failure` when an error is expected, and we should throw an exception when an error is not expected.** Here are some examples:

```
inline fun <reified T> String.readObjectOrNull(): T? {  
    //...  
    if (incorrectSign) {  
        return null  
    }  
    //...  
    return result  
}  
  
inline fun <reified T> String.readObject(): Result<T> {  
    //...  
    if (incorrectSign) {  
        return Result.failure(JsonParseException())  
    }  
    //...  
    return Result.success(result)  
}  
  
class JsonParseException : Exception()
```

## Using `Result` result type

We use the `Result` class from the Kotlin stdlib to return a result that can be either a success or a failure. Failure includes an exception, that keeps the information about the error. **We use `Result` instead of nullable type in functions that need to pass additional information in the case of failure.** For example, when we implement a function that is fetching information from the internet, `Result` should be preferred over `null`, because we can pass the information about the error, like the error code or the error message.

When we choose to return `Result`, the user of this function will be able to handle it using methods from the `Result` class:

```
userText.readObject<Person>()  
.onSuccess { showPersonAge(it) }  
.onFailure { showError(it) }
```

Using such error handling is simpler than using a try-catch block. It is also safer, because an exception can be missed and can stop our whole application; in contrast, a `null` value or a `Result` object needs to be explicitly handled, and it won't interrupt the flow of the application.

The difference between a nullable result and the `Result` object is that we should prefer the latter when we need to pass additional information in the case of failure; otherwise, we should prefer `null`.

The `Result` class has a rich API of methods you can use to handle your result, including:

- `isSuccess` and `isFailure` properties, which we use to check if the result represents a success or a failure (`isSuccess == !isFailure` is always true).
- `onSuccess` and `onFailure` methods, which call their lambda expressions when the result is, respectively, a success or a failure.
- `getOrNull` method, which returns the value if the result is a success, or `null` otherwise.
- `getOrThrow` method, which returns the value if the result is a success, or throws the exception from the failure otherwise.
- `getOrDefault` method, which returns the value if the result is a success, or the default value provided as an argument if the result is a failure.
- `getOrElse` method, which returns the value if the result is a success, or calls its functional argument and returns its result.
- `exceptionOrNull` method, which returns the exception if the result is a failure, or `null` otherwise.
- `map` method for transforming the success value.
- `recover` method for transforming a throwable value into a success value.
- `fold` method for handling both success and failure in a single method.

To transform a function that throws exceptions into one that returns `Result`, use `runCatching`.

```
fun getA(): Result<T> = runCatching { getAThrowing() }
```

## Using `null` result type

In Kotlin, `null` is a marker of a lack of value. When a function returns `null`, it means that it cannot return a value. For example:

- `List<T>.getOrNull(Int)` returns `null` when there is no value at the given index.
- `String.toIntOrNull()` returns `null` when `String` cannot be correctly parsed to `Int`.

- `Iterable<T>.firstOrNull((() -> Boolean))` returns `null` when there are no elements matching the predicate from the argument.

As you can see, `null` is used to indicate that a function cannot return the expected value. **We use nullable type instead of Result in functions that do not need to pass additional information in the case of failure, where the meaning of null is clear.** In the function `String.toIntOrNull()`, it is clear that `null` means that the string cannot be parsed to `Int`. In the function `Iterable<T>.firstOrNull(() -> Boolean)`, it is clear that `null` means that there are no elements matching the predicate. **For all functions that return null, the meaning of null should be clear.**

Nullable value needs to be unwrapped before it can be used. For dealing with them, Kotlin offers us many useful features, like the safe call operator `?.`, the Elvis operator `?:`, and smart casting.

```
val age = userText.readObjectOrNull<Person>()?.age ?: -1

val printer: Printer? = getFirstAvailablePrinter()
printer?.print() // Safe call
if (printer != null) printer.print() // Smart casting
```

## Null is our friend, not an enemy

Many Kotlin developers are ex-Java developers, who are thought to treat `null` like an enemy. For example, in Effective Java (2nd edition), Joshua Bloch presents **Item 43: Return empty arrays or collections, not nulls.** Such a suggestion would be absurd in Kotlin. An empty collection has a completely different meaning than `null`. Imagine we called the function `getUsers`: if it returned `null`, this would mean it couldn't produce a value, so we still don't know what the answer is; in contrast, if it returned an empty collection, this would mean that there are no users. These are different results, and they should not be confused. Kotlin's type system lets us express what is nullable and what is not, and it forces us to handle nulls consciously. We should not be afraid of nulls: we should embrace them and use them to express our intentions. Forget about all the suggestions to avoid nulls because they are not applicable in Kotlin. In Kotlin, `null` is our friend, not an enemy<sup>15</sup>.

## Defensive and offensive programming

In **Item 5: Specify your expectations for arguments and state**, I explained that we should throw exceptions to signal incorrect arguments or states, and in this

---

<sup>15</sup>See *Null is your friend, not a mistake* by Roman Elizarov. Link: <https://kt.academy/l/re-null>

item, I explained that we should in general avoid throwing exceptions and prefer returning `Result` or nullable types instead. These two statements seem to be in conflict, but they are not, because they refer to different kinds of situations.

Exceptions should not be part of our regular program execution flow; so, when you perform an operation that might either succeed or fail, like fetching data from a database or network, you should use `Result` or nullable types. This forces the developer to handle the failure case explicitly. Since it is part of the regular program execution flow, it is best to handle such a situation safely so all possible situations are handled correctly. This is an implementation of the *defensive programming* idea.

On the other hand, when a developer makes a mistake, like calling a method with incorrect arguments or calling a method on an object in an incorrect state, silencing such a situation would be dangerous because our program has encountered a situation that is clearly unexpected. We should loudly signal this situation so our program can be corrected. This is an implementation of the *offensive programming* idea.

Defensive and offensive programming do not contradict each other; they are more like yin and yang: different techniques that are both needed in our programs for the sake of safety, so we need to understand and use both of them appropriately.

## Summary

- When a function can fail, we should return `Result` or a nullable type instead of throwing an exception.
- We should use `Result` when we need to pass additional information in the case of failure.
- We should use a nullable type when the meaning of `null` is clear.
- We should not be afraid of nulls: we should embrace them and use them to express our intentions.
- We should use defensive programming to handle regular program execution flow, while offensive programming should be used to handle unexpected situations.

## Item 8: Close resources with use

There are resources that cannot be closed automatically, so we need to invoke the `close` method when we don't need them anymore. The Java standard library that we use in Kotlin/JVM contains a lot of these resources, such as:

- `InputStream` and `OutputStream`,
- `java.sql.Connection`,
- `java.io.Reader` (`FileReader`, `BufferedReader`, `CSSParser`),
- `java.net.Socket` and `java.util.Scanner`.

All these resources implement the `Closeable` interface, which extends `AutoCloseable`.

The problem is that in all these cases we need to be sure that we invoke the `close` method when we no longer need a resource because these resources are rather expensive and they aren't easily closed by themselves (the Garbage Collector will eventually handle it if we don't keep a reference to a resource, but this will take some time). Therefore, to be sure that they have been closed, we traditionally wrap such resources in a `try-finally` block and call `close` there:

```
fun countCharactersInFile(path: String): Int {  
    val reader = BufferedReader(FileReader(path))  
    try {  
        return reader.lineSequence().sumBy { it.length }  
    } finally {  
        reader.close()  
    }  
}
```

Such a structure is complicated and incorrect. It is incorrect because `close` can throw an error that will not be caught. Also, if we have errors from both the body of the `try` and from the `finally` blocks, only one will be properly propagated. The behavior we should expect is that the information about the new error is added to the previous error. The proper implementation of this is long and complicated, but it is so common that it has been extracted into the `use` function from the standard library. The `use` method should be used to properly close resources and handle exceptions. This function can be used on any `Closeable` object:

```
fun countCharactersInFile(path: String): Int {  
    val reader = BufferedReader(FileReader(path))  
    reader.use {  
        return reader.lineSequence().sumBy { it.length }  
    }  
}
```

The receiver (reader in this case) is also passed as an argument to the lambda, so the syntax can be shortened:

```
fun countCharactersInFile(path: String): Int {  
    BufferedReader(FileReader(path)).use { reader ->  
        return reader.lineSequence().sumBy { it.length }  
    }  
}
```

As this support is often needed for files and it is common to read files line by line, there is also a `useLines` function in the Kotlin Standard Library that gives us a sequence of lines (`String`) and closes the underlying reader once the processing is complete:

```
fun countCharactersInFile(path: String): Int {  
    File(path).useLines { lines ->  
        return lines.sumBy { it.length }  
    }  
}
```

This is the proper way to process even large files because this sequence will read lines on demand and does not hold more than one line at a time in memory. The cost is that this sequence can be used only once. If you need to iterate over the lines of a file more than once, you need to open it more than once. The `useLines` function can be also used as an expression:

```
fun countCharactersInFile(path: String): Int =  
    File(path).useLines { lines ->  
        lines.sumBy { it.length }  
    }
```

All the above implementations use sequences to operate on the file, and this is the correct way to do it. Thanks to this, we can always read only one line instead of loading the content of the whole file. More about this in Item 54: *Prefer Sequences for big collections with more than one processing step*.

## Summary

Operate on objects that implement `Closeable` or `AutoCloseable` using `use`. This is a safe and easy option. When you need to operate on a file, consider `useLines`, which produces a sequence to iterate over the next lines.

## Item 9: Write unit tests

Not Kotlin-specific

Basics

In this chapter, you've seen quite a few ways to make your code safer, but the ultimate way to achieve this is to have different kinds of tests. One kind of testing involves checking that our application behaves correctly from the user's perspective. These kinds of tests are too often the only ones recognized by management because their primary goal is generally to make the application behave correctly from outside, not internally. These kinds of tests (known as acceptance tests) do not even need developers at all. They can be handled by a sufficient number of testers or - as is generally better in the long run - by automatic tests written by test engineers. Such tests are useful for programmers, but they are not sufficient. They do not build proper assurance that concrete elements of our system behave correctly. They also do not provide fast feedback that is useful during development. For that, we need a different kind of test that is much more useful for developers and that is written by developers: unit tests.

Here is an example unit test that checks if our `fib` function, which calculates the Fibonacci number at the `n`-th position, gives us correct results for the first 5 numbers:

```
@Test
fun `fib works correctly for the first 5 positions`() {
    assertEquals(1, fib(0))
    assertEquals(1, fib(1))
    assertEquals(2, fib(2))
    assertEquals(3, fib(3))
    assertEquals(5, fib(4))
}
```

With unit tests, we typically check:

- Common use cases (the happy path) - typical ways we expect the element to be used. Just like in the example above, we test if the function works for a few small numbers.
- Common error cases or potential problems - cases that we suppose might not work correctly or that have been shown to be problematic.
- Edge cases and illegal arguments - for `Int`, we might check for really big numbers, like `Int.MAX_VALUE`. For a nullable type, we might check for `null` or an object filled with `null` values. There are no Fibonacci numbers for negative positions, so we might check how this function behaves with them.

Unit tests can be really useful during development as they give fast feedback on how an implemented element works. Tests are ever accumulating, so you can easily check for regression. They can also check cases that are hard to test manually. There is even an approach called Test-Driven Development (TDD), in which we first write a unit test and then an implementation to satisfy it<sup>16</sup>.

The biggest advantages of unit tests are:

- **Well-tested elements tend to be more reliable.** There is also a psychological aspect: when elements are well tested, we operate more confidently with them.
- **When an element is properly tested, we are not afraid to refactor it.** As a result, well-tested programs tend to get better and better. On the other hand, in programs that are not tested, developers are scared of touching legacy code because they might accidentally introduce an error without even knowing about it.
- **It is often much faster to check if something works correctly using unit tests rather than checking it manually.** A faster feedback loop makes development faster and more pleasurable<sup>17</sup>. It also helps reduce the cost of fixing bugs: the quicker you find them, the cheaper it is to fix them.

Clearly, there are also disadvantages to unit tests:

- It takes time to write unit tests. However, **in the long term, good unit tests probably save time as we spend less time debugging and looking for bugs later.** We also save a lot of time as running unit tests is much faster than manual testing or other kinds of automated tests.
- We need to adjust our code to make it testable. Such changes are often hard, but they generally also force developers to use good and well-established architectures.
- It is hard to write good unit tests because they require skills and understanding that are completely different than developers' usual skills. Poorly written unit tests can do more harm than good. **Everyone needs to learn how to properly unit test their code.** It is useful to take a course on Software Testing or Test-Driven Development (TDD) first.

The biggest challenge is to obtain the skills to effectively unit test and to write code that supports unit testing. **Experienced Kotlin developers should obtain such skills and at least learn to unit test the important parts of code.**

When you consider what to unit test, you should focus on:

---

<sup>16</sup>Formally, there are 3 steps in TDD: Red - write a unit test; Green - write only enough production code to make the failing unit test pass; Refactor - Refactor the code to clean it. Then we repeat these steps.

<sup>17</sup>I believe that no one likes waiting for a project to build and start.

- Complex functionalities
- Parts that will probably change over time or will be refactored
- Business logic
- Parts of public APIs
- Parts that have a tendency to break
- Production bugs that we have fixed

We do not need to stop there. Tests are an investment in application reliability and long-term maintainability.

## Summary

This chapter started with the idea that the first priority should be for our programs to behave correctly. This can be supported by using the good practices presented in this chapter, but the best way to ensure that our application behaves correctly is to check it by testing, especially unit testing. This is why a responsible chapter about safety needs at least a short section about unit testing, just as a responsible business application requires at least some unit tests.

# Chapter 2: Readability

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

—Martin Fowler, *Refactoring: Improving the Design of Existing Code*, p.

15

There is a very common misconception that Kotlin is designed to be concise. It isn't. There are languages that are much more concise. For instance, the most concise language I know is APL. This is John Conway's "Game of Life" implemented in APL:

```
life←{↑1 ⍺V.∧3 4=+/,¯1 0 1∘.Θ¯1 0 1∘.①c⍵}
```

Your first thought is probably "Wow, that's short". Then you might realize that you don't have some of these characters on your keyboard. There are other such languages; for example, here is the same program in J:

```
life=:[:+/((3 4=/:+/(,/, "0/~i:1)|.)*)*.1,:]
```

These two are very concise languages. This characteristic makes them champions in code golf contests. It also makes them absurdly hard to read. Let's be honest: even for experienced APL developers (and there are probably only a few of them in the world), it is a challenge to understand what this program does and how it works.

Kotlin never had ambitions to be very concise. It is designed to be **readable**. It is concise compared to other popular languages, but this is because Kotlin eliminates a lot of noise: boilerplate code and repetitive structures. This was done to help developers concentrate on what is important and thus make Kotlin more readable.

Kotlin allows programmers to design clean and meaningful code and APIs. Its features let us hide or highlight whatever we want. This chapter is about using these tools wisely; it serves as an introduction and provides a set of general suggestions. However, it also introduces the concept of readability, which we will refer to in the rest of this book, especially in *Part 2: Abstraction design*, where we will dive into topics related to class and function design.

Let's introduce the general problem with a slightly more abstract item about readability.

## Item 10: Design for readability

It is a known observation in programming that developers read code much more than they write it. A common estimate is that for every minute spent writing code, ten minutes are spent reading it<sup>18</sup>. If this seems unbelievable, just think about how much time you spend reading code when you are trying to find an error. I believe that everyone has been in this situation at least once in their career when they've been searching for an error for days or weeks, just to fix it by changing a single line. When we learn how to use a new API, it's often from reading code. We usually read code to understand the logic or how the implementation works. **Programming is mostly about reading, not writing.** Knowing this, it should be clear that we should code with readability in mind.

### Reducing cognitive load

Readability means something different to everyone. However, some rules are formed on the basis of experience or have come from cognitive science. Just compare the following two implementations:

```
// Implementation A
if (person != null && person.isAdult) {
    view.showPerson(person)
} else {
    view.showError()
}

// Implementation B
person?.takeIf { it.isAdult }?
    .let(view::showPerson)
    ?: view.showError()
```

Which one is better, A or B? Using the naive reasoning that the one with fewer lines is better is not a good answer. We could remove the line breaks from the first implementation, but this wouldn't make it more readable. Counting the number of characters would not be very useful either. Especially since the difference is not big. The first implementation has 79 characters, and the second has 68. The second implementation is only a bit shorter, but it is much less readable.

How readable both constructs are, depends on how fast we can understand each of them. This, in turn, depends greatly on how much our brain is trained to understand each idiom (structure, function, pattern). For a beginner in Kotlin,

---

<sup>18</sup>This ratio was popularized by Robert C. Martin in the book *Clean Code*.

surely implementation A is way more readable. It uses general idioms (`if/else`, `&&`, method calls). Implementation B has idioms that are typical of Kotlin (`safe call ?..`, `takeIf`, `let`, Elvis operator `?:`, a bounded function reference `view::showPerson`). Surely, all these idioms are commonly used throughout Kotlin, so they are well known by most experienced Kotlin developers. Still, it is hard to compare them. Kotlin isn't most developers' first language, and we have much more experience in general programming than in Kotlin programming. We don't write code only for experienced developers. The chances are that the junior you hired (after fruitless months of searching for a senior) does not know what `let`, `takeIf`, and bounded references are. It is also very likely that he has never seen the Elvis operator used this way. That person might spend a whole day puzzling over this single block of code. Additionally, even for experienced Kotlin developers, Kotlin is not the only programming language they use. Many developers reading your code might be experienced with Kotlin, but certainly they will have more general-programming experience. The brain will always need to spend more time recognizing Kotlin-specific idioms, than general-programming idioms. Even after years with Kotlin, it still takes much less time for me to understand implementation A. Every less-known idiom introduces a bit of complexity. When we analyze them all together in a single statement that we need to comprehend nearly all at once, this complexity grows quickly.

Notice that implementation A is easier to modify. Let's say we need to add an operation to the `if` block. This is easy in implementation A; however, in implementation B, we can no longer use function references. Adding something to the "else" block in implementation B is even harder because we need to use some function to be able to hold more than a single expression on the right side of the Elvis operator:

```
if (person != null && person.isAdult) {  
    view.showPerson(person)  
    view.hideProgressWithSuccess()  
} else {  
    view.showError()  
    view.hideProgress()  
}
```

```
person?.takeIf { it.isAdult }  
    ?.let {  
        view.showPerson(it)  
        view.hideProgressWithSuccess()  
    } ?: run {  
        view.showError()  
        view.hideProgress()  
    }  
}
```

Debugging implementation A is also much simpler. This should be no surprise, as debugging tools were made for such basic structures.

The general rule is that less-common “creative” structures are generally less flexible and not so well supported. Let’s say, for instance, that we need to add a third branch to show a different error when the variable `person` is `null`, and a different one when `person` is not an adult. In implementation A, which uses `if/else`, we can easily change `if/else` to `when` using IntelliJ refactoring and then easily add an additional branch. The same change to the code would be painful in implementation B. It would probably need to be totally rewritten.

Have you noticed that implementations A and B do not work the same way? Can you spot the difference? Go back and think about it now.

The difference lies in the fact that `let` returns a result from the lambda expression. This means that if `showPerson` returns `null`, then the second implementation will call `showError` as well! This is definitely not obvious; it teaches us that when we use less-familiar structures, it is easier to fall victim to unexpected code behavior.

The general rule here is that we want to reduce cognitive load. Our brains recognize patterns, on the basis of which they build an understanding of how programs work. When we think about readability, we want to shorten this distance. We prefer less code but also more common structures. We recognize familiar patterns when we see them often enough. We always prefer structures that we are familiar with in other disciplines.

## Do not get extreme

Just because I showed how `let` can be misused in the previous example, this does not mean it should always be avoided. It is a popular idiom that is reasonably used to make code better in various contexts. One common example is when we have a nullable mutable property, and we need to do an operation only if it is not `null`. Smart casting cannot be used because a mutable property can be modified by another thread, but a great way to deal with this is to use the safe call `let`:

```
class Person(val name: String)
var person: Person? = null

fun printName() {
    person?.let {
        print(it.name)
    }
}
```

Such an idiom is popular and widely recognizable. There are many more reasonable cases for `let`, for instance:

- To move an operation after its argument calculation
- To use it to wrap an object with a decorator

Here are examples of these two uses (both also use function references):

```
students
    .filter { it.result >= 50 }
    .joinToString(separator = "\n") {
        "${it.name} ${it.surname}, ${it.result}"
    }
    .let(::print)

var obj = FileInputStream("/file.gz")
    .let(::BufferedInputStream)
    .let(::ZipInputStream)
    .let(::ObjectInputStream)
    .readObject() as SomeObject
```

In both these cases, we pay the price: this code is harder to debug and is harder for less-experienced Kotlin developers to understand. But nothing comes for free, and this seems like a fair deal. The problem is when we introduce a lot of complexity for no good reason.

There will always be discussions about when something does or does not make sense. Balancing it is an art. It is good, though, to recognize how different structures introduce complexity or how they clarify things, especially when they are used together. The complexity of two structures used together is generally much more than the sum of their individual complexities.

## Conventions

We've acknowledged that different people have different views of what readability means. We constantly fight over function names, discuss what should be explicit or implicit, what idioms we should use, and much more. Programming is an art of expressiveness. Still, there are some conventions that need to be understood and remembered.

When one of my workshop groups in San Francisco asked me about the worst thing one can do in Kotlin, I gave them this:

```
val abc = "A" { "B" } and "C"  
print(abc) // ABC
```

All we need to make this terrible syntax possible is the following code:

```
operator fun String.invoke(f: ()->String): String =  
    this + f()  
  
infix fun String.and(s: String) = this + s
```

This code violates many rules that we will describe later:

- It violates operator meaning - `invoke` should not be used this way, because `String` cannot be invoked.
- The usage of the ‘lambda as the last argument’ convention here is confusing. It is fine to use it after functions, but we should be very careful when we use it for the `invoke` operator.
- `and` is clearly a bad name for this infix method. `append` or `plus` would be much better.
- We already have language features for `String` concatenation, and we should use them instead of reinventing the wheel.

Behind each of these suggestions, there is a more general rule that ensures a good Kotlin style. We will cover the most important ones in this chapter, starting with the first item, which will focus on overriding operators.

## Item 11: An operator's meaning should be consistent with its function name

Operator overloading is a powerful feature and, like most powerful features, it is dangerous as well. In programming, with great power comes great responsibility. As a trainer, I've often seen how people can get carried away when they first discover operator overloading. For example, one exercise involves making a function for calculating the factorial of a number:

```
fun Int.factorial(): Int = (1..this).product()  
  
fun Iterable<Int>.product(): Int =  
    fold(1) { acc, i -> acc * i }
```

As this function is defined as an extension function to `Int`, its usage is convenient:

```
print(10 * 6.factorial()) // 7200
```

All mathematicians know that there is a special notation for factorials. It is an exclamation mark after a number:

```
10 * 6!
```

There is no support in Kotlin for such an operator; however, as one of my workshop participants noticed, we can use operator overloading for `not` instead:

```
operator fun Int.not() = factorial()  
  
print(10 * !6) // 7200
```

We can do this, but should we? The simplest answer is NO. You only need to read the function declaration to notice that the name of this function is `not`. As this name suggests, it should **not** be used this way. It represents a logical operation, not a numeric factorial. This usage would be confusing and misleading. In Kotlin, all operators are just syntactic sugar for functions with concrete names, as presented in the table below. Every operator can always be invoked as a function instead of using the operator syntax. What would the following look like?

```
print(10 * 6.not()) // 7200
```

| Operator       | Corresponding Function |
|----------------|------------------------|
| <b>+a</b>      | a.unaryPlus()          |
| <b>-a</b>      | a.unaryMinus()         |
| <b>!a</b>      | a.not()                |
| <b>++a</b>     | a.inc()                |
| <b>--a</b>     | a.dec()                |
| <b>a+b</b>     | a.plus(b)              |
| <b>a-b</b>     | a.minus(b)             |
| <b>a*b</b>     | a.times(b)             |
| <b>a/b</b>     | a.div(b)               |
| <b>a..b</b>    | a.rangeTo(b)           |
| <b>a in b</b>  | b.contains(a)          |
| <b>a+=b</b>    | a.plusAssign(b)        |
| <b>a-=b</b>    | a.minusAssign(b)       |
| <b>a*=b</b>    | a.timesAssign(b)       |
| <b>a/=b</b>    | a.divAssign(b)         |
| <b>a==b</b>    | a.equals(b)            |
| <b>a&gt;b</b>  | a.compareTo(b) > 0     |
| <b>a&lt;b</b>  | a.compareTo(b) < 0     |
| <b>a&gt;=b</b> | a.compareTo(b) >= 0    |
| <b>a&lt;=b</b> | a.compareTo(b) <= 0    |

What each operator translates to in Kotlin.

The meaning of each operator in Kotlin always stays the same. This is a very important design decision. Some languages, like Scala, give you unlimited operator overloading capabilities. This amount of freedom is known to be highly misused by some developers. Reading code using an unfamiliar library for the first time might be difficult, even if it has meaningful names of functions and classes. Now imagine operators being used with another meaning that is known only to developers familiar with category theory. It would be way harder to understand. You would need to understand each operator separately, remember what it means in the specific context, and then keep all this in mind to connect the pieces in order to understand the whole statement. We don't have such a problem in Kotlin because each operator has a concrete meaning. For instance, when you see the following expression:

```
x + y == z
```

You know that this is the same as:

```
x.plus(y).equal(z)
```

Or it can be the following code if `plus` declares a nullable return type:

```
(x.plus(y))?.equal(z) ?: (z === null)
```

These functions have concrete names, and we expect all functions to do what their names indicate. This greatly restricts what each operator can be used for. Using `not` to return `factorial` is a clear breach of this convention and should never happen.

It is good to mention, that this rule was even breached by Kotlin stdlib, as it defines `div` extension on `Path` to allow the following use:

```
val path = Path("A")
val path2 = path / "B"
println(path2) // Prints: A/B
```

It is not so beautiful when explicit name is used. Such extension make Kotlin more "magical", and I suggest to avoid using them if we want to keep our code readable.

```
val path = Path("A")
val path2 = path.div("B")
println(path2) // Prints: A/B
```

## Unclear cases

The biggest problem is when it is unclear if some usage fulfills the conventions. For instance, what does it mean when we triple a function? For some people, it is clear that this means making another function that repeats this function 3 times:

```
operator fun Int.times(operation: () -> Unit): ()->Unit =
    { repeat(this) { operation() } }

val tripledHello = 3 * { print("Hello") }

tripledHello() // Prints: HelloHelloHello
```

For others, it might be clear that it means that we want to call this function 3 times<sup>19</sup>:

```
operator fun Int.times(operation: () -> Unit) {
    repeat(this) { operation() }
}

3 * { print("Hello") } // Prints: HelloHelloHello
```

When the meaning is unclear, it is better to favor descriptive extension functions. If we want to keep their usage operator-like, we can make them infix:

```
infix fun Int.timesRepeated(operation: () -> Unit) = {
    repeat(this) { operation() }
}

val tripledHello = 3 timesRepeated { print("Hello") }
tripledHello() // Prints: HelloHelloHello
```

Sometimes it is better to use a top-level function instead. Repeating a function 3 times is already implemented and distributed in the stdlib:

```
repeat(3) { print("Hello") } // Prints: HelloHelloHello
```

## When is it fine to break this rule?

There is one very important case in which it is fine to use operator overloading in a strange way: when we design a Domain Specific Language (DSL). Think of a classic HTML DSL example:

---

<sup>19</sup>The difference is that the first one produces a function while the other one calls a function. In the first case, the result of the multiplication is  $()\rightarrow\text{Unit}$ , but in the second case it is **Unit**.

```
body {  
    div {  
        +"Some text"  
    }  
}
```

You can see that to add text to an element, we use `String.unaryPlus`. This is acceptable because it is clearly part of the Domain Specific Language (DSL). In this specific context, it's not surprising to readers that different rules apply.

## Summary

Use operator overloading conscientiously. A function's name should always be coherent with its behavior. Avoid cases where operator meaning is unclear. Clarify it by using a regular function with a descriptive name instead. If you wish to have a more operator-like syntax, then use the `infix` modifier or a top-level function.

## Item 12: Use operators to increase readability

In the previous item, I warned about the misuse of operator overloading. In this chapter, I would like to show the usefulness of operators for improving readability.

Let's start with a clear example. With operators, we can operate on `BigDecimal` and `BigInteger` similarly to regular numbers.

```
val netPrice = BigDecimal("10")
val tax = BigDecimal("0.23")
val currentBalance = BigDecimal("20")
val newBalance = currentBalance - netPrice * tax
println(newBalance) // 17.70
```

We can also add duration to time.

```
val now = ZonedDateTime.now()
val duration = Duration.ofDays(1)
val sameTimeTomorrow = now + duration
```

You can compare them by using explicit methods:

```
val newBalance = currentBalance.minus(netPrice.times(tax))
val sameTimeTomorrow = now.plus(duration)
```

I hope that the value of using operators here is clear.

All classes that are `Comparable` can also be compared with comparison operators (`>`, `<`, `>=` and `<=`) or with a range check (`value in min..max`). This includes big numbers (`BigDecimal`, `BigInteger`) and objects used to represent time and duration (`Instant`, `ZonedDateTime`, `LocalDate`, `Duration`, etc.). This is important because we often operate on these types, and we often need to compare them (I hope it is clear that if you represent money, you should use `BigDecimal` instead of `Double`, which might round some numbers and lose precision).

```

val now = LocalDateTime.now()
val start = LocalDate.parse("2021-10-17").atStartOfDay()
val end = LocalDate.parse("2021-10-21").atStartOfDay()
if(now > start) { /*...*/ }
if(now < end) { /*...*/ }
if(now in start..end) { /*...*/ }

val price = BigDecimal("100.00")
val minPrice = BigDecimal("10.00")
val maxPrice = BigDecimal("1000.00")
if(price > minPrice) { /*...*/ }
if(price < maxPrice) { /*...*/ }
if(price in minPrice..maxPrice) { /*...*/ }

```

The above code is an alternative to using the following methods:

```

if(now.isAfter(start)) { /*...*/ }
if(now.isBefore(end)) { /*...*/ }
if(!now.isBefore(start) && !now.isAfter(end)) { /*...*/ }

if(price.compareTo(minPrice) > 0) { /*...*/ }
if(price.compareTo(maxPrice) < 0) { /*...*/ }
if(minPrice.compareTo(price) <= 0 &&
    price.compareTo(maxPrice) <= 0) { /*...*/ }

```

Although `isAfter` and `isBefore` might be more readable than comparison operators, I hope it seems clear that operators are clearly easier to understand in other cases.

It is worth noticing that there is an inconsistency between the `BigDecimal` functions `equals` and `compareTo`. The `equals` function checks the number of decimal places, so `BigDecimal("1.0")` is not equal to `BigDecimal("1.00")`. This is something you should consider when you compare two numbers. This is one of the reasons why we tend to use `BigDecimal` numbers with the same precision across a whole project. The `compareTo` function does not look at the precision, so it is possible that `A >= B`, `A <= B`, but `A != B` (so the contract of `compareTo` is violated, as explained in Item 44: Respect the contract of `compareTo`).

```

val num1 = BigDecimal("1.0")
val num2 = BigDecimal("1.00")
println(num1 == num2) // false
println(num1 >= num2 && num1 <= num2) // true

```

The last typical case in which I introduce an operator is when we need to check if an element is in a collection or a set. The classic way to do this is using `contains`, but we could also use the `in` operator.

```
val SUPPORTED_TAGS = setOf("ADMIN", "TRAINER", "ATTENDEE")
val tag = "ATTENDEE"

println(SUPPORTED_TAGS.contains(tag)) // true
// or
println(tag in SUPPORTED_TAGS) // true
```

Compare the above approaches and consider which is more readable. For me and the colleagues I discussed this matter with, using `in` does not always increase readability. It all depends on what the active element is. Here, `tag` is active, and putting it up-front makes this code easier to read, just as “There’s a soda in the fridge” is more intuitive than “The fridge contains a soda”. Now consider a case in which the collection is more important. For instance, “A human has a liver” seems more intuitive than “A liver is in a human”. The same with the following code:

```
val ADMIN_TAG = "ADMIN"
val admins = users.map { user.tags.contains(ADMIN_TAG) }
// or
val admins = users.map { ADMIN_TAG in user.tags }
```

For me, using `contains` in this case makes the code clear. I understand that some might feel differently about this, so please do not treat this as a hard rule (do not force it on reviews). Writing really readable code is great art, so all the rules should be treated as suggestions.

Operators can also be added to our own classes, like units of measure, money wrappers, other kinds of numbers, and others.

```
@JvmInline
value class Centimeter(private val value: Double) {
    operator fun plus(other: Centimeter): Centimeter =
        Centimeter(value + other.value)

    operator fun plus(other: Millimeter): Centimeter =
        Centimeter(value + other.value * 10)

    // ...
}
```

These are the most common cases where I introduce Kotlin operators, but there are clearly many more overloaded operators in the standard library.

## Item 13: Consider making types explicit

Kotlin has a great type inference system, which allows us to omit types when they are obvious to developers:

```
val num = 10
val name = "Marcin"
val ids = listOf(12, 112, 554, 997)
```

This improves not only development time but also readability when a type is clear from the context and additional specification is redundant and messy. However, this should not be overused in cases when a type is not clear:

```
val data = getSomeData()
```

We design our code for readability, and we should not hide important information that might be important to a reader. It is not valid to argue that return types can be omitted because a reader can always jump into the functional specification to check them there. Types might be inferred there as well. Therefore, a developer can end up going deeper and deeper. Also, a developer might read this code on GitHub or another environment that does not support jumping into implementations. Even if this is possible, we all have a very limited working memory, and wasting it like this is not a good idea. Type is important information, and it should be specified if it is not clear.

```
val data: UserData = getSomeData()
```

We specify explicit types to improve not only readability but also safety, as shown in Item 3: Eliminate platform types as soon as possible. **Type might be important information for both a developer and the compiler. Whenever it is, do not hesitate to specify it. It costs little and can help a lot.** This rule is especially important for public APIs<sup>20</sup>, where explicit types are often useful for API developers. **When we design a public API, we should always explicitly specify exposed types.** This is important for readability but also for safety. Let me show you an example.

Let's start from saying, that the inferred type is always the most specific type that is possible. That is why, in the below example, the type of `animal` is `Bear`, and `Camel` cannot be assigned to it:

---

<sup>20</sup>Elements (classes, functions, objects) that might be used from some outer module or some part of our code that is maintained by different developers. For instance, in libraries these are all public and protected classes, functions and object declarations.

```
open class Animal
class Bear : Animal()
class Camel : Animal()

fun main() {
    var animal = Bear()
    animal = Camel() // Error: Type mismatch
}
```

Now consider, that you implement a library, that provides an API for creating cars in your country. You specified the following interface, which is used by your clients to define factories and create cars:

```
interface CarFactory {
    fun produce(): Car
}
```

However, you've noticed that there is one particularly popular car manufactured in your country, so you decided to make it the default car:

```
val DEFAULT_CAR: Car = Fiat126P()
```

Since this car is produced in most of these factories, you made it the default. You inferred the type because you decided that `DEFAULT_CAR` is a `Car` anyway:

```
interface CarFactory {
    fun produce() = DEFAULT_CAR
}
```

Similarly, someone later looked at `DEFAULT_CAR` and decided that its type could be inferred:

```
val DEFAULT_CAR = Fiat126P()
```

Now, all your factories can only produce `Fiat126P`. Not good. If you had defined this interface for your own project, this problem would probably have been caught soon and easily fixed. However, if this interface is part of an external API, you might be informed first by angry users.

## Explicit API mode for library authors

Kotlin 1.4 introduced explicit API mode for library authors. When it is enabled, Kotlin forces us to specify types and visibility modifiers for all declarations that are part of our public API. This is a great feature that helps us to avoid mistakes like the one above. It is also a great way to improve the readability of our code. We can enable it in our `build.gradle(.kts)` file:

```
kotlin {  
    // ...  
  
    // for strict mode  
    explicitApi()  
  
    // for warning mode  
    explicitApiWarning()  
}
```

## Summary

Just because we can omit types, this doesn't mean that we should do it. We specify explicit types for readability and safety, especially for exposed APIs.

## Item 14: Consider referencing receivers explicitly

One common place where we might choose a longer structure to make something explicit is when we want to highlight that a function or a property is taken from the receiver instead of being a local or top-level variable. In the most basic situation, this means a reference to the class with which the method is associated:

```
class User: Person() {  
    private var beersDrunk: Int = 0  
  
    fun drinkBeers(num: Int) {  
        // ...  
        this.beersDrunk += num  
        // ...  
    }  
}
```

Similarly, we may explicitly reference an extension receiver (`this` in an extension method) to make its use more explicit. Just compare the following Quicksort implementation, which is written without explicit receivers:

```
fun <T : Comparable<T>> List<T>.quickSort(): List<T> {  
    if (size < 2) return this  
    val pivot = first()  
    val (smaller, bigger) = drop(1)  
        .partition { it < pivot }  
    return smaller.quickSort() + pivot + bigger.quickSort()  
}
```

This implementation uses them:

```
fun <T : Comparable<T>> List<T>.quickSort(): List<T> {  
    if (this.size < 2) return this  
    val pivot = this.first()  
    val (smaller, bigger) = this.drop(1)  
        .partition { it < pivot }  
    return smaller.quickSort() + pivot + bigger.quickSort()  
}
```

The usage is the same for both functions:

```
listOf(3, 2, 5, 1, 6).quickSort() // [1, 2, 3, 5, 6]
listOf("C", "D", "A", "B").quickSort() // [A, B, C, D]
```

## Many receivers

Using explicit receivers can be especially helpful when we are in the scope of more than one receiver. We are often in such a situation when we use the `apply`, `with`, or `run` functions. Such situations are dangerous, so we should avoid them. It is safer to use an object that uses an explicit receiver. To understand this problem, see the following code<sup>21</sup>:

```
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .apply { print("Created ${name}") }

    fun create(name: String): Node? = Node(name)
}

fun main() {
    val node = Node("parent")
    node.makeChild("child")
}
```

What is the result? Stop now and spend some time trying to figure it out before reading the answer.

You probably expect the result to be “Created parent.child”, but the actual result is “Created parent”. Why? To investigate, we can use an explicit receiver before `name`:

---

<sup>21</sup>Inspired by a puzzle originally added by Roman Dawydkin to the Dmitry Kandalov collection and later presented at KotlinConf by Anton Keks.

```
class Node(val name: String) {  
  
    fun makeChild(childName: String) =  
        create("$name.$childName")  
            .apply { print("Created ${this.name}") }  
            // Compilation error  
  
    fun create(name: String): Node? = Node(name)  
}
```

The problem is that the type `this` inside `apply` is `Node?`, so methods cannot be used directly. We would need to unpack such a receiver first by using, for instance, a safe call. If we do so, the result will finally be correct:

```
class Node(val name: String) {  
  
    fun makeChild(childName: String) =  
        create("$name.$childName")  
            .apply { print("Created ${this?.name}") }  
  
    fun create(name: String): Node? = Node(name)  
}  
  
fun main() {  
    val node = Node("parent")  
    node.makeChild("child")  
    // Prints: Created parent.child  
}
```

This is an example of bad usage of `apply`. We wouldn't have such a problem if we used `also` instead and called `name` on the parameter. Using `also` forces us to reference the function's receiver explicitly in the same way as an explicit receiver. In general, `also` and `let` are much better choices for additional operations or when we operate on a nullable value.

```
class Node(val name: String) {  
  
    fun makeChild(childName: String) =  
        create("$name.$childName")  
            .also { print("Created ${it?.name}") }  
  
    fun create(name: String): Node? = Node(name)  
}
```

When a receiver is unclear, we prefer to avoid it or clarify it using an explicit receiver. When we use a receiver without a label, we mean the closest one. When we want to use an outer receiver, we need to use a label. In such cases, it is especially useful to use this label explicitly. Here is an example showing both apply and an explicit receiver in use:

```
class Node(val name: String) {  
  
    fun makeChild(childName: String) =  
        create("$name.$childName").apply {  
            print("Created ${this?.name} in "+  
                " ${this@Node.name}")  
        }  
  
    fun create(name: String): Node? = Node(name)  
}  
  
fun main() {  
    val node = Node("parent")  
    node.makeChild("child")  
    // Created parent.child in parent  
}
```

This way, the direct receiver clarifies which receiver we mean. This is important information that might not only protect us from errors but also improve readability.

## DSL marker

There is a context in which we often operate in very nested scopes with different receivers, and we don't need to use explicit receivers at all. I am talking about Kotlin DSLs. We don't need to use receivers explicitly because this is how DSLs are designed. However, in DSLs, it is especially dangerous to accidentally use

functions from an outer scope. Think about this simple HTML DSL that makes an HTML table:

```
table {
    tr {
        td { +"Column 1" }
        td { +"Column 2" }
    }
    tr {
        td { +"Value 1" }
        td { +"Value 2" }
    }
}
```

Notice that, by default, in every scope we can also use methods from receivers from the outer scope. We might use this fact to mess with this DSL:

```
table {
    tr {
        td { +"Column 1" }
        td { +"Column 2" }
        tr {
            td { +"Value 1" }
            td { +"Value 2" }
        }
    }
}
```

To restrict such usage, we have a special meta-annotation (an annotation for annotations) that restricts the implicit usage of outer receivers. This is the `DslMarker`. When we use it on an annotation and later use this annotation on a class that serves as a builder, using an implicit receiver won't be possible inside this builder. Here is an example of how `DslMarker` might be used:

```
@DslMarker
annotation class HtmlDsl

fun table(f: TableDsl.() -> Unit) { /*...*/ }

@HtmlDsl
class TableDsl { /*...*/ }
```

When we use an annotation annotated with `DslMarker`, it is prohibited to use the outer receiver implicitly:

```
table {  
    tr {  
        td { +"Column 1" }  
        td { +"Column 2" }  
        tr { // COMPILATION ERROR  
            td { +"Value 1" }  
            td { +"Value 2" }  
        }  
    }  
}
```

Using functions from an outer receiver requires explicit receiver usage:

```
table {  
    tr {  
        td { +"Column 1" }  
        td { +"Column 2" }  
        this@table.tr {  
            td { +"Value 1" }  
            td { +"Value 2" }  
        }  
    }  
}
```

The DSL marker is a very important mechanism that we use to force the explicit usage of either the closest receiver or the outer receiver. Respect DSL design and use it accordingly.

## Summary

Do not change a scope's receiver just because you can. It might be confusing to have too many receivers that all give us methods we can use. An explicit argument or reference is generally better. When we do change a receiver, using explicit receivers improves readability because it clarifies where the function comes from. We can even use a label when there are many receivers, and we need to clarify which one a function comes from. If you want to force the use of explicit receivers from the outer scope, you can use the `DslMarker` meta-annotation.

## Item 15: Properties should represent a state, not a behavior

Kotlin properties look similar to Java fields, but they actually represent a different concept.

```
// Kotlin property
var name: String? = null

// Java field
String name = null;
```

Even though they can be used the same way to hold data, we need to remember that properties have many more capabilities, the first of which is the fact that they can always have custom setters and getters:

```
var name: String? = null
    get() = field?.toUpperCase()
    set(value) {
        if(!value.isNullOrEmpty()) {
            field = value
        }
    }
```

You can see here that we are using the `field` identifier. This is a reference to the backing field, which lets us hold data in this property. Such backing fields are generated by default because the default implementations of setters and getters use them. We can also implement custom accessors that do not use them; in this case, the property will not have a `field` at all. For instance, a Kotlin property can be defined using only a getter for a read-only `val` property :

```
val fullName: String
    get() = "$name $surname"
```

For a read-write `var` property, we can make a property by defining a getter and a setter. Such properties are known as *derived properties*, and they are not uncommon. They are the main reason why all properties in Kotlin are encapsulated by default. Just imagine that you have to hold a date in your object and you use `Date` from the Java stdlib. Then, at some point for some reason, the class cannot store the property of this type anymore, perhaps because of a serialization issue or maybe because you lifted this class to a common module. The problem is that

this property has been referenced throughout your project. With Kotlin, this is no longer a problem as you can move your data into a separate `millis` property and modify the `date` property not to hold data but instead to wrap/unwrap that other property.

```
var date: Date
    get() = Date(millis)
    set(value) {
        millis = value.time
    }
```

Properties do not need fields. Rather, they conceptually represent accessors (getter for `val`; getter and setter for `var`). This is why we can define them in interfaces:

```
interface Person {
    val name: String
}
```

This means that this interface promises to have a getter. We can also override properties:

```
open class Supercomputer {
    open val theAnswer: Long = 42
}

class AppleComputer : Supercomputer() {
    override val theAnswer: Long = 1_800_275_2273
}
```

For the same reason, we can delegate properties:

```
val db: Database by lazy { connectToDb() }
```

Because properties are essential functions, we can make extension properties as well:

```

val Context.preferences: SharedPreferences
  get() = PreferenceManager
    .getDefaultSharedPreferences(this)

val Context.inflater: LayoutInflater
  get() = getSystemService(
    Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater

val Context.notificationManager: NotificationManager
  get() = getSystemService(Context.NOTIFICATION_SERVICE)
    as NotificationManager

```

As you can see, **properties represent accessors, not fields**. This way, they can be used instead of some functions, but we should be careful what we use them for. Properties should not be used to represent algorithmic behavior, as in the example below:

```

// DON'T DO THIS!
val Tree<Int>.sum: Int
  get() = when (this) {
    is Leaf -> value
    is Node -> left.sum + right.sum
  }

```

Here, the `sum` property iterates over all elements, representing algorithmic behavior. So, this property is misleading: finding the answer can be computationally heavy for big collections, which is not expected for a getter. This should be a function, not a property:

```

fun Tree<Int>.sum(): Int = when (this) {
  is Leaf -> value
  is Node -> left.sum() + right.sum()
}

```

The general rule is that **we should use properties only to represent or set a state, and no other logic should be involved**. A useful heuristic to decide if something should be a property is: If I defined this property as a function, would I prefix it with `get/set`? If not, it should probably not be a property. More concretely, here are the most typical situations when we should use functions instead of properties:

- **An operation is computationally expensive or has computational complexity higher than O(1)** - A user does not expect using a property to be expensive.

- **An operation throws an exception** - A user does not expect that property getter or setter can throw an exception.
- **It involves business logic (how the application acts)** - when we read code, we do not expect a property to do anything more than simple actions like logging, notifying listeners, or updating a bound element.
- **It is not deterministic** - Calling a getter should not change a state, and calling a setter twice in succession should produce the same result (unless the object has been modified in another thread).
- **It is a conversion, such as `Int.toDouble()`** - It is a matter of convention that conversions are methods. Using a property would seem like referencing some part of the state that is wrapped over by the object.
- **Getters should not change property state** - We expect to be able to use getters freely without worrying about property state modifications.

For instance, calculating the sum of some elements requires iterating over all of them (this is behavior, not a state) and has linear complexity. Therefore, it should not be a property and is defined in the standard library as a function:

```
val s = (1..100).sum()
```

On the other hand, to get and set a state, we use properties in Kotlin, and we should not involve functions unless there is a good reason. We use properties to represent and set state; if you need to modify them later, use custom getters and setters:

```
// DON'T DO THIS!
class UserIncorrect {
    private var name: String = ""

    fun getName() = name

    fun setName(name: String) {
        this.name = name
    }
}

class UserCorrect {
    var name: String = ""
}
```

A simple rule of thumb is that **a property describes and sets a state, while a function describes a behavior**.

## Item 16: Avoid returning or operating on Unit?

During the recruitment process, a dear friend of mine was asked, “Why might one want to return `Unit?` from a function?”. Well, `Unit?` has only 2 possible values: `Unit` or `null`. Just like `Boolean` can be either `true` or `false`. Ergo, these types are isomorphic, so they can be used interchangeably. Why would we want to use `Unit?` instead of `Boolean` to represent something? I have no other answer than that one can use the Elvis operator or a safe call. So instead of:

```
fun keyIsCorrect(key: String): Boolean = //...  
  
if (!keyIsCorrect(key)) return
```

We are able to do this:

```
fun verifyKey(key: String): Unit? = //...  
  
verifyKey(key) ?: return
```

This appears to be the expected answer. What was missing in my friend’s interview was a way more important question: “Should we do it?”. This trick looks nice when we are writing the code, but not necessarily when we are reading it. Using `Unit?` to represent logical values is misleading and can lead to errors that are hard to detect. We’ve already discussed how this expression can be surprising:

```
getData()  
?.let { view.showData(it) }  
?: view.showError()
```

When `showData` returns `null` and `getData` returns not `null`, both `showData` and `showError` will be called. Using standard if-else is less tricky and more readable:

```
if (person != null && person.isAdult) {  
    view.showPerson(person)  
} else {  
    view.showError()  
}
```

Compare the following two notations:

```
if(!keyIsCorrect(key)) return
```

```
verifyKey(key) ?: return
```

I have never found even a single case when `Unit?` is the most readable option. It is misleading and confusing. It should nearly always be replaced by `Boolean`. This is why the general rule is that we should avoid returning or operating on `Unit?`.

## Item 17: Consider naming arguments

When you read code, it is not always clear what an argument means. Take a look at the following example:

```
val text = (1..10).joinToString("|")
```

What is "|"? If you know `joinToString` well, you know that it is the separator, although it could just as well be the prefix. It is not clear at all<sup>22</sup>. We can make it easier to read by clarifying arguments whose values do not clearly indicate what they mean. The best way to do this is by using named arguments:

```
val text = (1..10).joinToString(separator = "|")
```

We could achieve a similar result by using a naming variable:

```
val separator = "|"  
val text = (1..10).joinToString(separator)
```

However, naming the argument is more reliable. A variable name specifies a developer's intention but not necessarily its correctness. What if a developer made a mistake and placed the variable in the wrong position? What if the order of parameters changed? Named arguments protect us from such situations, but named values do not. This is why it is still reasonable to use named arguments when we have values in well-named variables:

```
val separator = "|"  
val text = (1..10).joinToString(separator = separator)
```

### When should we use named arguments?

Named arguments are longer, but they have two important advantages:

- The name indicates which value is expected.
- They are safer because they are independent of order.

An argument's name is important information not only for a developer using this function but also for one trying to understand how it was used. Take a look at this call:

---

<sup>22</sup>IntelliJ now helps by displaying hints when you put literals in a function call, but this can be turned off, or you might be using a different IDE.

```
sleep(100)
```

For how long will it sleep? 100 ms? Maybe 100 seconds? We can clarify it using a named argument:

```
sleep(timeMillis = 100)
```

This is not the only option for clarification in this case. In statically typed languages like Kotlin, the first mechanism that protects us when we pass arguments is the parameter type. We could use it here to express information about the time unit:

```
sleep(Millis(100))
```

Or, we could use an extension property to create DSL-like syntax, like `milliseconds` from Kotlin stdlib, which creates an instance of `Duration`:

```
sleep(100.milliseconds)
```

Types are a good way to pass such information, but this still might not be enough. Some arguments might still be unclear or might still be placed in the wrong positions. This is why I still suggest considering named arguments, especially for parameters:

- with default arguments,
- with the same type as other parameters,
- of functional types (those not in the last parameter position).

Let's consider each of these cases.

## Parameters with default arguments

When a parameter has a default argument, we should nearly always use it by name. Such optional parameters are changed more often than those that are required, and we don't want to miss such changes. A function's name generally indicates what its non-optional arguments are, but not what its optional ones are. This is why it is safer and generally cleaner to name optional arguments.<sup>23</sup>

## Many parameters with the same type

As we said, when parameters have different types, we are generally safe from placing an argument in an incorrect position. There is no such comfort when some parameters have the same type.

---

<sup>23</sup>Some best practices for other languages even suggest always naming optional arguments. One popular example is Effective Python by Brett Slatkin, in Item 19: Provide Optional Behavior with Keyword Arguments.

```
fun sendEmail(to: String, message: String) {  
    /*...*/  
}
```

With a function like this, it is good to clarify arguments using names:

```
sendEmail(  
    to = "contact@kt.academy",  
    message = "Hello, ..."  
)
```

## Parameters with function types

Finally, we should treat parameters with function types specially. There is a special position for such parameters in Kotlin: the last position. Sometimes a function name describes an argument of a function type. For instance, when we see `repeat`, we expect that a lambda after it is the block of code that should be repeated. When you see `thread`, it is intuitive that the block after it is the body of this new thread. Such names only describe the function used at the last position.

```
thread {  
    // ...  
}
```

All other arguments with function types should be named because it is easy to misinterpret them. For instance, take a look at this simple view DSL:

```
val view = linearLayout {  
    text("Click below")  
    button({ /* 1 */ }, { /* 2 */ })  
}
```

Which function is a part of this builder, and which one is an on-click listener? We should clarify this by naming the listener and moving the builder outside of the arguments:

```
val view = linearLayout {
    text("Click below")
    button(onClick = { /* 1 */ }) {
        /* 2 */
    }
}
```

Multiple optional parameters with function types can be especially confusing:

```
fun call(before: () -> Unit = {}, after: () -> Unit = {}) {
    before()
    print("Middle")
    after()
}

call({ print("CALL") }) // CALLMiddle
call { print("CALL") } // MiddleCALL
```

To prevent such situations, when no single argument of a function type has a special meaning, name all arguments with function types:

```
call(before = { print("CALL") }) // CALLMiddle
call(after = { print("CALL") }) // MiddleCALL
```

This is especially true for reactive libraries. For instance, in RxJava, when we subscribe to an Observable, we can set functions that should be called:

- on every received item
- in the case of an error,
- after the observable is finished.

In Java, I've often seen people using lambda expressions to set up these functions and specifying in comments which method each lambda expression is:

```
// Java
observable.getUsers()
    .subscribe((List<User> users) -> { // onNext
        // ...
    }, (Throwable throwable) -> { // onError
        // ...
    }, () ->{ // onCompleted
        // ...
    });
}
```

In Kotlin, we can make a step forward and use named arguments instead:

```
observable.getUsers()
    .subscribeBy(
        onNext = { users: List<User> ->
            // ...
        },
        onError = { throwable: Throwable ->
            // ...
        },
        onCompleted = {
            // ...
        }
    )
)
```

Notice that I changed the function name from `subscribe` to `subscribeBy`. This is because RxJava is written in Java, and **we cannot use named arguments when we call Java functions** because Java does not preserve information about function names. To be able to use named arguments, we often need to make Kotlin wrappers for these functions (extension functions that are alternatives to these functions).

## Summary

Named arguments are not only useful when we need to skip some default values: they are important information for other developers who read our code, and they can improve the safety of our code. We should consider them, especially when we have multiple parameters with the same type (or with functional types), and for optional arguments. When we have multiple parameters with functional types, they should almost always be named. An exception is the last function argument when it has a special meaning, like in DSL.

## Item 18: Respect coding conventions

### Basics

Kotlin has well-established coding conventions that are described in the documentation in a section aptly called “Coding Conventions”<sup>24</sup>. **These conventions are not optimal for all projects, but it is optimal for us as a community to have conventions that are respected in all projects.** Thanks to them:

- It is easier to switch between projects
- Code is more readable, even for external developers
- It is easier to guess how code works
- It is easier to later merge code with a common repository or to move some parts of code from one project to another

Programmers should get familiar with the conventions that are described in the documentation. They should also be respected when they change, as might happen to some degree over time. Since it is hard to do both, there are two tools that help:

- The IntelliJ formatter can be set up to automatically format according to the official Coding Conventions style. For that, go to Settings | Editor | Code Style | Kotlin, click on the “Set from...” link in the upper right corner and select “Predefined style / Kotlin style guide” from the menu.
- ktlint<sup>25</sup> - a popular linter that analyzes your code and notifies you about all coding convention violations.

Looking at Kotlin projects, I see that most of them are intuitively consistent with most of the conventions. This is probably because Kotlin mostly follows the Java coding conventions, and most Kotlin developers today used to be Java developers. One rule that I often see violated is how classes and functions should be formatted. According to the conventions, classes with a short primary constructor can be defined in a single line:

```
class FullName(val name: String, val surname: String)
```

However, classes with many parameters should be formatted such that every parameter is on another line<sup>26</sup>, and there is no parameter on the first line:

---

<sup>24</sup>Link: [kt.academy/l/kotlin-conventions](https://kt.academy/l/kotlin-conventions)

<sup>25</sup>Link: [kt.academy/l/ktlint](https://kt.academy/l/ktlint)

<sup>26</sup>It is possible to have parameters that are strongly related to each other, like x and y on the same line, but I am not in favor of that.

```
class Person(  
    val id: Int = 0,  
    val name: String = "",  
    val surname: String = ""  
) : Human(id, name) {  
    // body  
}
```

Similarly, this is how we format a long function:

```
public fun <T> Iterable<T>.joinToString(  
    separator: CharSequence = ", ",  
    prefix: CharSequence = "",  
    postfix: CharSequence = "",  
    limit: Int = -1,  
    truncated: CharSequence = "...",  
    transform: ((T) -> CharSequence)? = null  
) : String {  
    // ...  
}
```

Notice that these two are very different from the convention that leaves the first parameter on the same line and then indents all others to it.

```
// Don't do that  
class Person(val id: Int = 0,  
            val name: String = "",  
            val surname: String = "") : Human(id, name){  
    // body  
}
```

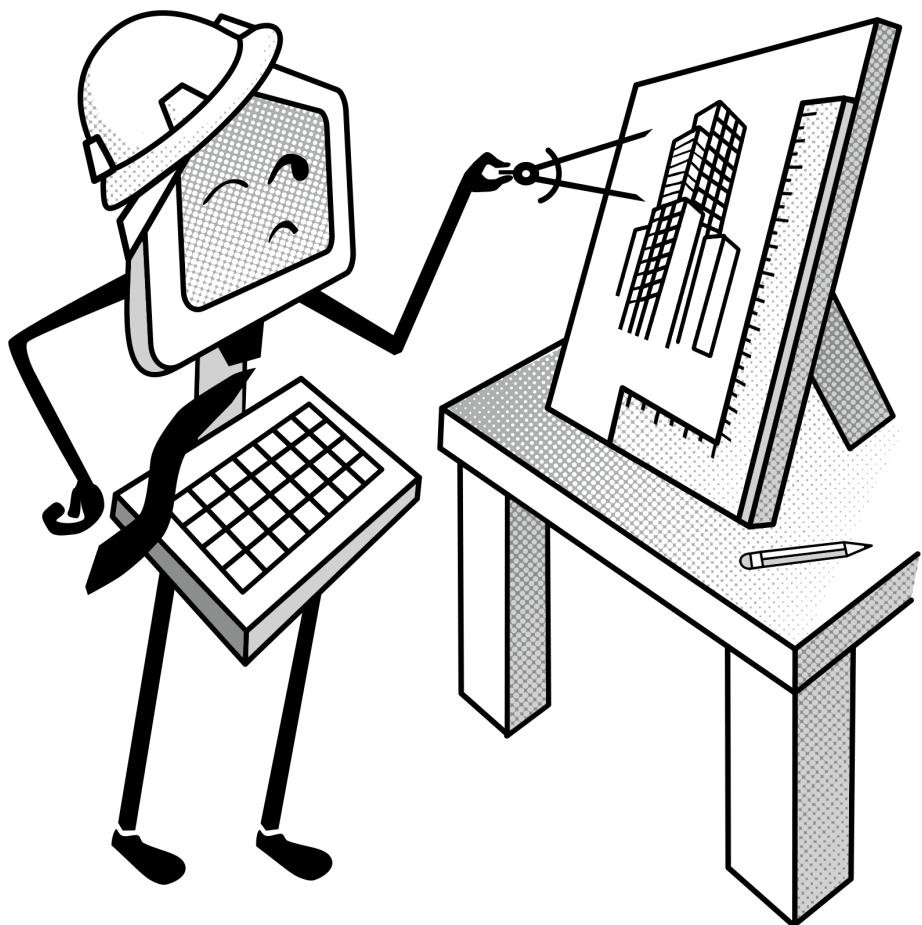
This can be problematic in two ways:

- Arguments in every class start with a different indentation based on the class name. Also, when we change the class name, we need to adjust the indentations of all primary constructor parameters.
- Classes defined this way still tend to be too wide. The width of a class defined this way is the class name with the `class` keyword and the longest primary constructor parameter, or the last parameter plus superclasses and interfaces.

Some teams might decide to use slightly different conventions. This is fine, but these conventions should then be respected throughout the project. **Every project should look like it was written by a single person, not a group of people fighting with each other.**

Coding conventions are often not respected enough by developers, but they are important, so a chapter dedicated to readability in a best practices book couldn't be closed without at least a short section dedicated to them. Read them, use static checkers to help you be consistent with them, and apply them in your projects. By respecting coding conventions, we make Kotlin projects better for us all.

## Part 2: Code design



# Chapter 3: Reusability

Have you ever wondered how the `System.out.print` function works<sup>27</sup>? It's one of the most basic functions, used again and again, but would you be able to implement it yourself if it vanished one day? The truth is that this wouldn't be an easy task, especially if the rest of `java.io` also vanished. You would need to implement the communication separately for each operating system you support in C using JNI<sup>28</sup>. Believe me, implementing it once is terrifying. Implementing it again and again in every project would be a horror.

The same applies to many other functions as well. Making Android views is so easy because Android has a complex API that supports it. Backend developers don't need to know much about the HTTP(S) protocol, even though they work with it every day. You don't need to know any sorting algorithms to call `sorted` on an iterable object. Thankfully, we don't need to have and use all this knowledge every day. Someone implemented it once, and now we can use it whenever we need. This demonstrates a key feature of Programming languages: *reusability*.

This all sounds great, but code reusability is as dangerous as it is powerful. A small change in the `print` function could break countless programs. If we extract a common part from A and B, we have an easier job in the future when we need to change them both, but it's harder and more error-prone when we need to change only one.

This chapter is dedicated to reusability and it touches on many things that developers do intuitively because we've learned them through practice - mainly by observing how something we did in the past impacts us now. We extracted something, and now it causes problems; we didn't extract something, and now we have a problem when we need to make some changes. Sometimes we have to deal with code written years ago by different developers, and we see how their decisions impact us now. Maybe we just looked at another language or project and we thought "Wow, this is short and readable because they did X". This is the way we typically learn, and this is one of the best ways to learn.

There is one problem though: this requires years of practice. To speed up this process and help you systematize this knowledge, this chapter will give you some generic rules to help you make your code better in the long term. It is a bit more theoretical than the rest of the book. If you're looking for concrete rules (as presented in the previous chapters), feel free to skip it.

---

<sup>27</sup>This is the function that stands behind `println` in Kotlin/JVM.

<sup>28</sup>Read about it in this article: [kt.academy/l/println](https://kt.academy/l/println)

## Item 19: Do not repeat knowledge

Not Kotlin-specific

Basics

The first big rule I was taught about programming was:

If you use copy-paste in your project, you are most likely doing something wrong.

This is a very simple heuristic, but it is also very wise. Even now, whenever I reflect on this I am amazed how well a single and clear sentence expresses the key idea behind the “Do not repeat knowledge” principle. This is also often known as the Don’t Repeat Yourself (DRY) principle, which comes from the Pragmatic Programmer book by Andy Hunt and Dave Thomas. Some developers might be familiar with the WET antipattern<sup>29</sup>, which sarcastically teaches us the same. DRY/WET are also strongly connected to the Single Source of Truth (SSOT) practice. As you can see, this rule is quite popular and has many names, but it is often misused or abused. To understand this rule and the reasons behind it clearly, we need to introduce a bit of theory.

### Knowledge

Let’s define knowledge in programming broadly as any piece of intentional information. It could be code or data, or it could be a lack of code or data, which means that we want to use the default behavior. For instance, when we inherit and we don’t override a method, it’s like saying that we want this method to behave the same as in the superclass.

Everything in our projects is some kind of knowledge when it is defined this way. Of course, there are many different kinds of knowledge: how an algorithm should work, what a UI should look like, what result we wish to achieve, etc. There are also many ways to express knowledge: for example by using code, configurations, or templates. In the end, every single piece of our program is information that can be understood by some tool, virtual machine, or directly by other programs.

There are two particularly important kinds of knowledge in our programs:

1. Logic - How we expect our program to behave and what it should look like.
2. Common algorithms - Implementation of algorithms to achieve the desired behavior.

---

<sup>29</sup>Stands for We Enjoy Typing, Waste Everyone’s Time or Write Everything Twice.

The main difference between these is that business logic changes a lot over time, while common algorithms generally do not change once they are defined. They might be optimized or we might replace one algorithm with another, but algorithms themselves are generally stable. Because of this difference, we'll concentrate on algorithms in the next item. For now, let's concentrate on the first point: the logic - knowledge about our program.

## Everything can change

There is a saying that the only constant in programming is change. Just think about projects from 10 or 20 years ago, which isn't a long time. Can you think of a single popular application or website that hasn't changed for so many years? Android was released in 2008. The first stable version of Kotlin was released in 2016. Not only technologies but also languages change so quickly. Think about your old projects. Most likely now you would use different libraries, architecture, and design.

Changes often occur where we don't expect them. There is a story that once, when Einstein was examining his students, one of them stood up and loudly complained that the questions were the same as the previous year. Einstein responded that it was true but the answers were totally different that year. Even things that you think are constant because they are based on law or science might change one day. Nothing is absolutely safe.

Standards of UI design and technologies change much faster. Our understanding of clients often needs to change on a daily basis. This is why the knowledge in our projects will also change. For instance, here are some very typical reasons for these changes:

- A company learns more about its users' needs or habits.
- Design standards change.
- We need to adjust to changes in a platform, libraries, or tools.

Most projects nowadays change their requirements and parts of internal structure every few months. This is often something desired. Many popular management systems are agile and can support constant changes in requirements. Slack was initially a game named Glitch<sup>30</sup>. The game didn't work out, but customers liked its communication features.

Things change, and we should be prepared for that. **The biggest enemy of change is knowledge repetition.** Just think for a second: what if we need to change something that is repeated in many places in our program? The simplest answer is that, in such a case, you just need to search for all the places where this knowledge

---

<sup>30</sup>See the presentation *How game mechanics can drive product loyalty* by Ali Rayl.

is repeated and change it everywhere. Searching can be frustrating, and it is also troublesome: What if you forget to change some repetitions? What if some of them have already been modified because they were integrated with other functionalities? It might be tough to change them all in the same way. These are real problems.

To make the problem less abstract, think of a universal button used in many different places in a project. If our graphic designer decides that this button needs to be changed, we would have a problem if we defined how it looks in every single usage. We would need to search our whole project and change every single instance separately. We would also need to ask the testers to check that we haven't missed any instances.

Another example: Let's say that we use a database in our project, then one day, we change the name of a table. If we forget to adjust all SQL statements that depend on this table, we might have a very problematic error. If we had a table structure that is defined only once, we wouldn't have such a problem.

In both examples, you can see how dangerous and problematic knowledge repetition is. It makes projects less scalable and more fragile. The good news is that we programmers have worked for years on tools and features that help us eliminate knowledge redundancy. On most platforms, we can define a custom style for a button or a custom view/component to represent it. Instead of writing SQL in text format, we can use an ORM (like Hibernate) or a DAO (like Exposed).

All these solutions represent different kinds of abstractions, and they protect us from different kinds of redundancy. An analysis of different kinds of abstractions is presented in Item 26: *Use abstraction to protect code against changes*.

## When should we allow code repetition?

There are situations where we can see two pieces of code that are similar but should not be extracted into one. In this case, they look similar but represent different knowledge.

Let's start with an example. Let's say we have two independent Android applications in the same project. Their build tool configurations are similar, so it might be tempting to extract them. But what if we do that? These two applications are independent, so if we need to change something in the configuration, we will most likely need to change it in only one of them. Changes after this reckless extraction are harder, not easier. Configuration reading is harder as well. Configurations have boilerplate code, but developers are already familiar with it. Making abstractions means designing our own API, but it is another thing to learn for a developer using this API. This is a perfect example of how problematic it is to extract something that is not conceptually the same knowledge.

The most important question to ask ourselves when we decide if two pieces of code represent similar knowledge is: **Are they more likely to change together or**

**separately?** Pragmatically, this is the most important question because this is the biggest result of extracting a common part: **it is easier to change them both at the same time, but harder to change only one usage.**

One useful heuristic is that if business rules come from different sources, we should assume that they are more likely to change independently. For such a case, we even have a rule that protects us from unintended code extraction. It is called the Single Responsibility Principle.

## The single responsibility principle

A very important rule that teaches us when we should not extract common code is the Single Responsibility Principle from SOLID. It states that “A class should have only one reason to change”. This rule<sup>31</sup> can be simplified by the statement that there should be no situations in which two actors need to change the same class. By actor, we mean the source of a change. Actors are often developers from different departments who know little about each other’s work and domains. Even if there is only a single developer in a project, there might be multiple managers, each of which should be treated as a separate actor. These are two sources of changes that know little about each other’s domains. The situation in which two actors edit the same piece of code is especially dangerous.

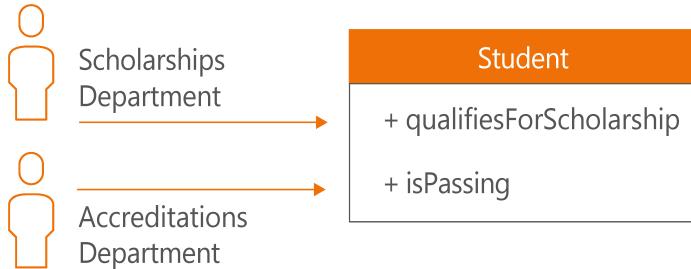
Let’s see an example. Imagine that we work for a university, and we have a `Student` class. This class is used by both the Scholarships Department and the Accreditations Department. Developers from those two departments have introduced two different functions:

- `isPassing` was created by the Accreditations Department and answers the question of whether a student is passing.
- `qualifiesForScholarship` was created by the Scholarships Department and answers the question of whether a student has enough points to qualify for a scholarship.

Both functions need to calculate how many points a student collected in the previous semester, so a developer extracted a function `calculatePointsFromPassedCourses`.

---

<sup>31</sup>As described by the software engineer Robert C. Martin in his book *Clean Architecture*.



```

class Student {
    // ...

    fun isPassing(): Boolean =
        calculatePointsFromPassedCourses() > 15

    fun qualifiesForScholarship(): Boolean =
        calculatePointsFromPassedCourses() > 30

    private fun calculatePointsFromPassedCourses(): Int {
        //...
    }
}
  
```

Then, the original rules changed, and the dean decided that less important courses should not qualify for the scholarship points calculation. The developer who was tasked with introducing this change checks the `qualifiesForScholarship` function, finds that it calls the private method `calculatePointsFromPassedCourses`, and changes it to omit courses that do not qualify. Unintentionally, that developer changed the behavior of `isPassing` as well. Students who were supposed to pass were informed that they had failed the semester. You can imagine their reaction<sup>32</sup>.

It is true that we could easily prevent such a situation if we had unit tests (Item 8: Write unit tests), but let's skip this aspect for now.

The developer might check where else the function is used. However, the problem is that this developer didn't know that this private function was used by another

<sup>32</sup>I imagine an angry mob of students storming university edifice with torches and rulers. Sounds abstract? Then read about St Scholastica Day riot.

property with a totally different responsibility. Private functions are rarely used by more than one function.

The problem, in general, is that it is easy to couple responsibilities located very close to each other (in the same class/file). A simple solution would be to extract these responsibilities into separate classes. We might have separate `StudentIsPassingValidator` and `StudentQualifiesForScholarshipValidator` classes, but in Kotlin, we don't need to use such heavy artillery (see more in *Chapter 4: Design abstractions*). We can just define `qualifiesForScholarship` and `calculatePointsFromPassedCourses` as extension functions on `Student` that are located in separate modules: one for which the Scholarships Department is responsible, and another for which the Accreditations Department is responsible.

```
// scholarship module
fun Student.qualifiesForScholarship(): Boolean {
    /*...*/
}

// accreditations module
fun Student.calculatePointsFromPassedCourses(): Boolean {
    /*...*/
}
```

What about extracting a function for calculating results? We can do this, but it cannot be a private function that is used as a helper for both of these methods. Instead, it can be:

1. A general public function defined in a module used by both departments. In such a case, the common part is treated as something common, so a developer should not change it without modifying the contract and adjusting usages.
2. Two separate helper functions, one for each department.

Both options are safe. The Single Responsibility Principle teaches us two things:

1. Knowledge from two different sources (here, two different departments) is very likely to change independently, so we should treat these sources as two different types of knowledge.
2. We should separate different types of knowledge because otherwise it is tempting to reuse parts of our code that should not be reused.

## Summary

Everything changes and it is our job to prepare for that: to recognize common knowledge and extract it. If some elements have similar parts that we will likely need to change for all instances, extract them as this will save time on searching through the project to update many instances. On the other hand, protect yourself from unintentional modifications by separating parts from different sources. Often, this is the most important side of the problem. I see many developers who are so terrified of the literal meaning of “Don’t Repeat Yourself” that they tend to look suspiciously at any 2 lines of code that look similar. Both extremes are unhealthy, and we need to always search for a balance. Sometimes, it is a tough decision whether something should be extracted or not. This is why designing information systems well is an art that requires time and a lot of practice.

## Item 20: Do not repeat common algorithms

### Basics

I often see developers reimplementing the same algorithms again and again. By algorithms, here I mean patterns that are not project-specific, so they do not contain any business logic and can be extracted into separate modules or even libraries. These might be mathematical operations, collection processing, or any other common behavior. Sometimes these algorithms can be long and complicated, like optimized sorting algorithms. There are also many simple examples though, like number coercion in a range:

```
val percent = when {  
    numberFromUser > 100 -> 100  
    numberFromUser < 0 -> 0  
    else -> numberFromUser  
}
```

Notice that we don't need to implement this because it is already in the stdlib as the `coerceIn` extension function:

```
val percent = numberFromUser.coerceIn(0, 100)
```

The advantages of extracting even short but repetitive algorithms are:

- **Programming is faster** because making a single call is easier than implementing an algorithm.
- **They are named, so we can recognize an algorithm by name instead of by reading its implementation.** This is easier for developers who are familiar with a given algorithm. This might be harder for new developers who are not familiar with a given concept, but it pays off to learn the names of repetitive algorithms. Once we learn these names, we can benefit from that in the future.
- **We eliminate noise, making it easier to notice something atypical.** In a long algorithm, it is easy to miss hidden pieces of atypical logic. Think of the difference between `sortedBy` and `sortedByDescending`. The sorting direction is clear when we call those functions, even though their bodies are nearly identical. If we needed to implement this logic every time, it would be easy to confuse whether the implemented sorting has ascending or descending order. Comments before an algorithm implementation are not helpful either. Practice shows that developers often change code without updating comments, so over time, we lose trust in comments.
- **They can be optimized once, and we profit from this optimization everywhere we use these functions.**

## Learn the standard library

Common algorithms have nearly always already been defined by someone else. Most libraries are just collections of common algorithms. The most special among them is the stdlib (standard library), which is a huge collection of utilities, mainly defined as extension functions. Learning the stdlib functions can be demanding, but it is worth it. Without it, developers have to reinvent the wheel time and time again. To see an example, take a look at this snippet from an open-source project:

```
override fun saveCallResult(item: SourceResponse) {
    var sourceList = ArrayList<SourceEntity>()
    item.sources.forEach {
        var sourceEntity = SourceEntity()
        sourceEntity.id = it.id
        sourceEntity.category = it.category
        sourceEntity.country = it.country
        sourceEntity.description = it.description
        sourceList.add(sourceEntity)
    }
    db.insertSources(sourceList)
}
```

Using `forEach` here is useless. I see no advantage to using it instead of a `for`-loop. What I do see in this code, though, is a mapping from one type to another. We can use the `map` function in such cases. Another thing to note is that the way `SourceEntity` is set up is far from perfect. This is a JavaBean pattern that is obsolete in Kotlin; instead, we should use a factory method or a primary constructor (Chapter 5: Object creation). If, for some reason, someone needs to keep it this way, we should at least use `apply` to set up all the properties of a single object implicitly. This is our function after a small clean-up:

```
override fun saveCallResult(item: SourceResponse) {
    val entries = item.sources.map(Source::toEntry)
    db.insertSources(entries)
}

private fun Source.toEntry() = SourceEntity().apply {
    id = this.id
    category = this.category
    country = this.country
    description = this.description
}
```

## Implementing your own utils

At some point in every project, we need some algorithms that are not in the standard library. For instance, what if we need to calculate the product of the numbers in a collection? This is a well-known abstraction, so it is good to define it as a universal utility function:

```
fun Iterable<Int>.product() =  
    fold(1) { acc, i -> acc * i }
```

You don't need to wait for more than one use. A product is a well-known mathematical concept, and its name should be clear to developers. Maybe another developer will need to use it in the future, and they'll be happy to see that it is already defined. Hopefully, that developer will find this function. It is bad practice to have duplicate functions achieving the same results. Each function needs to be tested, remembered, and maintained, all of which should be considered costs. We should not define functions we don't need, therefore, we should first search for an existing function before implementing our own.

Notice that `product`, just like most functions in the Kotlin stdlib, is an extension function. There are many ways we can extract common algorithms, starting from top-level functions and property delegates and ending up with classes. However, extension functions are a really good choice because:

- Functions do not hold states, so they are perfect for representing behavior, especially if it has no side effects.
- Compared to other top-level functions, extension functions are better because they are suggested only on objects with concrete types.
- It is more intuitive to modify an extension receiver than an argument.
- Compared to companion object or static methods, extensions are easier to find among hints since they are suggested on objects. For instance `"Text".isEmpty()` is easier to find than `TextUtils.isEmpty("Text")`. This is because when you place a dot after `"Text"`, you'll see as suggestions all the extension functions that can be applied to this object. To find `TextUtils.isEmpty`, you would need to guess where it is stored, and you might need to search through alternative util objects from different libraries.
- When we call a method, it is easy to confuse a top-level function with a method from the class or superclass, but their expected behavior is very different. Top-level extension functions do not have this problem because they must be invoked on an object.

## Summary

Do not repeat common algorithms. First, it is likely that there is a stdlib function that you can use instead. This is why it is good to learn the standard library. If you need a known algorithm that is not in the stdlib, or if you need a certain algorithm often, feel free to define it in your project. A good choice is to implement it as an extension function.

## Item 21: Use generics when implementing common algorithms

Just as we can pass a value to a function as an argument, we can pass a type as a type argument. Functions that accept type arguments (and thus have type parameters) are called generic functions<sup>33</sup>. A well-known example is the `filter` function from `stdlib`, which has type parameter `T`:

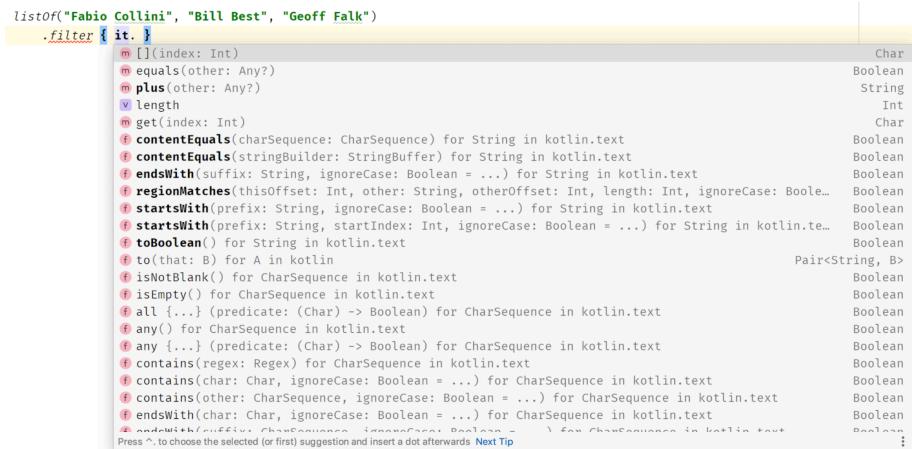
```
inline fun <T> Iterable<T>.filter(  
    predicate: (T) -> Boolean  
>: List<T> {  
    val destination = ArrayList<T>()  
    for (element in this) {  
        if (predicate(element)) {  
            destination.add(element)  
        }  
    }  
    return destination  
}
```

Type parameters are useful to the compiler since they allow it to check and correctly infer types a bit further, which makes our programs safer and programming more pleasurable for developers<sup>34</sup>. For instance, when we use `filter` inside a lambda expression, the compiler knows that an argument is of the same type as the type of elements in the collection, so it protects us from using something illegal, and the IDE can give us useful suggestions.

---

<sup>33</sup>For functions, we define type parameters in angle brackets between the `fun` keyword and the function name; for classes or interfaces, we define them in angle brackets after their name.

<sup>34</sup>Notice that all these benefits are for a programmer, not for the compiled program. Generics aren't that useful at runtime because they are generally erased during compilation due to JVM bytecode limitations (only reified types are not erased).



Generics were primarily introduced to classes and interfaces to allow the creation of collections with only concrete types, like `List<String>` or `Set<User>`. These types are lost during compilation, but when we are developing, the compiler forces us to pass only elements of the correct type. For instance, when we add `Int` to `MutableList<Int>`. Also, thanks to type arguments, the compiler knows that the returned type is `User` when we get an element from `Set<User>`. Thus, type parameters help us a lot in statically typed languages. Kotlin has powerful support for generics that is not well understood, and from my experience, even experienced Kotlin developers have gaps in their knowledge, especially about variance modifiers. So let's discuss the most important aspects of Kotlin generics in this and in Item 23: Consider using variance modifiers for generic types.

## Generic constraints

One important feature of type parameters is that they can be constrained to be a subtype of a concrete type. We set a constraint by placing the supertype after a colon. This type can include previous type parameters:

```
fun <T : Comparable<T>> Iterable<T>.sorted(): List<T> {
    /*...*/
}

fun <T, C : MutableCollection<in T>>
Iterable<T>.toCollection(destination: C): C {
    /*...*/
}

class ListAdapter<T: ItemAdapter>(//*...*/ { /*...*/ }
```

One important result of having a constraint is that instances of this type can use all the methods this type offers. Thus, when `T` is constrained as a subtype of `Iterable<Int>`, we know that we can iterate over an instance of type `T` and that elements returned by the iterator will be of type `Int`. When we constrain to `Comparable<T>`, we know that this type can be compared with itself. Another popular choice for a constraint is `Any`, which means that a type can be any non-null type:

```
inline fun <T, R : Any> Iterable<T>.mapNotNull(
    transform: (T) -> R?
): List<R> {
    return mapNotNullTo(ArrayList<R>(), transform)
}
```

In rare cases in which we might need to set more than one upper bound, we can use `where` to set more constraints:

```
fun <T: Animal> pet(animal: T) where T: GoodTempered {
    /*...*/
}

// OR

fun <T> pet(animal: T) where T: Animal, T: GoodTempered {
    /*...*/
}
```

## Summary

Type parameters are an important part of the Kotlin type system. We use them to have type-safe generic algorithms or generic objects. Type parameters can be constrained to be a subtype of a concrete type. When they are, we can safely use the methods offered by this type.

## Item 22: Avoid shadowing type parameters

It is possible to define properties and parameters with the same name due to shadowing. A local parameter can shadow an outer scope property. There is no warning because such a situation is not uncommon and is quite visible for developers:

```
class Forest(val name: String) {  
  
    fun addTree(name: String) {  
        // ...  
    }  
}
```

On the other hand, the same can happen when we shadow a class type parameter with a function type parameter. Such a situation is less visible and can lead to serious problems. This mistake is often made by developers who don't properly understand how generics work.

```
interface Tree  
class Birch: Tree  
class Spruce: Tree  
  
class Forest<T: Tree> {  
  
    fun <T: Tree> addTree(tree: T) {  
        // ...  
    }  
}
```

The problem is that the `Forest` and `addTree` type parameters are now independent of each other:

```
val forest = Forest<Birch>()  
forest.addTree(Birch())  
forest.addTree(Spruce())
```

Such a situation is rarely desired and might be confusing. One solution is that `addTree` could use the class type parameter `T`:

```
class Forest<T: Tree> {

    fun addTree(tree: T) {
        // ...
    }
}

// Usage
val forest = Forest<Birch>()
forest.addTree(Birch())
forest.addTree(Spruce()) // ERROR, type mismatch
```

If we need to introduce a new type parameter, it is better to name it differently. Note that it can be constrained to be a subtype of another type parameter:

```
class Forest<T: Tree> {

    fun <ST: T> addTree(tree: ST) {
        // ...
    }
}
```

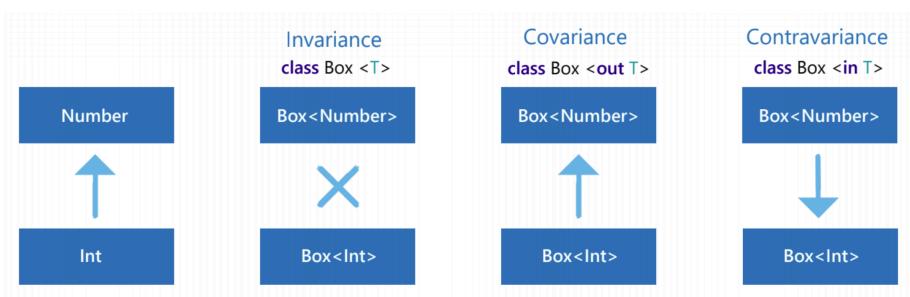
## Summary

Avoid shadowing type parameters, and be careful when you see that a type parameter is shadowed. Unlike for other kinds of parameters, this is not intuitive and might be highly confusing.

## Item 23: Consider using variance modifiers for generic types

Kotlin function types allows specifying variance modifiers: `in` and `out`. They are used to specify how a type parameter can be used. For instance, `in` means that a type parameter can only be used as an input, and `out` means that a type parameter can only be used as an output. In return, they give us more flexibility when using generic types. For instance `List` type parameter has `out` modifier, and thanks to that we can use `List<Int>` as a subtype of `List<Number>`<sup>35</sup>:

```
val ints: List<Int> = listOf(1, 2, 3)
val numbers: List<Number> = ints
```



Variance modifiers decide on relationship between generic types.

The simple heuristic is that if a type parameter is used only as an output (public result types), it should be marked as `out`, and if it is used only as an input (public parameter types), it should be marked as `in`. If it is used as both, it should not be marked with any variance modifier.

Variance modifiers are used in many Kotlin stdlib classes and interfaces.

---

<sup>35</sup>I provide a much more detailed explanation of variance modifiers, their limitations, patterns of usage, and the consequence of their usage in my other book *Advanced Kotlin*.

```
interface List<out E> : Collection<E> {
    /*...*/
}

fun interface ReadOnlyProperty<in T, out V> {
    operator fun getValue(
        thisRef: T,
        property: KProperty<*>
    ): V
}

interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

As you can see, those classes follow the heuristic presented above. Type parameter `T` is only used as an input, so it is marked as `in`. Type parameters `E` and `T` are only used as an output, so they are marked as `out`.

## Item 24: Reuse between different platforms by extracting common modules

Basics

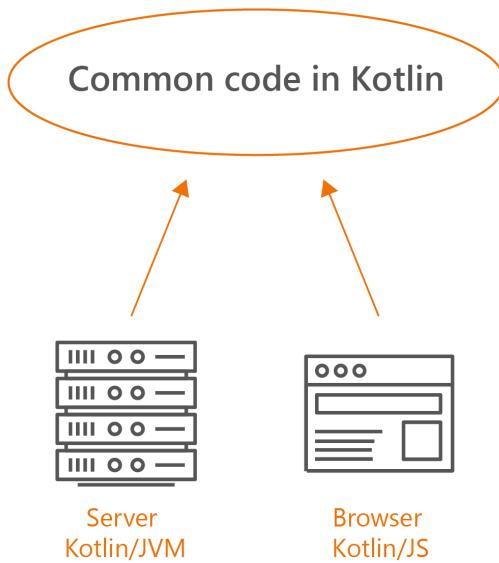
Companies rarely write applications only for a single platform<sup>36</sup>. They would rather develop a product for two or more platforms; nowadays, products often rely on several applications running on different platforms. Think of client and server applications communicating through network calls. As they need to communicate, there are often similarities that can be reused. Implementations of the same product for different platforms generally have even more similarities, especially their business logic, which is often nearly identical. These projects can profit significantly from sharing code.

### Full-stack development

Lots of companies are based on web development. Their product is a website, but in most cases these products need a backend application (also called server-side). On websites, JavaScript is king. It nearly has a monopoly on this platform. On the backend, a very popular option (if not the most popular) is Java. Since these languages are very different, it is common that backend and web development are separated. Things can change, however. Now Kotlin is becoming a popular alternative to Java for backend development. For instance, with Spring, the most popular Java framework, Kotlin is a first-class citizen. Kotlin can be used as an alternative to Java in every framework. There are also many Kotlin backend frameworks, such as Ktor. This is why many backend projects migrate from Java to Kotlin. A great thing about Kotlin is that it can also be compiled into JavaScript. There are already many Kotlin/JS libraries, and we can use Kotlin to write different kinds of web applications. For instance, we can write a web frontend using the React framework and Kotlin/JS. This allows us to write both the backend and the website all in Kotlin. Even better, **we can have parts that compile to both JVM bytecode and JavaScript**. These are shared parts where we can place, for instance, universal tools, API endpoint definitions, common abstractions, etc..

---

<sup>36</sup>In Kotlin, we view the JVM, Android, JavaScript, iOS, Linux, Windows, Mac and even embedded systems like STM32 as separate platforms.



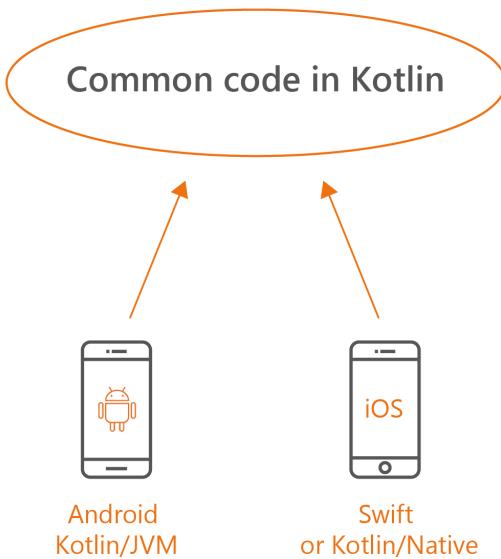
## Mobile development

This capability is even more important in the mobile world. We rarely build only for Android. Sometimes we can live without a server, but we generally need to implement an iOS application as well. Each application is written for a different platform using different languages and tools. In the end, Android and iOS versions of the same application are very similar. They are often designed differently, but they nearly always have the same logic inside. Using Kotlin's multiplatform capabilities, we can implement this logic only once and reuse it between these two platforms. We can make a common module and implement business logic there. Business logic should be independent of frameworks and platforms anyway (Clean Architecture). Such common logic can be written in pure Kotlin or using other common modules, and it can then be used on different platforms.

Common modules can be used directly in Android because both are built using Gradle. The experience is similar to having these common parts in our Android project.

For iOS, we compile these common parts to an Objective-C framework using

Kotlin/Native, which is compiled into native code<sup>37</sup> using LLVM<sup>38</sup>. We can then use the resulting code from Swift in Xcode or AppCode. Alternatively, we can implement our whole application using Kotlin/Native.



## Libraries

Defining common modules is also a powerful tool for libraries. In particular, libraries that are not highly platform-dependent can easily be moved to a common module, thus developers can use them from all languages running on the JVM, or JavaScript, or natively (so from Java, Scala, JavaScript, CoffeeScript, TypeScript, C, Objective-C, Swift, Python, C#, etc.).

## All together

We can use all these platforms together. Using Kotlin, we can develop for nearly all kinds of popular devices and platforms, and code can be reused between them

---

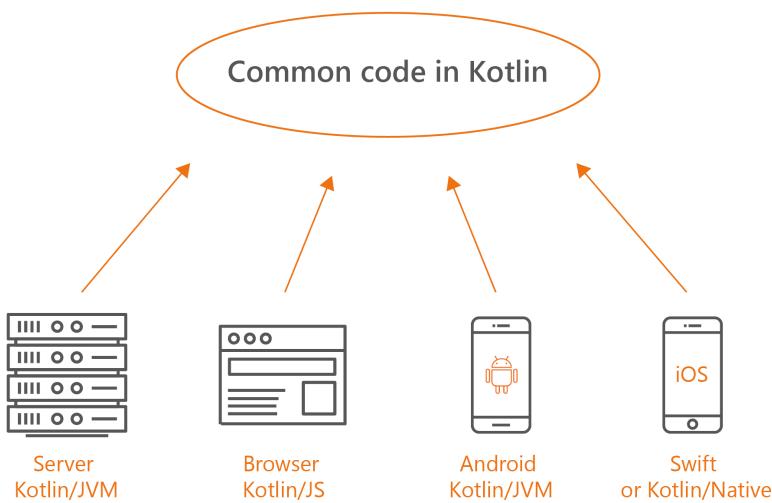
<sup>37</sup>Native code is code that is written to run on a specific processor. Languages like C, C++, Swift, Kotlin/Native are native because they are compiled into machine code for each processor they need to run on.

<sup>38</sup>Like Swift or Rust.

however we want. Here are just a few examples of what we can write in Kotlin:

- Backend in Kotlin/JVM, for instance on Spring or Ktor
- Website in Kotlin/JS, for instance in React
- Android in Kotlin/JVM, using the Android SDK
- iOS Frameworks that can be used from Objective-C or Swift using Kotlin/-Native
- Desktop applications in Kotlin/JVM, for instance in TornadoFX
- Raspberry Pi, Linux or macOS programs in Kotlin/Native

Here is a typical application visualized:



We are still learning how to organize our code to make code reuse safe and efficient in the long run, but it is good to know the possibilities this approach gives us. We can reuse between different platforms using common modules. This is a powerful way to eliminate redundancy and to reuse common logic and common algorithms.

## Summary

Kotlin multiplatform capabilities are a powerful tool for sharing code between different platforms and reusing it in different languages. There are many ways

how developers can benefit from this feature<sup>39</sup>.

---

<sup>39</sup>This item was just a brief presentation of Kotlin Multiplatform possibilities. If you look for details, see my other book *Advanced Kotlin*.

# Chapter 4: Abstraction design

Abstraction is one of the most important concepts in the programming world. In OOP (Object-Oriented Programming), abstraction is one of the three core concepts (along with encapsulation and inheritance). In the functional programming community, it is common to say that all we do in programming is abstraction and composition<sup>40</sup>. As you can see, we treat abstraction seriously. But what is an abstraction? The definition I find most useful comes from Wikipedia:

Abstraction is a process or result of generalization, removal of properties, or distancing of ideas from objects.

[https://en.wikipedia.org/wiki/Abstraction\\_\(disambiguation\)](https://en.wikipedia.org/wiki/Abstraction_(disambiguation))

In other words, by abstraction we mean a form of simplification used to hide complexity. A fundamental example in programming is interfaces, which are abstractions of classes because they express only a subset of traits. Concretely, abstraction is a set of methods and properties.



Reality



Abstraction

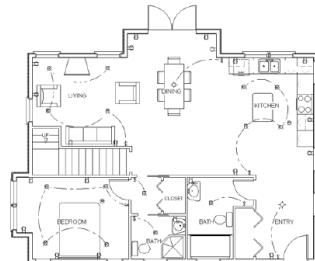
There is no single abstraction for every instance. There are many. In terms of objects, a class can be expressed by many interfaces or by multiple superclasses. A key feature of abstraction is that it decides what should be hidden and what should be exposed.

---

<sup>40</sup>Category Theory for Programmers by Bartosz Milewski.



Reality



Abstraction

## Abstraction in programming

We often forget how abstract everything we do in programming is. When we type a number, it is easy to forget that it is actually represented by zeros and ones. When we type a String, it is easy to forget that it is a complex object in which each character is represented by a defined charset, like UTF-8.

Designing abstractions is not only about separating modules or libraries. Whenever you define a function, you hide its implementation behind this function's signature. This is an abstraction!

Let's do a thought experiment: what if it wasn't possible to define a `maxOf` method that returns the biggest of two numbers?

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

Of course, we could get along without ever defining this function by always writing the full expression and never mentioning `maxOf` explicitly:

```
val biggest = if (x > y) x else y
```

```
val height =
  if (minHeight > calculatedHeight) minHeight
  else calculatedHeight
```

However, this would place us at a serious disadvantage. It would force us to always work at the level of the particular operations that happen to be primitives in the language (comparison, in this case) rather than in terms of higher-level operations. Our programs would be able to compute which number is bigger, but

our language would lack the ability to express the concept of choosing the bigger number.

This problem is not abstract at all. Until version 8, Java lacked the capability to easily express mapping on a list. Instead, we had to use repeatable structures to express this concept:

```
// Java
List<String> names = new ArrayList<>();
for (User user : users) {
    names.add(user.getName());
}
```

In Kotlin, since the beginning we have been able to express this using a simple function:

```
val names = users.map { it.name }
```

Lazy property initialization patterns still cannot be expressed in Java. In Kotlin, we use a property delegate to express this concept:

```
val connection by lazy { makeConnection() }
```

Who knows how many other concepts there are that we do not know how to extract and express directly?

One of the features we should expect from a powerful programming language is the ability to build abstractions by assigning names to common patterns<sup>41</sup>. In one of the most rudimentary forms, this is what we achieve by extracting functions, delegates, classes, etc. As a result, we can then work directly in terms of the abstractions.

## Car metaphor

Many things happen when you drive a car. It requires the coordinated work of the engine, alternator, suspension and many other elements. Just imagine how hard driving a car would be if it required understanding and following each of these elements in real time! Thankfully, it doesn't. As a driver, all we need to know is how to use a car's interface – the steering wheel, gear shifter, and pedals – to operate the vehicle. Anything under the hood can change. A mechanic can

---

<sup>41</sup>Structure and Interpretation of Computer Programs by Hal Abelson and Gerald Jay Sussman with Julie Sussman.

change from petrol to natural gas and then diesel without us even knowing about it. As cars introduce more and more electronic elements and special systems, the interface mostly remains the same. With such changes under the hood, the car's performance would likely also change, but we are able to operate it regardless.

A car has a well-defined interface. Despite all the complex components, it is simple to use. The steering wheel represents an abstraction for left-right direction change; the gear shifter is an abstraction for forward-backward direction change; the gas pedal is an abstraction for acceleration; and the brake an abstraction of deceleration. These are all we need in an automobile. These are abstractions that hide all the magic happening under the hood. Thanks to that, users do not need to know anything about their car's engineering. They only need to understand how to drive it. Similarly, creators or car enthusiasts can change everything in a car, and this is fine as long as the driving stays the same. Remember this metaphor as we will refer to it throughout this chapter.

Similarly, in programming, we use abstractions mainly to:

- Hide complexity
- Organize our code
- Give creators the freedom to change

The first reason was already described in *Chapter 3: Reusability*, and I assume that it is clear at this point why it is important to extract functions, classes or delegates to reuse common logic or common algorithms. In *Item 25: Each function should be written in terms of a single level of abstraction*, we will see how to use abstractions to organize code. In *Item 26: Use abstraction to protect code against changes*, we will see how to use abstractions to give ourselves the freedom to change things. Then, we will spend the rest of this chapter on creating and using abstractions.

This is a pretty high-level chapter, so the rules presented here are a bit more abstract. After this chapter, in *Chapter 5: Object creation* and *Chapter 6: Class design*, we will cover some more concrete aspects of OOP design. These chapters will dive into deeper aspects of class implementation and use, but they will both build on this chapter.

## Item 25: Each function should be written in terms of a single level of abstraction

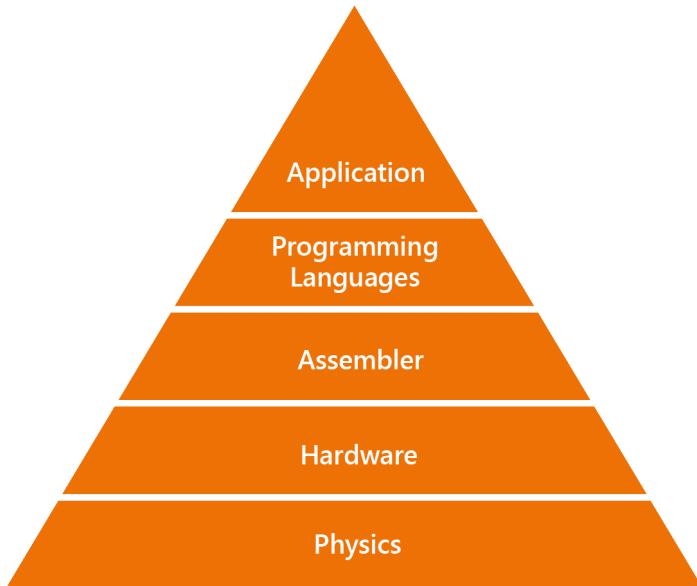
Not Kotlin-specific

Basics

A computer is an extremely complex device, but we can work with it thanks to the fact that its complexity is split into different elements in distinct layers.

From a programmer's perspective, the lowest abstraction layer of a computer is hardware. Going up from there, since we generally write code for processors, the next interesting layer is the processor control commands (machine code instructions). For readability, they are expressed in a very simple language that is one-to-one translated into those commands. This language is called Assembly. Programming in Assembly language is difficult, so building modern applications in this way is absolutely unthinkable. To simplify programming, software engineers introduced compilers: software that translates one language into another (generally a lower-level one). The first compilers were written in Assembly language, and they translated code written as text into Assembly instructions. This is how the first higher-level languages were created. They were in turn used to write compilers for better languages, thus introducing C, C++ and other high-level languages that are used to write programs and applications. Later, the concepts of abstract machines and interpreted languages were invented; it is hard to place languages like Java or JavaScript in this pyramid, but the general notion of abstraction layers remained as an idea.

The big advantage of having well-separated layers is that operating on a specific layer means that lower levels can be relied upon to work as expected, thus removing the need to fully understand the details. We can program without knowing anything about assembler or JVM bytecode. This is very convenient. Similarly, when assembler or JVM bytecode needs to change, programmers don't need to worry about making changes to applications as long as they only adjust the upper layer, which is what native languages or Java are compiled to. Programmers operate on a single layer, often building for upper layers. This is all developers need to know and it is very convenient.



## Level of abstraction

As you can see, layers have been built upon layers in computer science. This is why computer scientists started distinguishing how high-level something is. The higher the level, the further it is from physics. In programming, we say that the higher the level, the further the code is from the processor. The higher the level, the fewer details we need to worry about, but you're trading this simplicity for a lack of control. In C, memory management is an important part of your job. In Java, the Garbage Collector handles this automatically for you, but optimizing memory usage is much harder.

## The Single Level of Abstraction principle

Just as computer science problems are extracted into separate layers, we can create abstractions in our code as well. The most basic tool we use for that is a function. Also, the same as in computers, we prefer to operate on a single level of abstraction at a time. This is why the programming community developed the “Single Level of Abstraction” principle, which states that **Each function should be written in terms of a single level of abstraction**.

Imagine that you need to create a class to represent a coffee machine with a single button. Making coffee is a complex operation that needs many different

parts of a coffee machine. We'll represent it by a class with a single function named `makeCoffee`. We can definitely implement all the necessary logic inside this unique function:

```
class CoffeeMachine {  
  
    fun makeCoffee() {  
        // Declarations of hundreds of variables  
        // Complex logic to coordinate everything  
        // with many low-level optimizations  
    }  
}
```

This function could have hundreds of lines. Believe me, I've seen such things, especially in old programs. Such functions are absolutely unreadable. It would be really hard to understand the general behavior of the function because, when we read it, we would constantly focus on the details. It would also be hard to find anything. Just imagine that you are asked to make a small modification, such as modify the temperature of the water; to do this, you would probably need to understand the whole function, which would be absurdly hard. Our memory capacity is limited and we do not want a programmer to waste time on unnecessary details. This is why it is better to extract high-level steps as separate functions:

```
class CoffeeMachine {  
  
    fun makeCoffee() {  
        boilWater()  
        brewCoffee()  
        pourCoffee()  
        pourMilk()  
    }  
  
    private fun boilWater() {  
        // ...  
    }  
  
    private fun brewCoffee() {  
        // ...  
    }  
  
    private fun pourCoffee() {  
        // ...  
    }  
}
```

```
// ...
}

private fun pourMilk() {
    // ...
}

}
```

Now you can clearly see what the general flow of this function is. These private functions are just like chapters in a book. Thanks to that, if you need to change something, you can jump directly to where it is implemented. We have extracted the higher-level procedures, which has greatly simplified the comprehension of the first procedure. We have made it readable; if someone wants to understand it at a lower level, they can just jump there and read it. By extracting very simple abstractions, we have improved readability.

Following this rule, all these new functions should be just as simple. This is a general rule: functions should be small and have a minimal number of responsibilities<sup>42</sup>. If one of these functions is too complex, we should extract intermediary abstractions<sup>43</sup>. As a result, we should end up with many small and readable functions, all located at a single level of abstraction. At every level of abstraction, we operate on abstract terms (methods and classes); if you want to clarify them, you can always jump into their definition<sup>44</sup>. This way, we lose nothing from extracting these functions, and our code is more readable.

An additional bonus is that functions extracted this way are easier to reuse and test. Say that we now need to make a separate function to produce espresso coffee, which does not contain milk. When the parts of the process are extracted, we can now reuse them easily:

```
fun makeEspressoCoffee() {
    boilWater()
    brewCoffee()
    pourCoffee()
}
```

---

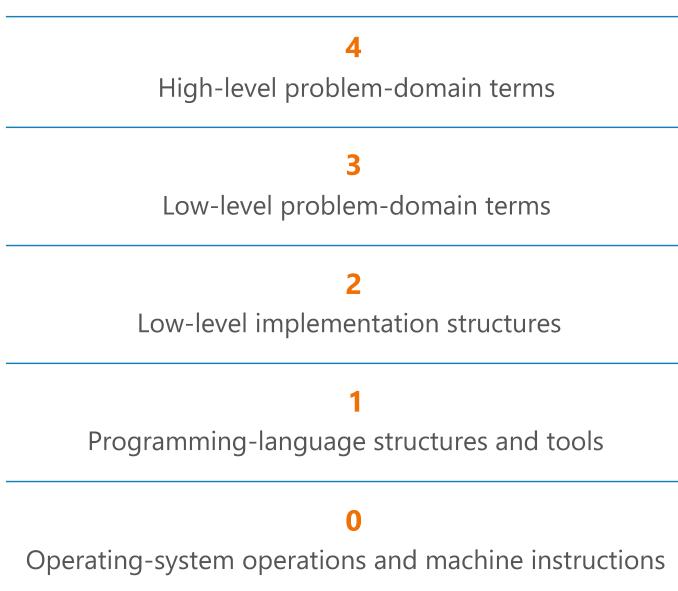
<sup>42</sup>Clean Code by Robert Cecil Martin.

<sup>43</sup>These might be functions, as well as classes or other kinds of abstractions. The differences will be shown in the next item, Item 26: Use abstraction to protect code against changes.

<sup>44</sup>In IntelliJ or Android Studio, we jump to element definition by holding the Ctrl key (Command on Mac) and clicking on the element name.

## Abstraction levels in program architecture

The notion of layers of abstractions is also applicable to levels that are higher than functions. We separate abstraction to hide the details of a subsystem, thus allowing the separation of concerns (SoC) to facilitate interoperability and platform independence. This means defining higher levels in problem-domain terms<sup>45</sup>.



This notion is also important when we design modular systems. Separate modules can hide layer-specific elements. When we write applications, the general understanding is that modules that represent inputs or outputs (views in the frontend, HTTP request handlers on the backend) are lower-layer modules. On the other hand, those representing use cases and business logic are higher-level layers<sup>46</sup>.

We say that projects with well-separated layers are stratified. In a well-stratified project, one can view the system at any single level and get a consistent view<sup>47</sup>.

<sup>45</sup>Code Complete by Steve McConnell, 2nd Edition, Section 34.6.

<sup>46</sup>Clean Architecture: A Craftsman's Guide to Software Structure and Design by Robert C. Martin, 1st Edition.

<sup>47</sup>Code Complete by Steve McConnell, 2nd Edition, Section 5.2.

Stratification is generally desired in programs.

## Summary

Making separate abstraction layers is a popular concept in programming. It helps us organize knowledge and hide the details of the subsystem, thus allowing the separation of concerns in order to facilitate interoperability and platform independence. We separate abstractions in many ways, like functions, classes, and modules. We should try not to make any of these layers too big. Smaller abstractions operating on a single layer are easier to understand. The general notion of abstraction level is that the closer it is to concrete actions, processor or input/output, the lower level it is. In a lower abstraction layers, we define a language of terms (API) for a higher layer or layers.

## Item 26: Use abstraction to protect code against changes

Not Kotlin-specific

Basics

*Walking on water and developing software from a specification are easy if both are frozen*

– Edward V Berard ; Essays on object-oriented software engineering,  
p. 46

When we hide actual code behind abstractions like functions or classes, we not only protect users from these details, but we also give ourselves the freedom to change this code later, often without users even being aware of it. For instance, when you extract a sorting algorithm into a function, you can later optimize its performance without changing the way it is used.

Returning to the car metaphor mentioned previously, car manufacturers and mechanics can change everything under the hood of a car, and as long as the operation remains the same, a user won't notice. This gives manufacturers the freedom to make more environmentally friendly cars or to add more sensors to make them safer.

In this item, we will see how different kinds of abstractions give us freedom by protecting us from a variety of changes. We'll examine three practical cases, then we'll discuss finding a balance in terms of the number of abstractions we create. Let's start with the simplest kind of abstraction: constant value.

### Constant

Literal constant values are rarely self-explanatory and are especially problematic when they repeat in our code. Moving these values into constant properties not only assigns a meaningful name to them, it also helps us better manage the values of these constants when they need to be changed. Let's see a simple example with password validation:

```
fun isPasswordValid(text: String): Boolean {  
    if(text.length < 7) return false  
    //...  
}
```

The number 7 can be understood on the basis of the context, but it would be easier if it were extracted into a constant:

```
const val MIN_PASSWORD_LENGTH = 7

fun isPasswordValid(text: String): Boolean {
    if(text.length < MIN_PASSWORD_LENGTH) return false
    //...
}
```

With that, it is easier to modify the minimum password size. We don't need to understand the validation logic; instead, we can just change this constant. This is why it is especially important to extract values that are used more than once. For instance, the maximum number of threads that can connect to our database at the same time:

```
val MAX_THREADS = 10
```

Once this value has been extracted, you can easily change it whenever you need. Just imagine how hard it would be to change it if this number was spread all over the project.

As you can see, extracting a constant:

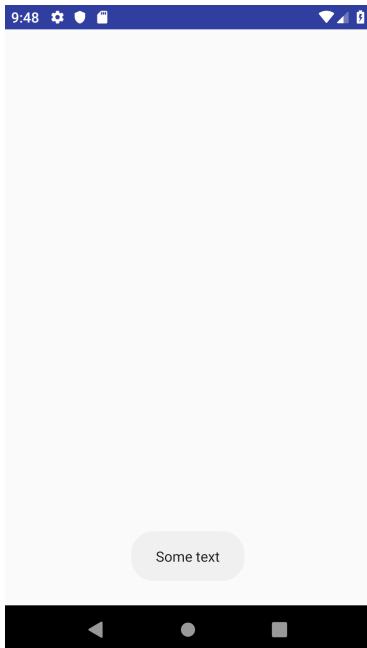
- Names it
- Helps us change its value in the future

We will also see similar results for different kinds of abstractions.

## Functions

Imagine that you are developing an application and you notice that you often need to display a toast message to users. This is how you do it programmatically:

```
Toast.makeText(this, message, Toast.LENGTH_LONG).show()
```



Toast message in Android

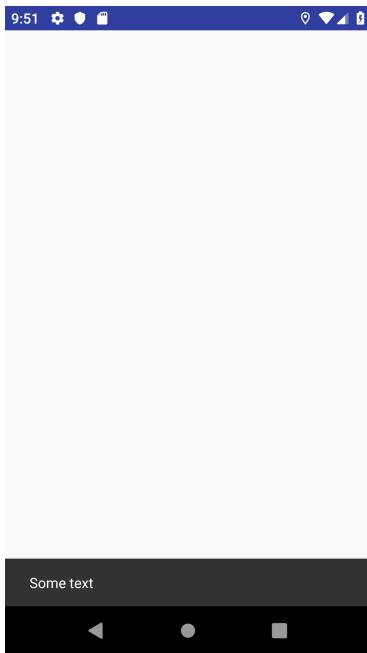
We can extract this common algorithm into a simple extension function that displays toast messages:

```
fun Context.toast(  
    message: String,  
    duration: Int = Toast.LENGTH_LONG  
) {  
    Toast.makeText(this, message, duration).show()  
}  
  
// Usage  
context.toast(message)  
  
// Usage in Activity or subclasses of Context  
toast(message)
```

This change helps us extract a common algorithm so that we don't need to remember how to display a toast every time. It would also help if the way to display a toast, in general, changed (which is rather unlikely), but there are still other kinds of changes we are not prepared for.

What if we had to change the way we display messages to users from toasts to snackbars (a different kind of message display)? A simple answer is that by having extracted this functionality, we can just change the implementation inside this function and rename it.

```
fun Context.snackbar(  
    message: String,  
    length: Int = Toast.LENGTH_LONG  
) {  
    //...  
}
```



Snackbar message in Android

This solution is far from perfect. First of all, renaming the function might be dangerous even if it is used only internally<sup>48</sup>, especially if other modules depend on this function. The next problem is that parameters cannot be automatically changed so easily, thus we are still stuck with the toast API to declare the message

---

<sup>48</sup>When a function is a part of an external API, we cannot easily adjust calls and so we are stuck with the old name for at least some time (Item 27: Specify API stability).

duration. This is very problematic. When we display a Snackbar, we should not depend on a field from `Toast`. On the other hand, changing all usages to use the Snackbar's enum would also be problematic:

```
fun Context.snackbar(
    message: String,
    duration: Int = Snackbar.LENGTH_LONG
) {
    //...
}
```

When we know that the way the message is displayed might change, we know that what is really important is not how this message is displayed but that we want to be able to display messages to users. What we need is a more abstract method to display a message. Having that in mind, a programmer could hide the toast display behind a higher-level function `showMessage`, which would be independent of the concept of toast:

```
fun Context.showMessage(
    message: String,
    duration: MessageLength = MessageLength.LONG
) {
    val toastDuration = when(duration) {
        SHORT -> Toast.LENGTH_SHORT
        LONG -> Toast.LENGTH_LONG
    }
    Toast.makeText(this, message, toastDuration).show()
}

enum class MessageLength { SHORT, LONG }
```

The biggest change here is the name. Some developers might neglect the importance of this change and say that a name is just a label, which doesn't matter. This perspective is valid from the compiler's point of view, but not from a developer's point of view. A function represents an abstraction, and the signature of this function informs us what abstraction it is. A meaningful name is very important.

A function is a very simple abstraction, but it is also very limited. A function does not hold a state. Changes in a function signature often influence all usages. A more powerful way to abstract away implementation is by using classes.

## Classes

Here is how we can abstract displaying messages into a class:

```
class MessageDisplay(val context: Context) {  
  
    fun show(  
        message: String,  
        duration: Length = Length.LONG  
    ) {  
        val toastDuration = when(duration) {  
            SHORT -> Toast.LENGTH_SHORT  
            LONG -> Toast.LENGTH_LONG  
        }  
        Toast.makeText(context, message, toastDuration)  
            .show()  
    }  
  
    enum class Length { SHORT, LONG }  
}  
  
// Usage  
val messageDisplay = MessageDisplay(context)  
messageDisplay.show("Message")
```

The key reason why classes are more powerful than functions is that they can hold a state and expose many functions (class member functions are called methods). In this case, we have a `context` in the class state, and it is injected via the constructor. By using a dependency injection framework, we can delegate the class creation:

```
@Inject  
lateinit var messageDisplay: MessageDisplay
```

Additionally, we can mock a class to test the functionality of other classes that depend on it.

```
val messageDisplay: MessageDisplay = mockk()
```

Furthermore, one could add more methods to set up the message display:

```
messageDisplay.setChristmasMode(true)
```

As you can see, classes give us more freedom, but they still have limitations. For instance, when a class is final, we know what exact implementation is under its

type. We have a bit more freedom with open classes because we could serve a subclass instead. This abstraction is still strongly bound to this class though. To get more freedom we can make our implementation even more abstract and hide this class behind an interface.

## Interfaces

When reading the Kotlin standard library, you might notice that nearly everything is represented as an interface. Just take a look at a few examples:

- The `listOf` function returns `List`, which is an interface. This is similar to other factory methods (we will explain them in Item 32: Consider factory functions instead of secondary constructors).
- Collection processing functions are extension functions on `Iterable` or `Collection`, and they return `List`, `Map`, etc. These are all interfaces.
- Property delegates are hidden behind `ReadOnlyProperty` or `ReadWriteProperty`, which are also interfaces. Actual classes are often private. The `lazy` function also declares the `Lazy` interface as its return type.

It is common practice for library creators to restrict the visibility of inner classes and expose them from behind interfaces, and there are good reasons for that. This way, library creators are sure that users do not use these classes directly, so they can change their implementations without any worries, as long as the interfaces stay the same. This is exactly the idea behind this item: by hiding objects behind an interface, we abstract away any actual implementation and we force users to depend only on this abstraction. This way, we reduce coupling.

In Kotlin, there is another reason behind returning interfaces instead of classes: Kotlin is a multiplatform language and the same `listOf` returns different list implementations for Kotlin/JVM, Kotlin/JS, and Kotlin/Native. This is an optimization as Kotlin generally uses platform-specific native collections, which is fine because they all respect the `List` interface.

Let's see how we can apply this idea to our message display. This is how it could look when we hide our class behind an interface:

```
interface MessageDisplay {
    fun show(
        message: String,
        duration: Length = LONG
    )
}

class ToastDisplay(val context: Context): MessageDisplay {

    override fun show(
        message: String,
        duration: Length
    ) {
        val toastDuration = when(duration) {
            SHORT -> Toast.LENGTH_SHORT
            LONG -> Toast.LENGTH_LONG
        }
        Toast.makeText(context, message, toastDuration)
            .show()
    }

    enum class Length { SHORT, LONG }
}
}
```

In return, we've got more freedom. For instance, we can inject the class that displays toasts on tablets and snackbars on phones. One might also use `MessageDisplay` in a common module shared between Android, iOS, and Web. Then, we could have a different implementation for each platform. For instance, on iOS and Web, it could display an alert.

Another benefit is that interface faking for testing is simpler than class mocking, and it does not need a mocking library:

```
val messageDisplay: MessageDisplay = TestMessageDisplay()
```

Finally, the declaration is more decoupled from the usage, so we have more freedom in changing actual classes like `ToastDisplay`. On the other hand, if we want to change the way it is used, we would need to change the `MessageDisplay` interface and all the classes that implement it.

## Next ID

Let's discuss one more example. Let's say that we need a unique ID in our project. A very simple way is to have a top-level property to hold the next ID and increment

it whenever we need a new ID:

```
var nextId: Int = 0

// Usage
val newId = nextId++
```

Seeing such usage spread around our code should cause some alerts. What if we wanted to change the way IDs are created? Let's be honest, this way is far from perfect:

- We start at 0 whenever we cold-start our program.
- It is not thread-safe.

If we accept this solution for now, we should protect ourselves from change by extracting ID creation into a function:

```
private var nextId: Int = 0
fun getNextId(): Int = nextId++

// Usage
val newId = getNextId()
```

Notice this solution only protects us from the need to change the way how ID is created. There are many changes that we are still prone to, the biggest of which is a change of ID type. What if one day we need to store ID as a `String`? Also notice that someone who sees that ID is represented as an `Int` might use some type-dependent operations. For instance, use comparison to check which ID is older. Such assumptions might lead to serious problems. To prevent this and to make it easy to change ID type in the future, we might extract ID as a class:

```
data class Id(private val id: Int)

private var nextId: Int = 0
fun getNextId(): Id = Id(nextId++)
```

Once again, it is clear that more abstractions give us more freedom, but they also make definitions and their usage harder to define and understand.

## Abstractions give freedom

We've presented a few common ways to introduce abstraction:

- Extracting constants
- Wrapping behaviors into functions
- Wrapping functions into classes
- Hiding classes behind interfaces
- Wrapping universal types into context-specific types

We've shown how each of these give us different kinds of freedom. Notice that there are many more options available, such as:

- Using generic type parameters
- Extracting inner classes
- Restricting creation, for instance by forcing object creation via factory methods<sup>49</sup>

On the other hand, there is a dark side to abstractions. They give us freedom and separate code, but they can often make code harder to understand and modify. Let's talk about problems with abstractions.

## Problems with abstraction

Adding new abstractions requires readers of our code to learn or already be familiar with specific concepts. When we define another abstraction, it is another thing that needs to be understood in our project. Of course, this is less of a problem when we restrict the visibility of our abstractions (Item 29: Minimize elements' visibility) or when we define abstractions that are used only for concrete tasks. This is why modularity is so important in bigger projects. We need to understand that defining abstractions incurs this cost, therefore we should not abstract everything by default.

We can infinitely extract abstractions, but this will soon do more harm than good. This fact was parodied in the FizzBuzz Enterprise Edition project<sup>50</sup>, where the authors showed that even for such a simple problem as Fizz Buzz<sup>51</sup>, one can extract a ridiculous amount of abstractions, which ends up making this solution extremely hard to comprehend and work on. At the time of writing this book, in

---

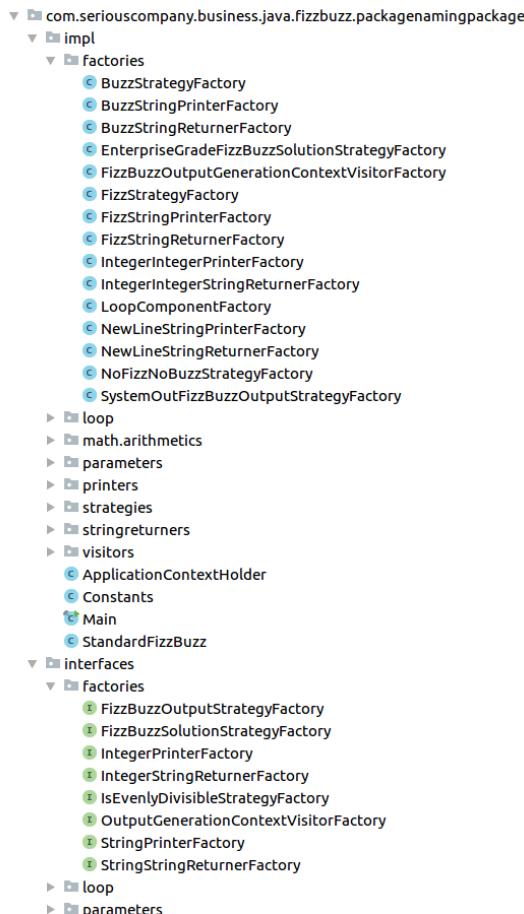
<sup>49</sup>More about this in Chapter 4: Object creation.

<sup>50</sup>[github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition](https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition)

<sup>51</sup>The problem is defined as: For numbers 1 through 100, if a number is divisible by 3, print Fizz; if a number is divisible by 5, print Buzz; if a number is divisible by both 3 and 5 (e.g., 15), print FizzBuzz; in all other cases, print the number.

the project, there are 61 classes and 26 interfaces. All that, to solve a problem that generally requires less than 10 lines of code. Sure, applying changes at any level is easy, but understanding what this code does and how it does it is extremely hard.

Abstractions can hide a lot. **On the one hand, it is easier to do development when there is less to think about; on the other hand, it becomes harder to understand the consequences of our actions when we use too many abstractions.** You might use the `showMessage` function and think that it still displays toast, but you might be surprised when it displays a Snackbar. When you see that an unintended toast message is displayed, you might look for `Toast.makeText` and have problems finding it because it is displayed using `showMessage`. Having too many abstractions makes it harder to understand our code. It can also make us anxious when we are not sure what the consequences of our actions are.



Part of the FizzBuzz Enterprise Edition structure of classes. In the description of this project, you can find the following sarcastic rationale: “This project is an example of how the popular FizzBuzz game might be built were it subject to the high-quality standards of enterprise software.”

To understand abstractions, examples are very helpful. Abstractions are made more real for us by unit tests or examples in the documentation that show how an element can be used. For the same reason, I filled this book with concrete examples for most ideas I present. It is hard to understand abstract descriptions, and it is also easy to misunderstand them.

## How to find a balance?

The rule of thumb is: every level of complexity gives us more freedom and organizes our code, but also makes it harder to understand what is really going

on in our project. Both extremes are bad. The best solution is always somewhere in the middle, but where exactly this is depends on many factors, like:

- Team size
- Team experience
- Project size
- Feature set
- Domain knowledge

We are constantly looking for balance in every project. Finding a proper balance is almost an art as it requires intuition gained over hundreds if not thousands of hours architecting and coding projects. Here are a few suggestions I can give:

- In bigger projects with more developers, it is much harder to change object creation and usage later, so we prefer more abstract solutions. Also, separation between modules or parts is especially useful in such projects.
- We care less about how difficult creation is when we use a dependency injection framework because we probably only need to define this creation once anyway.
- Testing or making different application variants might require us to use some abstractions.
- When your project is small and experimental, you can enjoy your freedom to directly make changes without the necessity of dealing with abstractions. However, when your project gets serious, organize it as soon as possible.

Another thing that we need to constantly think about is what might change and what are the odds of each change. For instance, there is only a very small chance that the API for the toast display will change, but there is a reasonable probability that we will need to change the way we display a message. Is there a chance we might need to mock this mechanism? Is there a chance that one day you will need a more generic mechanism or a mechanism that might be platform-independent? These probabilities are not 0, so how big are they? Observing how things change over the years gives us better and better intuition.

## Summary

Abstractions are not only to eliminate redundancy and to organize our code: they also help us when we need to change our code. Although using abstractions makes our code harder to understand. Abstractions are something we need to learn and understand. It is also harder to understand the consequences of using abstract structures. That is why we need to understand both the importance and risk of using abstractions, and we need to search for a balance in every project, as having too many or too few abstractions is not ideal.

## Item 27: Specify API stability

Life would be much harder if every car were totally different to drive. There are some elements in cars that are not universal, like the way we preset radio stations, and I often see car owners having trouble using them. We are too lazy to learn meaningless and temporary interfaces. We prefer stable and universal ones.

Similarly, in programming we much prefer stable and possibly standard Application Programming Interfaces (API). The main reasons are:

1. **When an API changes and developers get the update, they will need to manually update their code.** This can be especially problematic when many elements depend on this API. Fixing its use or providing an alternative might be hard, especially if our API has been used by another developer in part of our project that we're not familiar with. If it is a public library, we cannot adjust these uses ourselves; instead, our users have to make the changes. From a user's perspective, this isn't a convenient situation. Small changes in a library might require many changes in different parts of the codebase. When users are afraid of such changes, they continue using older library versions. This is a big problem because updating becomes harder and harder for them, and new updates might have things they need, like bug fixes or vulnerability corrections. Older libraries may no longer be supported or might stop working entirely. It is a very unhealthy situation when programmers are afraid to use newer stable releases of libraries.
2. **Users need to learn a new API.** This is additional energy users are generally unwilling to expend. What's more, **they need to update knowledge that has changed.** This is also painful for them, so they avoid it. It's not healthy either: outdated knowledge can lead to security issues and learning what changes were made in those libraries the hard way.

On the other hand, designing a good API is very hard, so creators often want to make changes to improve it. The solution that we (the programming community) developed is that we specify API stability.

The simplest way to specify API stability is that creators should specify in the documentation that some parts of an API are unstable. More formally, we specify the stability of the whole library or module using versions. There are many versioning systems, though there is one that is now so popular it can be treated nearly like a standard. It is Semantic Versioning (SemVer): in this system, we compose the version number from 3 parts: MAJOR.MINOR.PATCH. Each of those parts is a positive integer starting from 0, and we increment each of them when changes in the public API have concrete importance. So we increment:

- MAJOR version when you make incompatible API changes.

- MINOR version when you add functionality in a backward-compatible manner.
- PATCH version when you make backward-compatible bug fixes.

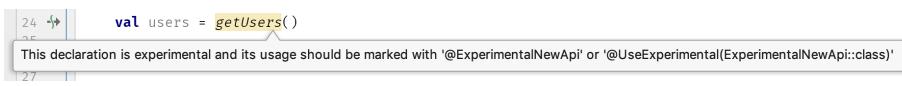
When we increment MAJOR, we set MINOR and PATCH to 0. When we increment MINOR we set PATCH to 0. Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format. Major version zero (0.y.z) is for initial development; with this version, anything may change at any time, and the public API should not be considered stable. Therefore, when a library or module follows SemVer and has MAJOR version 0, we should not expect it to be stable.

Do not worry about staying in beta for a long time. It took over 5 years for Kotlin to reach version 1.0. This was a very important time for this language since it changed a lot in this period.

When we introduce new elements into a stable API but they are not yet stable, we should first keep them for some time in another branch. When you want to allow some users to use this API (by merging code into the main branch and releasing it), you can use the `Experimental` meta-annotation to warn them that it is not yet stable. This makes elements visible, but using them displays a warning or an error (depending on the set `level` annotation property).

```
@Experimental(level = Experimental.Level.WARNING)
annotation class ExperimentalNewApi

@ExperimentalNewApi
suspend fun getUsers(): List<User> {
    //...
}
```



The screenshot shows a code editor with the following code:

```
24 ↗
  val users = getUsers()
This declaration is experimental and its usage should be marked with '@ExperimentalNewApi' or '@UseExperimental(ExperimentalNewApi::class)'
```

A tooltip at the bottom of the editor window states: "This declaration is experimental and its usage should be marked with '@ExperimentalNewApi' or '@UseExperimental(ExperimentalNewApi::class)'".

We should expect that such elements might change at any moment. Again, don't worry about keeping elements experimental for a long time. It might slow down adoption, but it gives us more time to design a good API.

When we need to change something that is part of a stable API, we initially annotate this element as `Deprecated` in order to help users deal with this transition:

```
@Deprecated("Use suspending getUsers instead")
fun getUsers(callback: (List<User>)->Unit) {
    //...
}
```

Also, when there is a direct alternative to the old function, specify it using ReplaceWith annotation, to allow the IDE to make an automatic transition:

```
@Deprecated("Use suspending getUsers instead",
ReplaceWith("getUsers()"))
fun getUsers(callback: (List<User>)->Unit) {
    //...
}
```

An example from the stdlib:

```
@Deprecated("Use readBytes() overload without "+
"estimatedSize parameter",
ReplaceWith("readBytes()"))
public fun InputStream.readBytes(
    estimatedSize: Int = DEFAULT_BUFFER_SIZE
): ByteArray {
    //...
}
```

Then we need to give users time to adjust. This should be a long time because users have responsibilities other than adjusting to new versions of libraries they use. In widely used APIs, this takes years. Finally, in a major release, we can remove the deprecated element.

## Summary

Users need to know about API stability. While a stable API is preferred, there is nothing worse than unexpected changes in an API that is supposed to be stable. Such changes can be really painful for users. Correct communication between module or library creators and their users is important and is achieved by using version names, documentation, and annotations. Also, each change in a stable API needs to follow a long process of deprecation.

## Item 28: Consider wrapping external APIs

Not Kotlin-specific

It is risky to heavily use an API that might be unstable, both when its creators clarify that it is unstable, and when we do not trust these creators to keep it stable. We should remember that we need to adjust every use in case of inevitable API changes, and we should consider limiting uses and separate them from our logic as much as possible. This is why we often wrap potentially unstable external library APIs in our own project.

This is not the only reason to wrap external APIs. This gives us a lot of freedom and stability:

- We are not afraid of API changes because we only need to change a single usage inside the wrapper.
- We can adjust the API to our project's style and logic.
- We can replace it with a different library if problems arise with this one.
- We can change the behavior of these objects if we need to (of course, do this responsibly).

There are also counterarguments to this approach:

- We need to define all these wrappers.
- Our internal API is internal, and developers need to learn it just for this project.
- There are no courses that teach how our internal API works. We also cannot expect answers on Stack Overflow.

Let me show you an example from my practice. In an Android project I co-created, we used the Picasso library to load and display images from URLs. A simple load might look as follows:

```
Picasso.get()
    .load(url)
    .into(imageView)
```

We needed to load images all around our application, probably in hundreds of places, so we decided to wrap this API in our own function. This is a simplified version of it.

```
fun ImageView.loadImage(url: String) {  
    Picasso.get()  
        .load(url)  
        .into(this)  
}
```

It's great we did that because it later turned out that we needed to load GIF images, which was not supported by Picasso, so we decided to replace this library entirely with a different one named Glide. Thanks to our wrapper, we only needed to change a single function.

```
fun ImageView.loadImage(url: String) {  
    Glide.with(context)  
        .load(url)  
        .into(this)  
}
```

Wrapping can be much more complicated than this, but it can still be worth it. Wrappers protect us not only from the pain of changing to an entirely different library, but also from changes in the API of the library we are using and from changes in our own logic.

## Summary

Wrapping an external API is a great way to protect our project from changes in this API. It also gives us a lot of freedom in adjusting the API to our needs. It is a good idea to wrap external APIs that are unstable or when we don't trust their creators to keep them stable. It is also a good idea to wrap APIs that we use heavily because it gives us more freedom in adjusting them to our needs. On the other hand, wrapping requires defining a lot of functions, and it makes our internal API more complicated. So, we should always consider wrapping external APIs because it's often worth it.

## Item 29: Minimize elements' visibility

When we design an API, there are many reasons to prefer it to be as lean as possible. Let's specify the most important ones:

**It is easier to learn and maintain a smaller interface.** Understanding a class is easier when there are only a few things we can do with it than when there are dozens. Maintenance is easier as well. When we make changes, we often need to understand the whole class. When fewer elements are visible, there is less to maintain and test.

**When we want to make changes, it is way easier to expose something new than to hide an existing element.** All publicly visible elements are part of our public API, therefore they can be used externally. The longer an element has been visible for, the more external usages it has. As such, changing these elements will be harder because they will require updating all usages. Restricting visibility would be even more of a challenge because you'll need to carefully consider each usage and provide an alternative. Giving an alternative might not be simple, especially if this element was implemented by another developer. It might also be tough to find out now what the original business requirements were. If it is a public library, restricting some elements' visibility might make some users angry because they'll need to adjust their implementation and will face the same problems - they'll need to implement alternative solutions probably years after they developed their code. It is much better to force developers to use a smaller API in the first place.

**A class cannot be responsible for its own state when properties that represent this state can be changed from the outside.** We might have assumptions, that class state needs to satisfy. When this state can be directly changed from the outside, the current class cannot guarantee its invariants because it might be changed externally by someone who doesn't know anything about our internal contract. Take a look at CounterSet from the snippet below. We correctly restricted the visibility of the `elementsAdded` setter. Without this, someone from outside might change it to any value and we wouldn't be able to trust that this value really represents how many elements were added. Notice that only setters are private. This is a very useful trick.

```
class CounterSet<T>(
    private val innerSet: MutableSet<T> = mutableSetOf()
) : MutableSet<T> by innerSet {

    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        elementsAdded += elements.size
        return innerSet.addAll(elements)
    }
}
```

For many cases, it is very helpful that all properties are encapsulated by default in Kotlin because we can always restrict the visibility of concrete accessors.

Protecting an object's internal state is especially important when we have properties that depend on each other. For instance, in the `mutableLazy` delegate implementation below, we expect that if `initialized` is true, `value` is initialized and contains a value of type `T`. Whatever we do, the setter of `initialized` should not be exposed, because otherwise it cannot be trusted, which could lead to an ugly exception on a different property.

```
class MutableLazyHolder<T>(val initializer: () -> T) {
    private var value: Any? = Any()
    private var initialized = false

    fun get(): T {
        if (!initialized) {
            value = initializer()
            initialized = true
        }
        return value as T
    }

    fun set(value: T) {
        this.value = value
        initialized = true
    }
}
```

```
    }  
}
```

**It is easier to track how a class changes when it has more restricted visibility.** This makes the property state easier to understand and is especially important when we are dealing with concurrency. State changes are a problem for parallel programming, so it is better to control and restrict them as much as possible.

## Using visibility modifiers

To achieve a smaller interface from outside without internal sacrifices, we restrict elements' visibility. In general, if there is no reason for an element to be visible, we prefer to hide it. This is why if there is no good reason to have less restrictive visibility, it is good practice to make the visibility of classes and elements as restrictive as possible. We do this using visibility modifiers.

For class members, these are the 4 visibility modifiers we can use together with their behavior:

- `public` (default) - visible everywhere for clients who can see the declaring class.
- `private` - visible inside this class only.
- `protected` - visible inside this class and in subclasses.
- `internal` - visible inside this module for clients who see the declaring class.

Top-level elements have 3 visibility modifiers:

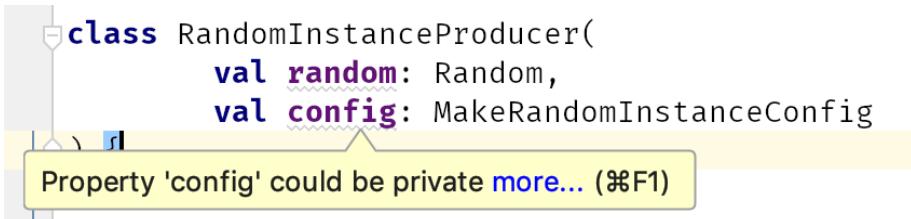
- `public` (default) - visible everywhere.
- `private` - visible inside the same file only.
- `internal` - visible inside this module.

Note that a module is not the same as a package. In Kotlin, a module is defined as a set of Kotlin sources compiled together. This might be:

- a Gradle source set,
- a Maven project,
- an IntelliJ IDEA module,
- a set of files compiled with one invocation of the Ant task.

If your module might be used by another module, change the visibility of public elements that you don't want to expose to `internal`. If an element is designed for inheritance and is only used in a class and subclasses, make it `protected`. If you

use an element only in the same file or class, make it `private`. This convention is supported by Kotlin, which suggests restricting visibility to private if an element is used only locally:



This rule should not be applied to properties in classes that are designed primarily to hold data (data model classes, DTO). If your server returns a user with an age, and you decide to parse it, you don't need to hide it just because you don't use it at the moment. It is there to be used and it is better to have it visible. If you don't need it, get rid of this property entirely.

```
class User(
    val name: String,
    val surname: String,
    val age: Int
)
```

One big limitation is that when we inherit an API, we cannot restrict the visibility of a member by overriding it. This is because the subclass can always be used as its superclass. This is just another reason to prefer composition instead of inheritance ([Item 36: Prefer composition over inheritance](#)).

## Summary

The rule of thumb is that: **Elements' visibility should be as restrictive as possible**. A public API consists of its visible elements, and we prefer it as lean as possible because:

- It is easier to learn and maintain a smaller interface.
- When we want to make changes, it is way easier to expose something than to hide it.
- A class cannot be responsible for its own state when properties that represent this state can be changed from outside.
- It is easier to track how an API changes when it has more restricted visibility.

## Item 30: Define contracts with documentation

Think again about the function to display a message from Item 26: Use abstraction to protect code against changes:

```
fun Context.showMessage(  
    message: String,  
    length: MessageLength = MessageLength.LONG  
) {  
    val toastLength = when(length) {  
        SHORT -> Toast.LENGTH_SHORT  
        LONG -> Toast.LENGTH_LONG  
    }  
    Toast.makeText(this, message, toastLength).show()  
}  
  
enum class MessageLength { SHORT, LONG }
```

We extracted the showMessage function to give ourselves the freedom to change how the message is displayed. However, this function is not well documented. Another developer might read its code and assume that this function always displays a toast. This is the opposite of what we wanted to achieve by naming this function showMessage, so in a way that does not suggest a concrete message type. To make our intention clear, it would be better to add a meaningful KDoc comment explaining what should be expected from this function.

```
/**  
 * Universal way for the project to display a short  
 * message to a user.  
 * @param message The text that should be shown to  
 * the user  
 * @param length How long to display the message.  
 */  
fun Context.showMessage(  
    message: String,  
    duration: MessageLength = MessageLength.LONG  
) {  
    val toastDuration = when(duration) {  
        SHORT -> Toast.LENGTH_SHORT  
        LONG -> Toast.LENGTH_LONG  
    }
```

```

    Toast.makeText(this, message, toastDuration).show()
}

enum class MessageLength { SHORT, LONG }

```

In many cases, there are details that are not clearly inferred by the name at all. For instance, powerset, even though it is a well-defined mathematical concept, needs an explanation since it is not well known:

```

/**
 * Powerset returns a set of all subsets of the receiver
 * including itself and the empty set
 */
fun <T> Collection<T>.powerset(): Set<Set<T>> =
    if (isEmpty()) setOf(emptySet())
    else take(size - 1)
        .powerset()
        .let { it + it.map { it + last() } }

```

Notice that this description gives us some freedom as it does not specify the order of these elements. As a user, we should not depend on how these elements are ordered. The implementation hidden behind this abstraction can be optimized without changing how this function looks from the outside:

```

/**
 * Powerset returns a set of all subsets of the receiver
 * including itself and empty set
 */
fun <T> Collection<T>.powerset(): Set<Set<T>> =
    powerset(this, setOf(setOf()))

private tailrec fun <T> powerset(
    left: Collection<T>,
    acc: Set<Set<T>>
): Set<Set<T>> = when {
    left.isEmpty() -> acc
    else -> {
        val head = left.first()
        val tail = left.drop(1)
        powerset(tail, acc + acc.map { it + head })
    }
}

```

The general problem is that **when a behavior is not documented and an element name is not clear, developers will depend on the current implementation instead of on the abstraction we intended to create.** We solve this problem by describing the behavior that can be expected.

## Contracts

Whenever we describe a behavior, users treat it as a promise and on this basis they adjust their expectations. We call any such expected behavior a contract of an element. Just like in a real-life contract, the other side expects us to honor it. The same is true here: users will expect us to honor this contract once it is stable (Item 27: Specify API stability).

At this point, defining a contract might sound scary, but actually it is great for both sides. **When a contract is well specified, creators do not need to worry about how the class is used, and users do not need to worry about how something is implemented under the hood.** Users can rely on this contract without knowing anything about the actual implementation. For creators, a contract gives the freedom to change everything as long as the contract is satisfied. **Both users and creators depend on the abstractions defined in the contract, therefore they can work independently.** Everything will work perfectly fine as long as the contract is respected. This is reassurance and freedom for both sides.

What if we don't define a contract? **Without users knowing what they can and cannot do, they'll depend on implementation details instead. A creator who doesn't know what users depend on would be either blocked or would risk breaking users implementations.** As you can see, it is important to define a contract.

## Defining a contract

How do we define a contract? There are various ways, including:

- Names - when a name is connected to a more general concept, we expect this element to be consistent with this concept. For instance, when you see the `sum` method, you don't need to read its comment to know how it will behave. This is because summation is a well-defined mathematical concept.
- Comments and documentation - this is the most powerful way as it can describe everything that is needed.
- Types - Types say a lot about objects. Each type specifies a set of often well-defined methods, and some types also have set-up responsibilities in their documentation. When we see a function, information about the return type and argument types are very meaningful.

## Do we need comments?

Looking back, it is amazing to see how opinions in the community fluctuate. When Java was still young, there was a very popular concept called literate programming that suggested explaining everything in comments<sup>52</sup>. A decade later, comments are widely criticized and many say that we should omit them completely and concentrate on writing readable code instead (I believe that the most influential book that suggested that was the Clean Code by Robert C. Martin).

No extreme is healthy. I absolutely agree that we should first concentrate on writing readable code. However, it should also be understood that comments before elements can describe them at a higher level and define their contracts. **Additionally, comments are now often used to automatically generate documentation, which generally is treated as a source of truth in projects.**

Sure, we often do not need comments. For instance, many functions are self-explanatory and don't need any special description. We might, for instance, assume that a product is a clear mathematical concept that is known by programmers therefore no comments are needed:

```
fun List<Int>.product() = fold(1) { acc, i -> acc * i }
```

Obvious comments are noise that only distracts us. Do not write comments that only describe what is clearly expressed by a function name and parameters. The following example demonstrates an unnecessary comment because its meaning can be inferred from the method's name and parameter type:

```
// Product of all numbers in a list
fun List<Int>.product() = fold(1) { acc, i -> acc * i }
```

I also agree that when we just need to organize our code, instead of writing comments in the implementation, we should rather extract a function. Take a look at the example below:

---

<sup>52</sup>Read more about this concept in the book Literate Programming by Donald Knuth.

```
fun update() {  
    // Update users  
    for (user in users) {  
        user.update()  
    }  
  
    // Update books  
    for (book in books) {  
        updateBook(book)  
    }  
}
```

The `update` function is clearly composed of extractable parts, and comment suggests that these parts can be described with a different explanation. Therefore, it is better to extract these parts into separate abstractions such as methods, whose names should be clear enough to explain what they mean (just like in Item 25: *Each function should be written in terms of a single level of abstraction*).

```
fun update() {  
    updateUsers()  
    updateBooks()  
}  
  
private fun updateBooks() {  
    for (book in books) {  
        updateBook(book)  
    }  
}  
  
private fun updateUsers() {  
    for (user in users) {  
        user.update()  
    }  
}
```

However, comments are often useful and important. To find examples, take a look at nearly any public function from the Kotlin standard library. They have well-defined contracts that give a lot of freedom. For instance, take a look at the `listOf` function:

```
/**  
 * Returns a new read-only list of given elements.  
 * The returned list is serializable (JVM).  
 * @sample samples.collections.Collections.Lists.  
 readOnlyList  
 */  
public fun <T> listOf(vararg elements: T): List<T> =  
    if (elements.size > 0) elements.asList()  
    else emptyList()
```

It only promises to return a `List` that is read-only and serializable on JVM. Nothing else. The list does not need to be immutable. No concrete class is promised. This contract is minimalistic but it's enough for the needs of most Kotlin developers. You can also see that it points to sample uses, which are also useful when we are learning how to use an element.

## The KDoc format

When we document functions using comments, the official format in which we present these comments is called KDoc. All KDoc comments start with `/**` and end with `*/`, and all internal lines generally start with `*`. Descriptions there are written in KDoc markdown.

The structure of a KDoc comment is the following:

- The first paragraph of the documentation text is the summary description of the element.
- The second part is the detailed description.
- Every next line begins with a tag. These tags are used to reference an element to describe it.

Here are the supported tags:

- `@param <name>` - Documents a value parameter of a function or a type parameter of a class, property or function.
- `@return` - Documents the return value of a function.
- `@constructor` - Documents the primary constructor of a class.
- `@receiver` - Documents the receiver of an extension function.
- `@property <name>` - Documents the property of the specified class. Used for properties defined in the primary constructor.
- `@throws <class>, @exception <class>` - Documents an exception which can be thrown by a method.

- `@sample <identifier>` - Embeds the body of the specified function into the documentation for the current element in order to show an example of how the element could be used.
- `@see <identifier>` - Adds a link to the specified class or method.
- `@author` - Specifies the author of the documented element.
- `@since` - Specifies the version of the software in which the documented element was introduced.
- `@suppress` - Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.

In both descriptions and in text that describes tags we can link classes, methods, properties or parameters. Links are in square brackets, or double square brackets when we want to have a different description than the name of the linked element.

```
/**  
 * This is an example descriptions linking to [element1],  
 * [com.package.SomeClass.element2] and  
 * [this element with custom description][element3]  
 */
```

All these tags will be understood by Kotlin documentation generation tools. The official one is called Dokka. They generate documentation files that can be published online and presented to outside users. Here is an example documentation with a shortened description:

```
/**  
 * Immutable tree data structure.  
 *  
 * Class represents an immutable tree with from 1 to  
 * an infinite number of elements. In the tree we hold  
 * elements on each node, and nodes can have left and  
 * right subtrees...  
 *  
 * @param T the type of elements this tree holds.  
 * @property value the value kept in this node of the tree.  
 * @property left the left subtree.  
 * @property right the right subtree.  
 */  
class Tree<T>(  
    val value: T,
```

```
    val left: Tree<T>? = null,
    val right: Tree<T>? = null
) {
    /**
     * Creates a new tree based on the current but with
     * [element] added.
     * @return newly created tree with additional element.
     */
    operator fun plus(element: T): Tree { ... }
}
```

Notice that not everything needs to be described. The best documentation is short and precisely describes what might be unclear.

## The type system and expectations

Type hierarchy is an important source of information about an object. An interface is more than just a list of methods we promise to implement. Classes and interfaces can also have some expectations. If a class promises something, all of its subclasses should guarantee that too. This principle is known as the Liskov substitution principle, and it is one of the most important rules in object-oriented programming. It is generally translated to “if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program”. A simple explanation of why this is important is that every class can be used as a superclass; so, if a class does not behave as we expect its superclass to behave, we might end up with unexpected failure. In programming, children should always satisfy parents’ contracts.

One important implication of this rule is that we should properly specify contracts for open functions. For instance, coming back to our car metaphor, we could represent a car in our code using the following interface:

```
interface Car {
    fun setWheelPosition(angle: Float)
    fun setBreakPedal(pressure: Double)
    fun setGasPedal(pressure: Double)
}

class GasolineCar: Car {
    // ...
}

class GasCar: Car {
```

```
// ...
}

class ElectricCar: Car {
    // ...
}
```

The problem with this interface is that it leaves a lot of questions. What does `angle` in the `setWheelPosition` function mean? In which units is it measured? What if it is not clear for someone what the gas and brake pedals do? People using instances of type `Car` need to know how to use them, and all the classes that implement this interface should behave similarly when they are used as a `Car`. We can address these concerns with documentation:

```
interface Car {
    /**
     * Changes car direction.
     *
     * @param angle Represents position of wheels in
     * radians relatively to car axis. 0 means driving
     * straight, pi/2 means driving maximally right,
     * -pi/2 maximally left.
     * Value needs to be in (-pi/2, pi/2)
     */
    fun setWheelPosition(angle: Float)

    /**
     * Decelerates vehicle speed until 0.
     *
     * @param pressure The percentage of brake pedal use.
     * Number from 0 to 1 where 0 means not using break
     * at all, and 1 means maximal pedal pedal use.
     */
    fun setBreakPedal(pressure: Double)

    /**
     * Accelerates vehicle speed until max speed possible
     * for the user.
     *
     * @param pressure The percentage of gas pedal use.
     * Number from 0 to 1 where 0 means not using gas at
     * all, and 1 means maximal gas pedal use.
     */
}
```

```
 */
fun setGasPedal(pressure: Double)
}
```

Now, all cars have a defined standard that describes how they all should behave.

Most classes in the stdlib and in popular libraries have well-defined and well-described contracts and expectancies for their children. We should also define contracts for our elements because they will make these interfaces truly useful. They will give us the freedom to use classes that implement these interfaces in the way their contract guarantees.

## Leaking implementation

Implementation details always leak. In cars, different kinds of engines behave a bit differently. We are still able to drive all cars, but we can feel a difference. This is fine even though it is not described in the contract.

In programming languages, implementation details leak as well. For instance, calling a function using reflection works, but it is significantly slower than a normal function call (unless it is optimized by the compiler). We will see more examples in the chapter about performance optimization. As long as a language works as it promises, everything is fine. We just need to remember and apply good practices.

In our abstractions, implementations will also leak, but we should still protect them as much as we can. We protect them by encapsulation, which can be described as “You can do what I allow, and nothing more”. The more we encapsulate our classes and functions, the more freedom we have inside them because we don’t need to think about how someone else might depend on our implementation.

## Summary

When we define an element, especially part of an external API, we should define a contract. We do this using names, documentation, comments, and types. The contract specifies what these elements’ expectations are. It can also describe how an element should be used.

A contract gives users confidence about how elements behave now and will behave in the future; it gives creators the freedom to change what is not specified in the contract. The contract is a kind of agreement, and it works well as long as both sides respect it.

## Item 31: Respect abstraction contracts

Both contracts and visibility are kind of an agreement between developers. This agreement can nearly always be violated by a user. Technically, everything in a single project can be hacked. For instance, it is possible to use reflection to open and use anything we want:

```
class Employee {
    private val id: Int = 2
    override fun toString() = "User(id=$id)"

    private fun privateFunction() {
        println("Private function called")
    }
}

fun callPrivateFunction(employee: Employee) {
    employee::class.declaredMemberFunctions
        .first { it.name == "privateFunction" }
        .apply { isAccessible = true }
        .call(employee)
}

fun changeEmployeeId(employee: Employee, newId: Int) {
    employee::class.java.getDeclaredField("id")
        .apply { isAccessible = true }
        .set(employee, newId)
}

fun main() {
    val employee = Employee()
    callPrivateFunction(employee)
    // Prints: Private function called

    changeEmployeeId(employee, 1)
    print(employee) // Prints: User(id=1)
}
```

Just because you can do something doesn't mean that it is fine to do it. Here, we very strongly depend on implementation details, such as the names of the private property and the private function. They are not part of the contract at all, so they might change at any moment. This is like a ticking time bomb for our program.

Remember that a contract is like a warranty. As long as you use your computer correctly, the warranty protects you. When you open your computer and start hacking it, you lose your warranty. The same principle applies here: when you break the contract, it is your problem when the implementation changes and your code stops working.

## Contracts are inherited

It is especially important to respect contracts when we inherit from classes, or when we extend interfaces from another library. Remember that a children should respect parents contracts. For instance, every class extends `Any` that has `equals` and `hashCode` methods. Both those methods have well-established contracts that we need to respect. If we don't, our objects might not work correctly. For instance, when `hashCode` is not consistent with `equals`, our object might not behave correctly on `HashSet`. The behavior shown below is incorrect because a set should not allow duplicates:

```
class Id(val id: Int) {
    override fun equals(other: Any?) =
        other is Id && other.id == id
}

val mutableSet = mutableSetOf(Id(1))
mutableSet.add(Id(1))
mutableSet.add(Id(1))
print(mutableSet.size) // 3
```

In this case, the violated contract is that `hashCode` implementation should be consistent with `equals`. We will discuss it in the Item 43: Respect the contract of `hashCode`. We will also discuss many other contracts defined by methods from `Any` in the Chapter 6: Class design. For now, remember to check and respect the expectations on the functions you override.

## Summary

If you want your programs to be stable, respect contracts. If you are forced to break them, document this fact well. Such information will be very helpful to whoever maintains your code, even if that's just you in a few years' time.

# Chapter 5: Object creation

Although Kotlin can be written in a purely functional style, it can also be written in object-oriented programming style (OOP), much like Java. In OOP, we need to create every object we use, or at least define how it ought to be created, and different object creation methods have different characteristics. It is important to know which options we have, which is why this chapter shows different ways of defining object creation and explains their advantages and disadvantages.

If you are familiar with the *Effective Java* book by Joshua Bloch, then you may notice some similarities between this chapter and that book. This is no coincidence as this chapter mirrors the first chapter of Effective Java to some extent; however, Kotlin is very different from Java, therefore only a few morsels of knowledge are exchangeable between the two. For instance, static methods are not allowed in Kotlin, but we have very good alternatives like top-level functions and companion object functions, which don't work the same way as static functions, so it is important to understand them. There are also similarities with other items, but the changes that Kotlin has introduced are important. The good news is that these changes are mostly intended to provide more possibilities or force a better style. Kotlin is a powerful and really well-designed language, and this chapter mainly focuses on opening your eyes to these new possibilities.

## Item 32: Consider factory functions instead of constructors

To create an object from a class, you need to use a constructor. In Kotlin, it is typically the primary constructor<sup>53</sup>:

```
class LinkedList<T>(  
    val head: T,  
    val tail: LinkedList<T>?  
)  
  
val list = LinkedList(1, LinkedList(2, null))
```

It is typical of Kotlin classes that their primary constructor defines properties that are essential part of this object state; as a result, primary constructor parameters are strongly bound to object structure. Using primary constructor for object creation is enough for simple classes, but more complex cases require different ways of constructing them. Think of the `LinkedList` from the above snippet. We might want to create it:

- Based on a set of items passed with the `vararg` parameter.
- From a collection of a different type, like `List` or `Set`.
- From another instance of the same type.

It is poor practice to define such functions as constructors. Don't do that. It is better to define them as functions (like `linkedListOf`, `toLinkedList` or `copy`), and here are a few reasons why:

- **Unlike constructors, functions have names.** Names explain how an object is created and what the arguments are. For example, let's say that you see the following code: `ArrayList(3)`. Can you guess what the argument means? Is it supposed to be the first element in the newly created list, or is it the initial capacity of the list? It is definitely not self-explanatory. In such a situation, a name like `ArrayList.withCapacity(3)` would clear up any confusion. Names are really useful: they explain arguments or characteristic ways of object creation. Another reason to have a name is that it solves potential conflicts between constructors with the same parameter types.
- **Unlike constructors, functions can return an object of any subtype of their return type.** This is especially important when we want to hide actual object implementations behind an interface. Think of `listOf` from

---

<sup>53</sup>See the section about primary/secondary constructors in the dictionary.

stdlib. Its declared return type is `List`, which is an interface. But what does this really return? The answer depends on the platform we use. It is different for Kotlin/JVM, Kotlin/JS, and Kotlin/Native because they each use different built-in collections. This is an important optimization that was implemented by the Kotlin team. It also gives Kotlin creators much more freedom. The actual type of a list might change over time, but as long as new objects still implement the `List` interface and act the same way, everything will be fine. Another example is `lazy` that declares `Lazy` interface as its result type, and depending on the thread safety mode, in JVM it returns either `SynchronizedLazyImpl`, `SafePublicationLazyImpl` or `UnsafeLazyImpl`. Each of those classes is private, so their implementations are protected.

- **Unlike constructors, functions are not required to create a new object each time they're invoked.** This can be helpful because when we create objects using functions, we can include a caching mechanism to optimize object creation or to ensure object reuse for some cases (like in the Singleton pattern). We can also define a static factory function that returns `null` if the object cannot be created, like `Connections.createOrNull()`, which returns `null` when `Connection` cannot be created for some reason.
- **Factory functions can provide objects that might not yet exist.** This is intensively used by creators of libraries that are based on annotation processing. In this way, programmers can operate on objects that will be generated or used via a proxy without building the project.
- **When we define a factory function outside an object, we can control its visibility.** For instance, we can make a top-level factory function accessible only in the same file (`private` modifier) or in the same module (`internal` modifier).
- **Factory functions can be inlined, so their type parameters can be reified<sup>54</sup>.** Libraries use this to provide a more convenient API.
- **A constructor needs to immediately call a constructor of a superclass or a primary constructor. When we use factory functions, we can postpone constructor usage.** That allows us to include a more complex algorithm in object creation.

Functions used to create an object are called **factory functions**. They are very important in Kotlin. When you search through Kotlin's official libraries, including the standard library, you will have trouble finding a non-private constructor, not to mention a secondary constructor. Most libraries expose only factory functions, most applications expose only primary constructors.

There are many kinds of factory functions we can use. We can create a list with `listOf`, `toList`, `List`, etc. Those are all factory functions. Let's learn about the most important kinds of factory functions and their conventions:

---

<sup>54</sup>Reified type parameters are explained in Item 51: Use the `inline` modifier for functions with parameters of functional types.

1. Companion object factory functions
2. Top-level factory functions
3. Builders
4. Conversion methods
5. Copying methods
6. Fake constructors
7. Methods in factory classes

## Companion Object Factory Functions

In Java, every function has to be placed in a class. This is why most factory functions in Java are static functions that are placed either in the class they are producing or in some accumulator of static functions (like `Files`). Since the majority of the Kotlin community originated in Java, it has become popular to mimic this practice by defining factory functions in companion objects:

```
class LinkedList<T>(
    val head: T,
    val tail: LinkedList<T>?
) {

    companion object {
        fun <T> of(vararg elements: T): LinkedList<T> {
            /*...*/
        }
    }
}

// Usage
val list = LinkedList.of(1, 2)
```

The same can also be done with interfaces:

```
class LinkedList<T>(
    val head: T,
    val tail: LinkedList<T>?
) : MyList<T> {
    // ...
}

interface MyList<T> {
```

```
// ...

companion object {
    fun <T> of(vararg elements: T): MyList<T> {
        // ...
    }
}

// Usage
val list = MyList.of(1, 2)
```

The advantage of this practice is that it is widely recognized among different programming languages. In some languages, like C++, it is called a **Named Constructor Idiom** as its usage is similar to a constructor, but with a name. It is also highly interoperable with other languages. From my personal experience, we used **companion object factory functions** most often when we were writing tests in Groovy. You just need to use `JvmStatic` annotation before the function, and you can easily use such a function in Groovy or Java in the same way as you use it in Kotlin.

The disadvantage of this practice is its complexity. Writing `List.of` is longer than `listOf` because it requires applying a suggestion two times instead of one. A companion object factory function needs to be defined in a companion object, while a top-level function can be defined anywhere.

It is worth mentioning that a companion object factory function can be defined as an extension to a companion object. It is possible to define an extension function to a companion object as long as such an object (even an empty one) exists.

```
interface Tool {
    companion object { /*...*/ }
}

fun Tool.Companion.createBigTool(/*...*/): Tool {
    //...
}

val tool = Tool.createBigTool()
```

There are some naming conventions for companion object factory functions. They are generally a Java legacy, but they still seem to be alive in our community:

- `from` - A type-conversion function that expects a single argument and returns a corresponding instance of the same type, for example:  
`val date: Date = Date.from(instant)`
- `of` - An aggregation function that takes multiple arguments and returns an instance of the same type that incorporates them, for example:  
`val faceCards: Set<Rank> = EnumSet.of(JACK, QUEEN, KING)`
- `valueOf` - A more verbose alternative to `from` and `of`, for example:  
`val prime: BigInteger = BigInteger.valueOf(Integer.MAX_VALUE)`
- `instance` or `getInstance` - Used in singletons to get the object instance. When parameterized, it will return an instance parameterized by arguments. Often, we can expect the returned instance to always be the same when the arguments are the same, for example:  
`val luke: StackWalker = StackWalker.getInstance(options)`
- `createInstance` or `newInstance` - Like `getInstance`, but this function guarantees that each call returns a new instance, for example:  
`val newArray = Array.newInstance(classObject, arrayLen)`
- `get{Type}` - Like `getInstance`, but used if the factory function is in a different class. Type is the type of the object returned by the factory function, for example:  
`val fs: FileStore = Files.getFileStore(path)`
- `new{Type}` - Like `newInstance`, but used if the factory function is in a different class. Type is the type of object returned by the factory function, for example:  
`val br: BufferedReader = Files.newBufferedReader(path)`

Those conventions map to other kind of factory functions. For example, `listOf` suggests that it creates a list from a set of elements, so it is an aggregation function. `createViewModel` suggests that it creates a new instance of a `ViewModel`, so it is a `new{Type}` function.

Companion objects are often treated as an alternative to static elements, but they are much more than that. Companion objects can implement interfaces and extend classes. This is a response to a popular request to allow inheritance for “static” elements. You can create abstract builders that are extended by concrete companion objects:

```
abstract class ActivityFactory {
    abstract fun getIntent(context: Context): Intent

    fun start(context: Context) {
        val intent = getIntent(context)
        context.startActivity(intent)
    }

    fun startForResult(
        activity: Activity,
        requestCode: Int
    ) {
        val intent = getIntent(activity)
        activity.startActivityForResult(
            intent,
            requestCode
        )
    }
}

class MainActivity : AppCompatActivity() {
    // ...

    companion object : ActivityFactory() {
        override fun getIntent(context: Context): Intent =
            Intent(context, MainActivity::class.java)
    }
}

// Usage
val intent = MainActivity.getIntent(context)
MainActivity.start(context)
MainActivity.startForResult(activity, requestCode)
```

Notice that such abstract companion object factories can hold values, and so they can implement caching or support fake creation for testing. The advantages of companion objects are not as well used as they could be in the Kotlin programming community. Still, if you look at the implementations of the Kotlin team's libraries, you will see that companion objects are used extensively. For instance, in the Kotlin Coroutines library, nearly every companion object of a coroutine context implements a `CoroutineContext.Key` interface, which serves as a key we

use to identify this context<sup>55</sup>.

## Top-level factory functions

A popular way to create an object is by using top-level factory functions. Some common examples are `listOf`, `setOf`, `mapOf`, `lazy`, `sequence`, `flow`, etc.

```
fun <T> lazy(mode: LazyThreadSafetyMode, init: () -> T): Lazy<T>
=>
    when (mode) {
        SYNCHRONIZED -> SynchronizedLazyImpl(init)
        PUBLICATION -> SafePublicationLazyImpl(init)
        NONE -> UnsafeLazyImpl(init)
    }
```

Top-level factory functions are also used in projects to create objects in the way specific to the project. For instance, this is how one project could define Retrofit service creation:

```
fun createRetrofitService(baseUrl: String): Retrofit {
    return Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
}

fun <T> createService(clazz: Class<T>, baseUrl: String): T {
    val retrofit = createRetrofitService(baseUrl)
    return retrofit.create(clazz)
}
```

Object creation using top-level functions is a perfect choice for small and commonly created objects like `List` or `Map` because `listOf(1,2,3)` is simpler and more readable than `List.of(1,2,3)`. However, public top-level functions need to be used judiciously. Public top-level functions have a disadvantage: they are available everywhere, therefore it is easy to clutter up the developer's IDE tips. This problem becomes more serious when top-level functions have the same names as class methods and therefore get confused with them. This is why top-level functions should be named wisely.

---

<sup>55</sup>This mechanism is better explained in my Kotlin Coroutines book.

## Builders

A very important kind of top-level factory function is builders. A good example is a list or a sequence builder:

```
val list = buildList {
    add(1)
    add(2)
    add(3)
}
println(list) // [1, 2, 3]

val s = sequence {
    yield("A")
    yield("B")
    yield("C")
}
println(s.toList()) // [A, B, C]
```

The typical way to implement a builder in Kotlin is using a top-level function and a DSL pattern<sup>56</sup>. In Kotlin Coroutines, builders are the standard way to start a coroutine or define a flow:

```
// Starting a coroutine
scope.launch {
    val processes = repo.getActiveProcesses()
    for (process in processes) {
        launch {
            process.start()
            repo.markProcessAsDone(process.id)
        }
    }
}

// Defining a flow
val flow = flow {
    var lastId: String = null
    do {
        val page = fetchPage(lastId)
```

---

<sup>56</sup>This will be explained soon in Item 34: Consider defining a DSL for complex object creation.

```
        emit(page.data)
        lastId = page.lastId
    } while (!page.isLast)
}
```

We will discuss DSLs in detail in [Item 34: Consider defining a DSL for complex object creation](#). Of course, you can also meet builders defined using Java Builder Pattern, that look like this:

```
val user = UserBuilder()
    .withName("Marcin")
    .withSurname("Moskala")
    .withAge(30)
    .build()
```

In Kotlin we consider them less idiomatic than DSL builders. However, they are still used in some libraries.

## Conversion methods

We often convert from one type to another. You might convert from `List` to `Sequence`, from `Int` to `Double`, from RxJava `Observable` to `Flow`, etc. For all these, the standard way is to use **conversion methods**. Conversion methods are methods used to convert from one type to another. They are typically named `to{Type}` or `as{Type}`. For example:

```
val sequence: Sequence = list.asSequence()

val double: Double = i.toDouble()

val flow: Flow = observable.asFlow()
```

The `to` prefix means that we are actually creating a new object of another type. For instance, if you call `toList` on a `Sequence`, you will get a new `List` object, which means that all elements of the new list are calculated and accumulated into a newly created list when this function is called. The `as` prefix means that the newly created object is a wrapper or an extracted part of the original object. For example, if you call `asSequence` on a `List`, the result object will be a wrapper around the original list. Using `as` conversion functions is more efficient but can lead to synchronization problems or unexpected behavior. For example, if you call `asSequence` on a `MutableList`, you will get a `Sequence` that references the original list.

```
fun main() {
    val seq1 = sequence<Int> {
        repeat(10) {
            print(it)
            yield(10)
        }
    }
    seq1.asSequence() // Nothing printed
    seq1.toList() // Prints 0123456789

    val l1 = mutableListOf(1, 2, 3, 4)
    val l2 = l1.toList()
    val seq2 = l1.asSequence()
    l1.add(5)
    println(l2) // Prints [1, 2, 3, 4]
    println(seq2.toList()) // Prints [1, 2, 3, 4, 5]
}
```

We often define our own conversion functions to convert between our own types. For example, when we need to convert between `UserJson` and `User` in an example application. Such methods are often defined as extension functions.

```
class User(
    val id: UserId,
    val name: String,
    val surname: String,
    val age: Int,
    val tokens: List<Token>
)

class UserJson(
    val id: UserId,
    val name: String,
    val surname: String,
    val age: Int,
    val tokens: List<Token>
)

fun User.toUserJson() = UserJson(
    id = this.id,
    name = this.name,
    surname = this.surname,
```

```
    age = this.age,  
    tokens = this.tokens  
)  
  
fun UserJson.toUser() = User(  
    id = this.id,  
    name = this.name,  
    surname = this.surname,  
    age = this.age,  
    tokens = this.tokens  
)
```

## Copying methods

When you need to make a copy of an object, define a copying method instead of defining a **copying constructor**. When you just want to make a direct copy, a good name is `copy`. When you need to apply a change to this object, a good name starts with `with` and the name of the property that should be changed (like `withSurname`).

```
val user2 = user.copy()  
val user3 = user.withSurname(newSurname)
```

Data classes support the `copy` method, which can modify any primary constructor property, as we will see in Item 37: Use the `data` modifier to represent a bundle of data.

## Fake constructors

Constructors in Kotlin are used the same way as top-level functions:

```
class A  
  
fun b() = A()  
  
val a1 = A()  
val a2 = b()
```

They are also referenced in the same way as top-level functions (and constructor references implement a function type):

```
val reference: () -> A = ::A
```

From a usage point of view, capitalization is the only distinction between constructors and functions. By convention, classes begin with an uppercase letter, and functions begin with a lowercase letter. However, technically, functions can begin with an uppercase letter. This is used in different places, for example, in the Kotlin standard library. `List` and `MutableList` are interfaces. They cannot have constructors, but Kotlin developers wanted to allow the following `List` construction:

```
List(4) { "User$it" } // [User0, User1, User2, User3]
```

This is why the following functions are included in the Kotlin stdlib:

```
public inline fun <T> List(
    size: Int,
    init: (index: Int) -> T
): List<T> = MutableList(size, init)

public inline fun <T> MutableList(
    size: Int,
    init: (index: Int) -> T
): MutableList<T> {
    val list = ArrayList<T>(size)
    repeat(size) { index -> list.add(init(index)) }
    return list
}
```

These top-level functions look and act like constructors, but they have all the advantages of factory functions. Lots of developers are unaware of the fact that they are top-level functions under the hood. This is why they are often called *fake* constructors. They are a specific kind of top-level factory functions.

It is a very popular pattern in Kotlin libraries to expose only an interface, and produce its instance using a fake constructor. This way the actual implementation can be hidden. That has all advantages of factory functions, like:

- Hiding the actual implementation behind an interface (see `Job`, `CoroutineScope`, `Mutex` from `kotlinx.coroutines`).
- Depending on arguments, a different implementation can be returned, optimized for the given case (see `Channel` from `kotlinx.coroutines`).
- An algorithm can be used to create an object, which is not possible with a constructor (see `List` and `MutableList`).

Here are examples of fake constructors from the Kotlin Coroutines library:

```
fun Job(parent: Job? = null): CompletableJob = JobImpl(parent)

fun CoroutineScope(context: CoroutineContext): CoroutineScope =
    ContextScope(
        if (context[Job] != null) context else context + Job()
    )
```

Fake constructors should conceptually behave like regular constructors, otherwise you should prefer a different factory function kind.

There is one more way to declare a fake constructor. A similar result can be achieved using a companion object with the `invoke` operator. Take a look at the following example:

```
class Tree<T> {

    companion object {
        operator fun <T> invoke(
            size: Int,
            generator: (Int) -> T
        ): Tree<T> {
            //...
        }
    }

    // Usage
    Tree(10) { "$it" }
```

However, implementing `invoke` in a companion object to make a fake constructor is considered less idiomatic. I do not recommend it, primarily because it violates *Item 11: An operator's meaning should be consistent with its function name*. What does it mean to invoke a companion object? Remember that the name can be used instead of the operator:

```
Tree.invoke(10) { "$it" }
```

Invocation is a different operation from object construction. Using the `invoke` operator in this way is inconsistent with its name. More importantly, this approach is more complicated than just a top-level function. Just compare what reflection looks like when we reference a constructor, a fake constructor, and the `invoke` function in a companion object:

Constructor:

```
val f: ()->Tree = ::Tree
```

Fake constructor:

```
val f: ()->Tree = ::Tree
```

Invoke in a companion object:

```
val f: ()->Tree = Tree.Companion::invoke
```

I recommend using standard top-level functions when you need a fake constructor. However, these should be used sparingly to suggest typical constructor-like usage when we cannot define a constructor in the class itself, or when we need a capability that constructors do not offer (like a reified type parameter).

## Methods on factory classes

There are many creational patterns associated with factory classes. For instance, an abstract factory or a prototype. Every creational pattern has some advantages.

Factory classes hold advantages over factory functions because classes can have a state. For instance, this is a very simple factory class that produces students with sequential id numbers:

```
data class Student(  
    val id: Int,  
    val name: String,  
    val surname: String  
)  
  
class StudentsFactory {  
    var nextId = 0  
    fun next(name: String, surname: String) =  
        Student(nextId++, name, surname)  
}  
  
val factory = StudentsFactory()  
val s1 = factory.next("Marcin", "Moskala")  
println(s1) // Student(id=0, name=Marcin, Surname=Moskala)  
val s2 = factory.next("Igor", "Wojda")  
println(s2) // Student(id=1, name=Igor, Surname=Wojda)
```

Factory classes can have properties that can be used to optimize object creation. When we can hold a state, we can introduce different kinds of optimizations or capabilities. We can, for instance, use caching or speed up object creation by duplicating previously created objects.

In practice, we most often extract factory classes when object creation requires multiple services or repositories. Extracting object creation logic helps us better organize our code.

```
class UserFactory(
    private val uuidProvider: UuidProvider,
    private val timeProvider: TimeProvider,
    private val tokenService: TokenService,
) {
    fun create(newUserData: NewUserData): User {
        val id = uuidProvider.next()
        return User(
            id = id,
            creationTime = timeProvider.now(),
            token = tokenService.generateToken(id),
            name = newUserData.name,
            surname = newUserData.surname,
            // ...
        )
    }
}
```

## Summary

As you can see, Kotlin offers a variety of ways to specify factory functions, and they all have their own use. We should have them in mind when we design object creation. Each of them is reasonable for different cases. The most important thing is to be aware of the differences between them and to use them appropriately. The most popular factory function types are:

- Companion object factory functions
- Top-level factory functions (including fake constructors and builders)
- Conversion functions
- Methods on factory classes

## Item 33: Consider a primary constructor with named optional arguments

When we define an object and specify how it can be created, the most popular option is to use the primary constructor:

```
class User(var name: String, var surname: String)  
  
val user = User("Marcin", "Moskała")
```

Not only are primary constructors very convenient, but in most cases it is actually very good practice to build objects using them. It is common that we need to pass arguments that determine an object's initial state, as illustrated by the following examples, starting from the most obvious one: data model objects that represent data<sup>57</sup>. For such an object, its whole state is initialized using a constructor and then held as properties.

```
data class Student(  
    val name: String,  
    val surname: String,  
    val age: Int  
)
```

Here's another common example in which we create a presenter<sup>58</sup> for displaying a sequence of indexed quotes. We inject dependencies into this object using a primary constructor<sup>59</sup>:

---

<sup>57</sup>A data model is not necessarily a data class, and vice versa. The former concept represents classes in our project that represent data, while the latter is special support for such classes. This special support is a set of functions that we might not need or that we might need for classes that do not serve as our data model.

<sup>58</sup>A presenter is a kind of object that is used in the Model View Presenter (MVP) architecture, which used to be popular in Android before its successor MVVM became popular.

<sup>59</sup>Dependency injection is a technique in which an object receives other objects that it depends on. By itself, it does not need any library (or framework) like Koin or Dagger, although I find them useful.

```
class QuotationPresenter(    private val view: QuotationView,    private val repo: QuotationRepository){    private var nextQuoteId = -1  
  
    fun onStart() {        onNext()  
    }  
  
    fun onNext() {        nextQuoteId = (nextQuoteId + 1) % repo.quotesNumber        val quote = repo.getQuote(nextQuoteId)        view.showQuote(quote)  
    }  
}
```

Note that `QuotationPresenter` has more properties than those declared in the primary constructor. Here, `nextQuoteId` is a property that is always initialized with the value `-1`. This is perfectly fine, especially when the initial state is set up using default values or primary constructor parameters.

To better understand why the primary constructor is such a good choice in the majority of cases, we must first consider common Java patterns related to the use of constructors:

- the telescoping constructor pattern
- the builder pattern

We will see the problems that these solve and the better alternatives that Kotlin offers.

## Telescoping constructor pattern

The telescoping constructor pattern is nothing more than a set of constructors for different possible sets of arguments:

```
class Pizza {  
    val size: String  
    val cheese: Int  
    val olives: Int  
    val bacon: Int  
  
    constructor(  
        size: String,  
        cheese: Int,  
        olives: Int,  
        bacon: Int  
    ) {  
        this.size = size  
        this.cheese = cheese  
        this.olives = olives  
        this.bacon = bacon  
    }  
  
    constructor(  
        size: String,  
        cheese: Int,  
        olives: Int  
    ) : this(size, cheese, olives, 0)  
  
    constructor(  
        size: String,  
        cheese: Int  
    ) : this(size, cheese, 0)  
  
    constructor(size: String) : this(size, 0)  
}
```

Well, this code doesn't really make any sense in Kotlin because we can use default arguments instead:

```
class Pizza(  
    val size: String,  
    val cheese: Int = 0,  
    val olives: Int = 0,  
    val bacon: Int = 0  
)
```

Default values are not only cleaner and shorter, but their usage is also more powerful than the telescoping constructor. We can specify just `size` and `olives` without mentioning other parameters:

```
val myFavorite = Pizza("L", olives = 3)
```

We can also add another named argument either before or after `olives`:

```
val myFavorite = Pizza("L", olives = 3, cheese = 1)
```

As you can see, default arguments are more powerful than the telescoping constructor because:

- We can define a subset of parameters with the default arguments we want.
- We can provide arguments in any order.
- We can explicitly name arguments to make it clear what each value means.

The last reason is quite important. Think of the following object creation:

```
val villagePizza = Pizza("L", 1, 2, 3)
```

It is short, but is it clear? I bet that even the person who declared the `pizza` class won't remember the positions of the `bacon` and `cheese` parameters. Sure, in an IDE we can see an explanation, but what about those who just scan code or read it on Github? When arguments are unclear, we should explicitly state what their names are using *named arguments*:

```
val villagePizza = Pizza(  
    size = "L",  
    cheese = 1,  
    olives = 2,  
    bacon = 3  
)
```

As you can see, **constructors with default arguments surpass the telescoping constructor pattern**. However, there are more popular construction patterns in Java, one of which is the Builder Pattern.

## Builder pattern

Named parameters and default arguments are not allowed in Java. This is the main reason why Java developers use the builder pattern, which allows them to:

- name parameters,
- specify parameters in any order,
- have default values.

Here is an example of a builder defined in Kotlin:

```
class Pizza private constructor(  
    val size: String,  
    val cheese: Int,  
    val olives: Int,  
    val bacon: Int  
) {  
    class Builder(private val size: String) {  
        private var cheese: Int = 0  
        private var olives: Int = 0  
        private var bacon: Int = 0  
  
        fun setCheese(value: Int): Builder {  
            cheese = value  
            return this  
        }  
  
        fun setOlives(value: Int): Builder {  
            olives = value  
            return this  
        }  
  
        fun setBacon(value: Int): Builder {  
            bacon = value  
            return this  
        }  
  
        fun build() = Pizza(size, cheese, olives, bacon)  
    }  
}
```

With the builder pattern, we can set these parameters as we want by using their names:

```
val myFavorite = Pizza.Builder("L").setOlives(3).build()

val villagePizza = Pizza.Builder("L")
    .setCheese(1)
    .setOlives(2)
    .setBacon(3)
    .build()
```

As we've already mentioned, these advantages of Java's builder pattern are fully satisfied by Kotlin's named and default arguments:

```
val villagePizza = Pizza(
    size = "L",
    cheese = 1,
    olives = 2,
    bacon = 3
)
```

When comparing these two simple usages, you can see the advantages of named parameters over the builder:

- Named parameters are shorter — a constructor or factory method with default arguments is much easier to implement than the builder pattern. It is a time-saver both for the developer who implements this code and for those who read it. This is a significant difference because implementation of the builder pattern can be time-consuming. Builder modifications are harder to apply. For instance, changing the name of a parameter requires not only changing the name of the function used to set it but also the name of the parameter in this function, the body of this function, the internal field used to keep it, the parameter name in the private constructor, etc.
- Named parameters are cleaner — when you want to see how an object is constructed, everything you need is in a single method instead of being scattered around a whole builder class. How are objects held? Do they interact? These are questions that are not so easy to answer when we have a big builder.
- Named parameters offer simpler usage — the primary constructor is a built-in concept. The builder pattern is an artificial concept and therefore requires some knowledge about it. For instance, a developer can easily forget to call the `build` function (or, in other cases, `create`).
- Named parameters have no problems with concurrence — this is a rare problem, but function parameters are always immutable in Kotlin, while properties in most builders are mutable. Therefore, it is harder to implement a thread-safe build function for a builder.

All this doesn't mean that we should always use a constructor instead of a builder, so let's look at some cases where the various advantages of the builder pattern shine.

Builders can require a set of values for a name (`setPositiveButton`, `setNegativeButton`, and `addRoute`), and they allow us to aggregate (`addRoute`):

```
val dialog = AlertDialog.Builder(context)
    .setMessage(R.string.fire_missiles)
    .setPositiveButton(R.string.fire) { d, id ->
        // FIRE MISSILES!
    }
    .setNegativeButton(R.string.cancel) { d, id ->
        // User cancelled the dialog
    }
    .create()

val router = Router.Builder()
    .addRoute(path = "/home", ::showHome)
    .addRoute(path = "/users", ::showUsers)
    .build()
```

To achieve similar behavior with a constructor, we would need to introduce special types to hold more data in a single argument:

```
val dialog = AlertDialog(
    context,
    message = R.string.fire_missiles,
    positiveButtonDescription =
        ButtonDescription(R.string.fire) { d, id ->
            // FIRE MISSILES!
        },
    negativeButtonDescription =
        ButtonDescription(R.string.cancel) { d, id ->
            // User cancelled the dialog
        }
)

val router = Router(
    routes = listOf(
        Route("/home", ::showHome),
        Route("/users", ::showUsers)
```

```
)  
)
```

This notation is generally badly received in the Kotlin community and we tend to prefer a DSL (Domain Specific Language) builder for such cases:

```
val dialog = context.alert(R.string.fire_missiles) {  
    positiveButton(R.string.fire) {  
        // FIRE MISSILES!  
    }  
    negativeButton {  
        // User cancelled the dialog  
    }  
}  
  
val route = router {  
    "/home" directsTo ::showHome  
    "/users" directsTo ::showUsers  
}
```

These kinds of DSL builders are generally preferred over the classic Builder pattern since they give more flexibility and cleaner notation, but it is true that making a DSL is harder. On the other hand, making a builder is already hard. If we decide to invest more time to allow better notation at the cost of a less obvious definition, why not take this one step further? In return, we will have more flexibility and readability. In the next chapter, we talk more about using DSLs for object creation.

Another advantage of the classic Builder pattern is that it can be used as a factory that might be filled partially and passed further. Here is a partially filled example builder that is used to specify a default dialog in our application:

```
fun Context.makeDefaultDialogBuilder() =  
    AlertDialog.Builder(this)  
        .setIcon(R.drawable.ic_dialog)  
        .setTitle(R.string.dialog_title)  
        .setOnCancelListener { it.cancel() }
```

The alternatives I find more suitable in Kotlin are either making a default object and using `copy` to customize its properties, or creating this class using a function with optional parameters.

```
data class DialogConfig(  
    val icon: Int,  
    val title: Int,  
    val onCancelListener: (() -> Unit)?,  
    //...  
)  
  
val defaultDialogConfig = DialogConfig(  
    icon = R.drawable.ic_dialog,  
    title = R.string.dialog_title,  
    onCancelListener = { it.cancel() },  
    //...  
)  
  
// or  
  
fun defaultDialogConfig(  
    val icon: Int = R.drawable.ic_dialog,  
    val title: Int = R.string.dialog_title,  
    val onCancelListener: (() -> Unit)? = { it.cancel() }  
) = DialogConfig(icon, title, onCancelListener, /*...*/)
```

Both options seem to be popular for unit testing. They seem to be a good alternative to the classic Builder pattern, so I don't find the Builder pattern advantageous.

In the end, the classic Builder pattern is rarely the best option in Kotlin. It is sometimes chosen:

- to make code consistent with libraries written in other languages that used the Builder pattern,
- when we design an API to be easily used in other languages that do not support default arguments or DSLs.

Otherwise, we generally prefer either a primary constructor with default arguments or an expressive DSL builder.

## Summary

Creating objects using a primary constructor is the most appropriate approach for the vast majority of objects in our projects. Telescoping constructor patterns should be treated as obsolete in Kotlin. I recommend using default values instead

as they are cleaner, more flexible, and more expressive. The classic builder pattern is rarely a good choice. In simpler cases, we can just use a primary constructor with named arguments; when we need to create a more complex object, we can define a DSL builder.

## Item 34: Consider defining a DSL for complex object creation

A set of Kotlin features used together allows us to make a configuration-like Domain Specific Language (DSL). Such DSLs are useful when we need to define more complex objects or a hierarchical structure of objects. They are not easy to define, but once this has been done they hide boilerplate code and the complexity of our code, therefore a developer can express his or her intentions more clearly.

For instance, Kotlin DSL is a popular way to express both classic HTML and React HTML. This is how this could look:

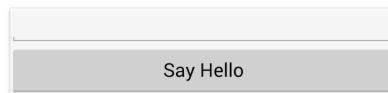
```
body {  
    div {  
        a("https://kotlinlang.org") {  
            target = ATarget.blank  
            +"Main site"  
        }  
    }  
    +"Some content"  
}
```

Main site  
Some content

**View from the above HTML DSL**

Views on other platforms can also be defined using DSLs. Here is a simple Android view defined using the Anko library:

```
verticalLayout {  
    val name = editText()  
    button("Say Hello") {  
        onClick { toast("Hello, ${name.text}!") }  
    }  
}
```

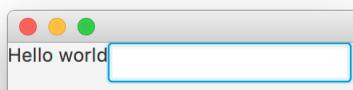


View from the above Android View DSL

It is similar with desktop applications. Here is a view defined in TornadoFX (that is built on top of JavaFX):

```
class HelloWorld : View() {
    override val root = hbox {
        label("Hello world") {
            addClass(heading)
        }

        textfield {
            promptText = "Enter your name"
        }
    }
}
```



View from the above TornadoFX DSL

DSLs are also often used to define data or configurations. Here is an API definition in Ktor which also uses a DSL:

```
fun Routing.api() {
    route("news") {
        get {
            val newsData = NewsUseCase.getAcceptedNews()
            call.respond(newsData)
        }
        get("propositions") {
            requireSecret()
            val newsData = NewsUseCase.getPropositions()
```

```
        call.respond(newsData)
    }
}
// ...
}
```

Here are test case specifications defined in Kotlin Test:

```
class MyTests : StringSpec({
    "length should return size of string" {
        "hello".length shouldBe 5
    }
    "startsWith should test for a prefix" {
        "world" should startWith("wor")
    }
})
```

We can even use Gradle DSL to define Gradle configuration:

```
plugins {
    `java-library`
}

dependencies {
    api("junit:junit:4.12")
    implementation("junit:junit:4.12")
    testImplementation("junit:junit:4.12")
}

configurations {
    implementation {
        resolutionStrategy.failOnVersionConflict()
    }
}

sourceSets {
    main {
        java.srcDir("src/core/java")
    }
}
```

```
java {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    test {
        testLogging.showExceptions = true
    }
}
```

Creating complex and hierarchical data structures is easier with DSLs. Inside these DSLs, we can use everything that Kotlin offers, and we have useful hints as DSLs in Kotlin are fully type-safe (unlike Groovy). It is likely that you have already used some Kotlin DSLs, but it is also important to know how to define them yourself so you can use them better and autonomously.

## Defining your own DSL

To understand how to make your own DSLs, it is important to understand the notion of function types with a receiver. Before that, we'll first briefly review the notion of function types themselves. A function type is a type that represents an object that can be used as a function. For instance, the `filter` function contains a function type to represent a predicate that decides if an element can be accepted or not.

```
inline fun <T> Iterable<T>.filter(
    predicate: (T) -> Boolean
): List<T> {
    val list = arrayListOf<T>()
    for (elem in this) {
        if (predicate(elem)) {
            list.add(elem)
        }
    }
    return list
}
```

Here are a few examples of function types:

- `()->Unit` - Function with no arguments that returns `Unit`.
- `(Int)->Unit` - Function that takes `Int` and returns `Unit`.

- `(Int) -> Int` - Function that takes `Int` and returns `Int`.
- `(Int, Int) -> Int` - Function that takes two arguments of type `Int` and returns `Int`.
- `(Int) -> () -> Unit` - Function that takes `Int` and returns another function. This other function has no arguments and returns `Unit`.
- `((()) -> Unit) -> Unit` - Function that takes another function and returns `Unit`. This other function has no arguments and returns `Unit`.

The basic ways of creating instances of function types are:

- Using lambda expressions
- Using anonymous functions
- Using function references

For instance, think about the following function:

```
fun plus(a: Int, b: Int) = a + b
```

Analogous functions can be created in the following ways:

```
val plus1: (Int, Int) -> Int = { a, b -> a + b }
val plus2: (Int, Int) -> Int = fun(a, b) = a + b
val plus3: (Int, Int) -> Int = Int::plus
```

In the above example, property types are specified, therefore argument types in the lambda expression and in the anonymous function can be inferred. However, it could be the other way around: if we specify the argument types, then the function type can be inferred.

```
val plus4 = { a: Int, b: Int -> a + b }
val plus5 = fun(a: Int, b: Int) = a + b
```

Function types are there to represent objects that represent functions. An anonymous function even looks the same as a normal function but it has no name. A lambda expression is a shorter notation for an anonymous function.

However, if we have function types to represent functions, what about extension functions? Can we express them as well?

```
fun Int.myPlus(other: Int) = this + other
```

It was mentioned before that we create an anonymous function in the same way as a normal function but without a name. So, anonymous extension functions are defined the same way as well:

```
val myPlus = fun Int.(other: Int) = this + other
```

What type does `myPlus` have? The answer is that there is a special type to represent extension functions that is called *function type with a receiver*. It looks similar to a normal function type, but it additionally specifies the receiver type before its arguments, and they are separated using a dot:

```
val myPlus: Int.(Int) -> Int =
    fun Int.(other: Int) = this + other
```

Such a function can be defined using a lambda expression, specifically a lambda expression with receiver, since inside its scope the `this` keyword references the extension receiver (an instance of type `Int` in this case):

```
val myPlus: Int.(Int) -> Int = { this + it }
```

An object created using an anonymous extension function or lambda expression with a receiver can be invoked in 3 ways:

- Like a standard object, using the `invoke` method.
- Like a non-extension function.
- Same as a normal extension function.

```
myPlus.invoke(1, 2)
myPlus(1, 2)
1.myPlus(2)
```

The most important trait of the function type with a receiver is that it changes what `this` refers to. To see how this trait can be used, think of a class that needs to be defined property by property:

```
class Dialog {
    var title: String = ""
    var text: String = ""
    fun show() { /*...*/ }
}

fun main() {
    val dialog = Dialog()
    dialog.title = "My dialog"
    dialog.text = "Some text"
    dialog.show()
}
```

Referencing the dialog repeatedly is not very convenient, but if we were to use a lambda expression with receiver, it would be `this`, and we would be able to just skip it (because a receiver can be used implicitly):

```
class Dialog {  
    var title: String = ""  
    var text: String = ""  
    fun show() { /*...*/  
    }  
}  
  
fun main() {  
    val dialog = Dialog()  
    val init: Dialog.() -> Unit = {  
        title = "My dialog"  
        text = "Some text"  
    }  
    init.invoke(dialog)  
    dialog.show()  
}
```

Following this path, someone might define a function that takes all the common parts of dialog creation and displaying and leaves only the setting of properties to the user:

```
class Dialog {  
    var title: String = ""  
    var text: String = ""  
    fun show() { /*...*/  
    }  
}  
  
fun showDialog(init: Dialog.() -> Unit) {  
    val dialog = Dialog()  
    init.invoke(dialog)  
    dialog.show()  
}  
  
fun main() {  
    showDialog {  
        title = "My dialog"  
        text = "Some text"  
    }  
}
```

```
    }  
}
```

This is our simplest DSL example. Since most of this builder function is repeatable, it has been extracted into an `apply` function that can be used instead of defining a DSL builder for setting properties.

```
inline fun <T> T.apply(block: T.() -> Unit): T {  
    this.block()  
    return this  
}  
  
Dialog().apply {  
    title = "My dialog"  
    text = "Some text"  
}.show()
```

A function type with a receiver is the most basic building block of Kotlin DSLs. Let's create a very simple DSL that allows us to make the following HTML table:

```
fun createTable(): TableBuilder = table {  
    tr {  
        for (i in 1..2) {  
            td {  
                +"This is column $i"  
            }  
        }  
    }  
}
```

Starting from the beginning of this DSL, we can see a function `table`. We are at the top-level without any receivers, so it needs to be a top-level function; however, inside its function argument you can see that we use `tr`. The `tr` function should be allowed only inside the `table` definition. This is why the `table` function argument should have a receiver with such a function. Similarly, the `tr` function argument needs to have a receiver that will contain a `td` function.

```
fun table(init: TableBuilder.() -> Unit): TableBuilder {
    //...
}

class TableBuilder {
    fun tr(init: TrBuilder.() -> Unit) { /*...*/ }
}

class TrBuilder {
    fun td(init: TdBuilder.() -> Unit) { /*...*/ }
}

class TdBuilder
```

How about this statement:

```
+ "This is row $i"
```

What is that? It is only a unary plus operator on a `String`, and it needs to be defined inside `TdBuilder`:

```
class TdBuilder {
    var text = ""

    operator fun String.unaryPlus() {
        text += this
    }
}
```

Now our DSL is well defined. To make it work properly, at every step we need to create a builder and initialize it using a function from the functional parameter (`init` in the example below). Then, the builder will contain all the data specified in this `init` function argument. This is the data we need. Therefore, we can either return this builder, or we can produce another object that holds this data. In this example, we'll just return the builder. This is how the `table` function could be defined:

```
fun table(init: TableBuilder.() -> Unit): TableBuilder {
    val tableBuilder = TableBuilder()
    init.invoke(tableBuilder)
    return tableBuilder
}
```

Notice that we can use the `apply` function, as shown before, to shorten this function:

```
fun table(init: TableBuilder.() -> Unit) =
    TableBuilder().apply(init)
```

Similarly, we can use it in other parts of this DSL to make them more concise:

```
class TableBuilder {
    var trs = listOf<TrBuilder>()

    fun tr(init: TrBuilder.() -> Unit) {
        trs = trs + TrBuilder().apply(init)
    }
}

class TrBuilder {
    var tds = listOf<TdBuilder>()

    fun td(init: TdBuilder.() -> Unit) {
        tds = tds + TdBuilder().apply(init)
    }
}
```

This is a simple (but functional) DSL builder for HTML table creation. It could be improved using a `DslMarker`, as explained in Item 14: Consider referencing receivers explicitly.

## When should we use DSLs?

DSLs give us a way to express any kind of information you want, in a clear and structured way. The problem is that it is never clear to users how this information will be used later. In Anko, TornadoFX, or HTML DSL, we trust that the view will be correctly built based on our definitions, but it is often hard to track exactly how. Some more complicated uses can be hard to discover. The usage of DSLs can

be also confusing to those not used to them, not to mention their maintenance. How they are defined can be a cost in terms of both performance and developer confusion. DSLs are overkill when we can use other simpler features instead. However, they are very useful when we need to express:

- complicated data structures,
- hierarchical structures,
- a huge amount of data.

Everything can be expressed without DSL-like structures by using builders or just constructors instead. **DSLs are about boilerplate elimination of such structures.** You should consider using DSLs when you see repeatable boilerplate code<sup>60</sup> and there are no simpler Kotlin features that can help.

## Summary

A DSL is a special language inside a language. It can make it really simple to create complex objects and even whole object hierarchies, like HTML code or complex configuration files. On the other hand, DSL implementations might be confusing or difficult for new developers. They are also hard to define. This is why they should be only used when they offer real value, such as the creation of really complex objects, or for complex object hierarchies. This is why they are also preferably defined in libraries rather than in projects. It is not easy to make a good DSL, but a well-defined one can make a project much better.

---

<sup>60</sup>Repeatable code that does not contain any important information for a reader.

## Item 35: Consider using dependency injection

Not Kotlin-specific

Basics

One of the most important patterns in modern programming is *dependency injection*. It is a technique that allows you to create objects that depend on other objects without creating these dependencies on your own. This is important because it makes your code more flexible and reusable. In Kotlin, the most popular way to do dependency injection is to use constructor-based dependency injection.

The idea behind constructor-based dependency injection is simple. Instead of creating instances of the classes your class needs to compose (dependencies of this class), you specify the types of these dependencies in the constructor of your class. This way, instead of creating its dependencies on its own, a class receives them from the outside. This is called *inversion of control*. Let me show you an example of a class that creates its dependencies:

```
class UserService {  
    private val userRepository = DatabaseUserRepository()  
    private val emailService = MailchimpEmailService()  
  
    fun registerUser(email: String, password: String) {  
        val user = userRepository.createUser(email, password)  
        emailService.sendEmail(user.email, "Welcome!")  
    }  
}
```

And this is how this class would look if we used constructor-based dependency injection:

```
class UserService(  
    private val userRepository: UserRepository,  
    private val emailService: EmailService  
) {  
    fun registerUser(email: String, password: String) {  
        val user = userRepository.createUser(email, password)  
        emailService.sendEmail(user.email, "Welcome!")  
    }  
}
```

Dependency injection can also be done using setters or property delegates, but I suggest constructor-based dependency injection because it's easier to test and makes dependencies explicit.

The advantages of dependency injection are:

- It makes dependencies explicit. When you look at the constructor of a class, you know what dependencies it has.
- It makes testing easier. You can easily mock dependencies and test your class in isolation.
- It makes your code more flexible. You can easily replace dependencies with other implementations.
- It makes your code more reusable. If you define your dependency with an interface, you can reuse your class in different contexts by providing different implementations of dependencies.

When we use dependency injection, we delegate the actual object creation to outside our active object. Nevertheless, we finally need to define how these dependencies are created. For this, it is popular to use a dependency injection framework: a library that allows you to define how dependencies are created and is then used to create concrete dependencies for you. Consider the following structure of classes:

```
interface DatabaseClient { /* ... */ }
class PostgresDatabaseClient : DatabaseClient { /* ... */ }

interface UserRepository { /* ... */ }
class DatabaseUserRepository(
    private val databaseClient: DatabaseClient
) : UserRepository { /* ... */ }

interface EmailClient { /* ... */ }
class MailchimpEmailClient : EmailClient { /* ... */ }

interface EmailService { /* ... */ }
class MailchimpEmailService(
    private val emailClient: EmailClient
) : EmailService { /* ... */ }

class UserService(
    private val userRepository: UserRepository,
    private val emailService: EmailService
) { /* ... */ }
```

If we used Koin, a popular Kotlin dependency injection framework, this is how we could define how each dependency is created:

```
val userModule = module {
    single<DatabaseClient> { PostgresDatabaseClient() }
    single<UserRepository> { DatabaseUserRepository(get()) }
    single<EmailClient> { MailchimpEmailClient() }
    single<EmailService> { MailchimpEmailService(get()) }
    single { UserService(get(), get()) }
}
```

The created module can be used to initialize and start the dependency injection framework; then, we can get instances of dependencies that are defined all around our project.

```
val userRepo: UserRepository by inject()
val userService: UserService = get()
```

Dependency injection frameworks have several big advantages:

- We need to define only once how components should be created. After that, we can just define what dependencies we need and they are created by our dependency injection framework.
- We can easily replace dependencies with other implementations. We just need to change the definition of how dependencies are created.
- We can easily mock dependencies in tests. We can provide different implementations of dependencies for tests.
- We can easily reuse our classes in different contexts. We can provide different implementations of dependencies for different contexts.
- We can easily create singletons: dependencies that are created only once and then reused.

## Summary

- Dependency injection is a pattern that allows us to create dependencies outside our class.
- Dependency injection makes dependencies explicit, makes testing easier, and makes code more flexible and reusable.
- Dependency injection frameworks allow us to define how dependencies are created and use them all around our project.

# Chapter 6: Class design

Classes are the most important abstraction in the Object-Oriented Programming (OOP) paradigm. Since OOP is the most popular paradigm in Kotlin, classes are also very important for us. This chapter is about class design. It is not about system design, since that would require much more space and there are already many great books on this topic, such as Clean Architecture by Robert C. Martin, or Design Patterns by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. Instead, we will mainly look at the contracts that Kotlin classes are expected to fulfill: how we use Kotlin structures and what is expected from us when we use them. When and how should we use inheritance? How do we expect data classes to be used? When should we use function types instead of interfaces with a single method? What are the contracts of `equals`, `hashCode`, and `compareTo`? When should we use extensions instead of members? These are the kinds of questions we will answer here. They are all important because breaking these contracts might cause serious problems, and following them will help you make your code safer and cleaner.

## Item 36: Prefer composition over inheritance

Not Kotlin-specific

Inheritance is a powerful feature, but it is designed to create a hierarchy of objects with an “is a” relationship. When such a relationship is not clear, inheritance might be problematic and dangerous. When all we need is a simple code extraction or reuse, inheritance should be used with caution; instead, we should prefer a lighter alternative: class composition.

### Simple behavior reuse

Let’s start with a simple problem: we have two classes with partially similar behavior. They should both display a progress bar before some action and hide it afterwards.

```
class ProfileLoader {  
  
    fun load() {  
        // show progress bar  
        // load profile  
        // hide progress bar  
    }  
}  
  
class ImageLoader {  
  
    fun load() {  
        // show progress bar  
        // load image  
        // hide progress bar  
    }  
}
```

In my experience, many developers would extract this common behavior by extracting a common superclass:

```
abstract class LoaderWithProgressBar {

    fun load() {
        // show progress bar
        action()
        // hide progress bar
    }

    abstract fun action()
}

class ProfileLoader : LoaderWithProgressBar() {

    override fun action() {
        // load profile
    }
}

class ImageLoader : LoaderWithProgressBar() {

    override fun action() {
        // load image
    }
}
```

This approach works for such a simple case, but it has important downsides we should be aware of:

- **We can only extend one class.** Extracting functionalities using inheritance often leads to either excessively complex hierarchies of types or to huge BaseXXX classes that accumulate many functionalities.
- **When we extend, we take everything from a class,** which leads to classes that have functionalities and methods they don't need (a violation of the Interface Segregation Principle).
- **Using superclass functionality is much less explicit.** In general, it is a bad sign when a developer reads a method and needs to jump into superclasses many times to understand how this method works.

These are strong reasons that should make us think about an alternative, and a very good one is composition. By composition, we mean holding an object as a property (we compose it) and using its functionalities. This is an example of how we can use composition instead of inheritance to solve our problem:

```
class ProgressBar {
    fun show() {
        /* show progress bar */
    }
    fun hide() {
        /* hide progress bar */
    }
}

class ProfileLoader {
    val progressBar = ProgressBar()

    fun load() {
        progressBar.show()
        // load profile
        progressBar.hide()
    }
}

class ImageLoader {
    val progressBar = ProgressBar()

    fun load() {
        progressBar.show()
        // load image
        progressBar.hide()
    }
}
```

Notice that composition is harder. We need to include the composed object and use it in every single class. This is the key reason why many prefer inheritance. However, this additional code is not useless: it informs the reader **that** a progress bar is used and **how** it is used. It also gives the developer more power over how this progress bar works.

Another thing to note is that composition is better when we want to extract multiple pieces of functionality. For instance, information that loading has finished:

```
class ImageLoader {  
    private val progressBar = ProgressBar()  
    private val finishedAlert = FinishedAlert()  
  
    fun load() {  
        progressBar.show()  
        // load image  
        progressBar.hide()  
        finishedAlert.show()  
    }  
}
```

We cannot extend more than a single class. Therefore, if we wanted to use inheritance instead, we would be forced to place both functionalities in a single superclass. This often leads to a complex hierarchy of the types that are used to add these functionalities. Such hierarchies are very hard to read and often also to modify. Just think about what happens if we need an alert in two subclasses but not in a third one? One way to deal with this problem is to use a parameterized constructor:

```
abstract class InternetLoader(val showAlert: Boolean) {  
  
    fun load() {  
        // show progress bar  
        innerLoad()  
        // hide progress bar  
        if (showAlert) {  
            // show alert  
        }  
    }  
  
    abstract fun innerLoad()  
}  
  
class ProfileLoader : InternetLoader(showAlert = true) {  
  
    override fun innerLoad() {  
        // load profile  
    }  
}  
  
class ImageLoader : InternetLoader(showAlert = false) {
```

```
override fun innerLoad() {  
    // load image  
}  
}
```

This is a bad solution. Having functionality blocked by a flag (`showAlert` in this case) is a bad sign. This problem is compounded when the subclass cannot block other unneeded functionality. This is a trait of inheritance: it takes everything from the superclass, not only what is needed.

## Taking the whole package

When we use inheritance, we take everything from the superclass: methods, expectations (contract), and behavior. Therefore, it is a great tool for representing a hierarchy of objects, but it's not so great when we just want to reuse some common parts. For such cases, composition is better because we can choose the behavior we need. As an example, let's say that in our system we have decided to represent a Dog that can bark and sniff:

```
abstract class Dog {  
    open fun bark() {  
        /*...*/  
    }  
    open fun sniff() {  
        /*...*/  
    }  
}
```

What if then we need to create a robot dog that can bark but can't sniff?

```
class Labrador : Dog()  
  
class RobotDog : Dog() {  
    override fun sniff() {  
        error("Operation not supported")  
        // Do you really want that?  
    }  
}
```

Notice that such a solution violates the interface-segregation principle as `RobotDog` has a method it doesn't need. It also violates the Liskov Substitution Principle as

it breaks superclass behavior. On the other hand, what if your `RobotDog` also needs to be a `Robot` class because `Robot` can calculate (i.e., it has the `calculate` method)? Multiple inheritance is not supported in Kotlin.

```
abstract class Robot {  
    open fun calculate() {  
        /*...*/  
    }  
}  
  
class RobotDog : Dog(), Robot() // Error
```

These are serious design problems and limitations that do not occur when you use composition instead. When we use composition, we choose what we want to reuse. To represent type hierarchy, it is safer to use interfaces, and we can implement multiple interfaces. What has not yet been shown is that inheritance can lead to unexpected behavior.

## Inheritance breaks encapsulation

To some degree, when we extend a class, we depend not only on how it works from the outside but also on how it is implemented inside. This is why we say that inheritance breaks encapsulation. Let's look at an example inspired by the book *Effective Java* by Joshua Bloch. Let's say that we need a set that will know how many elements have been added to it during its lifetime. Such a set can be created using inheritance from `HashSet`:

```
class CounterSet<T> : HashSet<T>() {  
    var elementsAdded: Int = 0  
    private set  
  
    override fun add(element: T): Boolean {  
        elementsAdded++  
        return super.add(element)  
    }  
  
    override fun addAll(elements: Collection<T>): Boolean {  
        elementsAdded += elements.size  
        return super.addAll(elements)  
    }  
}
```

This implementation might look good, but it doesn't work correctly:

```
val counterList = CounterSet<String>()
counterList.addAll(listOf("A", "B", "C"))
print(counterList.elementsAdded) // 6
```

Why is that? The reason is that `HashSet` uses the `add` method under the hood of `addAll`. The counter is then incremented twice for each element added using `addAll`. This problem can be naively solved by removing the custom `addAll` function:

```
class CounterSet<T> : HashSet<T>() {
    var elementsAdded: Int = 0
    private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return super.add(element)
    }
}
```

However, this solution is dangerous. What if the creators of Java decided to optimize `HashSet.addAll` and implement it in a way that doesn't depend on the `add` method? If they did that, this implementation would break with a Java update. Together with this implementation, any other libraries which depend on our current implementation would break as well. The Java creators know this, so they are cautious of making changes to these types of implementations. The same problem affects any library creator or even developers of large projects. So, how can we solve this problem? We should use composition instead of inheritance:

```
class CounterSet<T> {
    private val innerSet = HashSet<T>()
    var elementsAdded: Int = 0
    private set

    fun add(element: T) {
        elementsAdded++
        innerSet.add(element)
    }

    fun addAll(elements: Collection<T>) {
        elementsAdded += elements.size
        innerSet.addAll(elements)
    }
}
```

```
}
```

```
val counterList = CounterSet<String>()
counterList.addAll(listOf("A", "B", "C"))
print(counterList.elementsAdded) // 3
```

One problem is that in this case we lose polymorphic behavior because `CounterSet` is not a `Set` anymore. To keep this behavior, we can use the delegation pattern. The delegation pattern means our class implements an interface, composes an object that implements the same interface, and forwards methods defined in the interface to this composed object. Such methods are called *forwarding methods*. Take a look at the following example:

```
class CounterSet<T> : MutableSet<T> {
    private val innerSet = HashSet<T>()
    var elementsAdded: Int = 0
    private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        elementsAdded += elements.size
        return innerSet.addAll(elements)
    }

    override val size: Int
        get() = innerSet.size

    override fun contains(element: T): Boolean =
        innerSet.contains(element)

    override fun containsAll(elements: Collection<T>):
        Boolean = innerSet.containsAll(elements)

    override fun isEmpty(): Boolean = innerSet.isEmpty()

    override fun iterator() =
        innerSet.iterator()
```

```
override fun clear() =
    innerSet.clear()

override fun remove(element: T): Boolean =
    innerSet.remove(element)

override fun removeAll(elements: Collection<T>):
Boolean = innerSet.removeAll(elements)

override fun retainAll(elements: Collection<T>):
Boolean = innerSet.retainAll(elements)

}
```

The problem now is that we need to implement a lot of forwarding methods (nine, in this case). Thankfully, Kotlin introduced interface delegation support that is designed to help in this kind of scenario. When we delegate an interface to an object, Kotlin will generate all the required forwarding methods during compilation. Here is Kotlin's interface delegation in action:

```
class CounterSet<T>(
    private val innerSet: MutableSet<T> = mutableSetOf()
) : MutableSet<T> by innerSet {

    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        elementsAdded += elements.size
        return innerSet.addAll(elements)
    }

}
```

This is a case in which delegation is a good choice: we need polymorphic behavior and inheritance would be dangerous. However, delegation is not common. In most cases, polymorphic behavior is not needed or we use it in a different way, so composition without delegation is more suitable.

The fact that inheritance breaks encapsulation is a security concern, but in many cases the behavior is specified in a contract or we don't depend on it in subclasses (this is generally true when methods are designed for inheritance). There are other reasons to choose the composition pattern, one of which is that it is easier to reuse and gives us more flexibility.

## Restricting overriding

To prevent developers from extending classes that are not designed for inheritance, we can just keep them final. However, if for some reason we need to allow inheritance, all methods are still final by default. To let developers override them, they must be set to open:

```
open class Parent {  
    fun a() {}  
    open fun b() {}  
}  
  
class Child : Parent() {  
    override fun a() {} // Error  
    override fun b() {}  
}
```

Use this mechanism wisely and open only those methods that are designed for inheritance. Also, remember that when you override a method, you can make it final for all subclasses:

```
open class ProfileLoader : InternetLoader() {  
  
    final override fun load() {  
        // load profile  
    }  
}
```

In this way, you can limit the number of methods that can be overridden in subclasses.

## Summary

There are a few important differences between composition and inheritance:

- **Composition is more secure** - We depend not on how a class is implemented but only on its externally observable behavior.
- **Composition is more flexible** - We can extend only a single class but we can compose many. When we inherit, we take everything; but when we compose, we can choose what we need. When we change the behavior of a superclass, we change the behavior of all subclasses. It is hard to change the behavior of only some subclasses. When a class we have composed changes, it will only change our behavior if it has changed its contract with the outside world.
- **Composition is more explicit** - This is both an advantage and a disadvantage. When we use a method from a superclass, we can do so implicitly, like methods from the same class. This requires less work, but it can be confusing and is more dangerous as it is easy to confuse where a method comes from (is it from the same class, a superclass, the top level, or is it an extension?). When we call a method on a composed object, we know where it comes from.
- **Composition is more demanding** - We need to use a composed object explicitly. When we add some functionalities to a superclass, we often do not need to modify the subclasses. When we use composition, we more often need to adjust usages.
- **Inheritance gives us strong polymorphic behavior** - This is also a double-edged sword. On one hand, it is convenient that a dog can be treated like an animal. On the other hand, it is very constraining: it must be an animal. Every subclass of an animal should be consistent with animal behavior. The superclass defines the contract, and the subclasses should respect it.

It is a general OOP rule to prefer composition over inheritance, but Kotlin encourages composition even more by making all classes and methods final by default and by making interface delegation a first-class citizen. This makes this rule even more important in Kotlin projects.

So, when is inheritance more reasonable? The rule of thumb is that **we should use inheritance when there is a definite “is a” relationship**. In other words, every class that uses inheritance needs to **be** its superclass. All unit tests written for superclasses should also pass for their subclasses, and every usage in production code should be exchangeable (Liskov substitution principle). Object-oriented frameworks for displaying views are good examples: Application in JavaFX, Activity in Android, UIViewController in iOS, and React.Component in React. The same is true when we define our own special kind of view element that always has the same set of functionalities and characteristics. Just remember to design these classes with inheritance in mind, and specify how inheritance should be used. Also, all the methods that are not designed for inheritance should be kept final.

## Item 37: Use the data modifier to represent a bundle of data

In modern projects, we almost solely operate on only two kinds of objects:

- Active objects, like services, controllers, repositories, etc. Such classes don't need to override any methods from `Any` because the default behavior is perfect for them. Each such object is considered unique because even if two accidentally have the same state, this state changes independently, so we don't need to override `equals` and `hashCode`. We don't want to expose such objects' inner state in an uncontrolled way, so they don't need to override `toString`.
- Data model class objects, which represent bundles of data. For such objects, we use the `data` modifier, which overrides the `toString`, `equals`, and `hashCode` methods. It makes two objects with the same data (the same primary constructor properties) equal. It also makes the `toString` method display the name of the class and the values and names of all primary constructor properties. It also makes the `hashCode` method coherent with `equals`. The `data` modifier also implements the `copy` and `componentN` (`component1`, `component2`, etc.) methods for convenience of modifying and destructuring such objects.

Let's start from a short overview of the methods that the `data` modifier overrides.

### The methods that `data` modifier overrides

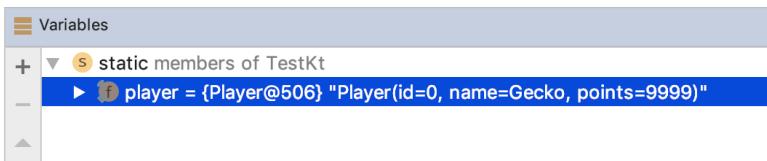
When we add the `data` modifier, it generates the following methods:

- `toString`
- `equals` and `hashCode`
- `copy`
- `componentN` (`component1`, `component2`, etc.)

Let's discuss them in turn.

`toString` displays the name of the class and the values and names of all primary constructor properties. It is useful for logging and debugging.

```
print(player) // Player(id=0, name=Gecko, points=9999)
```



`equals` checks if all primary constructor properties are equal. `hashCode` is coherent with it (see 41: Respect the contract of `hashCode`).

```
player == Player(0, "Gecko", 9999) // true
player == Player(0, "Ross", 9999) // false
```

`copy` is especially useful for immutable data classes. It creates a new object where each primary constructor's properties have the same value by default, but each of them can be changed using named arguments.

```
val newObj = player.copy(name = "Thor")
print(newObj) // Player(id=0, name=Thor, points=9999)
```

This is what `copy` would look like for the class `Person` if we wrote it ourselves:

```
// This is how `copy` generated by `data` modifier
// for `Person` class looks like under the hood
fun copy(
    id: Int = this.id,
    name: String = this.name,
    points: Int = this.points
) = Person(id, name, points)
```

Notice that the `copy` method makes a shallow copy of an object, but this is not a problem when the object is immutable as we do not need deep copies for such objects.

`componentN` functions (`component1`, `component2`, etc.) allow position-based destructuring, as in the example below:

```
val (id, name, pts) = player
```

Destructuring in Kotlin translates directly into variable definitions using the `componentN` functions, so this is what the code above will be compiled to under the hood:

```
// After compilation
val id: Int = player.component1()
val name: String = player.component2()
val pts: Int = player.component3()
```

These are currently all the functionalities that the `data` modifier provides. Don't use it if you don't need `toString`, `equals`, `hashCode`, `copy` or destructuring. If you need some of these functionalities for a class representing a bundle of data, use the `data` modifier instead of implementing the methods yourself.

## When and how should we use destructuring?

Kotlin currently provides only position-based property destructuring, that has pros and cons. The biggest advantage is that we can name variables however we want. We can also destructure everything we want as long as it provides `componentN` functions. This includes `List` and `Map.Entry`, both of which have `componentN` functions defined as extensions:

```
val visited = listOf("China", "Russia", "India")
val (first, second, third) = visited
println("$first $second $third")
// China Russia India

val trip = mapOf(
    "China" to "Tianjin",
    "Russia" to "Petersburg",
    "India" to "Rishikesh"
)
for ((country, city) in trip) {
    println("We loved $city in $country")
    // We loved Tianjin in China
    // We loved Petersburg in Russia
    // We loved Rishikesh in India
}
```

On the other hand, position-based destructuring is dangerous. We need to adjust every destructuring when the order or number of elements in a data class changes. When we use this feature, it is very easy to introduce errors into our code by changing the order of the primary constructor's properties.

```
data class FullName(  
    val firstName: String,  
    val secondName: String,  
    val lastName: String  
)  
  
val elon = FullName("Elon", "Reeve", "Musk")  
val (name, surname) = elon  
print("It is $name $surname!") // It is Elon Reeve!
```

We need to be careful with destructuring. It is useful to use the same names as data class primary constructor properties. In the case of an incorrect order, an IntelliJ/Android Studio warning will be shown. It might be even useful to upgrade this warning to an error.

```
data class FullName(  
    val firstName: String,  
    val secondName: String,  
    val lastName: String  
)  
  
val elon = FullName("Elon", "Reeve", "Musk")  
val (firstName, lastName) = elon  
print("++ is ${firstName}/$lastName") // ++ is Elon Reeve!  
Variable name 'lastName' matches the name of a different component more... (⌘F1)
```

Do not destructure to get just the first value as this might be really confusing and misleading for anyone who will read your code in the future, especially when you destructure in lambda expressions.

```
data class User(val name: String)  
  
fun main() {  
    val user = User("John")  
  
    // Don't do that  
    val (name) = user  
    print(name) // John  
  
    user.let { a -> print(a) } // User(name=John)  
    // Don't do that  
    user.let { (a) -> print(a) } // John  
}
```

Destructuring a single value in lambda is very confusing, especially since parentheses around arguments in lambda expressions are either optional or required in some languages.

## Prefer data classes instead of tuples

Data classes offer more than what is generally provided by tuples. Historically, they replaced tuples in Kotlin since they are considered better practice<sup>61</sup>. The only tuples that are left are `Pair` and `Triple`, but they are data classes under the hood:

```
public data class Pair<out A, out B>(
    public val first: A,
    public val second: B
) : Serializable {

    public override fun toString(): String =
        "($first, $second)"
}

public data class Triple<out A, out B, out C>(
    public val first: A,
    public val second: B,
    public val third: C
) : Serializable {

    public override fun toString(): String =
        "($first, $second, $third)"
}
```

These tuples remained because they are very useful for local purposes, like:

- When we immediately name values:

---

<sup>61</sup>Kotlin had support for tuples when it was still in the beta version. We were able to define a tuple by brackets and a set of types, like `(Int, String, String, Long)`. What we achieved behaved the same as data classes in the end, but it was far less readable. Can you guess what type this set of types represents? It can be anything. Using tuples is tempting, but using data classes is nearly always better. This is why tuples were removed and only `Pair` and `Triple` are left.

```
val (description, color) = when {
    degrees < 5 -> "cold" to Color.BLUE
    degrees < 23 -> "mild" to Color.YELLOW
    else -> "hot" to Color.RED
}
```

- To represent an aggregate not known in advance, as is commonly found in standard library functions:

```
val (odd, even) = numbers.partition { it % 2 == 1 }
val map = mapOf(1 to "San Francisco", 2 to "Amsterdam")
```

In other cases, we prefer data classes. Take a look at an example: let's say that we need a function that parses a full name into a name and a surname. One might represent this name and surname as a `Pair<String, String>`:

```
fun String.parseName(): Pair<String, String>? {
    val indexOfLastSpace = this.trim().lastIndexOf(' ')
    if (indexOfLastSpace < 0) return null
    val firstName = this.take(indexOfLastSpace)
    val lastName = this.drop(indexOfLastSpace)
    return Pair(firstName, lastName)
}

// Usage
val fullName = "Marcin Moskała"
val (firstName, lastName) = fullName.parseName() ?: return
```

The problem is that when someone reads this code, it is not clear that `Pair<String, String>` represents a full name. What is more, it is not clear what the order of the values is, therefore someone could think that the surname goes first:

```
val fullName = "Marcin Moskała"
val (lastName, firstName) = fullName.parseName() ?: return
print("His name is $firstName") // His name is Moskała
```

To make usage safer and the function easier to read, we should use a data class instead:

```
data class FullName(  
    val firstName: String,  
    val lastName: String  
)  
  
fun String.parseName(): FullName? {  
    val indexOfLastSpace = this.trim().lastIndexOf(' ')  
    if (indexOfLastSpace < 0) return null  
    val firstName = this.take(indexOfLastSpace)  
    val lastName = this.drop(indexOfLastSpace)  
    return FullName(firstName, lastName)  
}  
  
// Usage  
val fullName = "Marcin Moskała"  
val (firstName, lastName) = fullName.parseName() ?: return
```

It costs nearly nothing and improves the function significantly:

- The return type of this function is more clear.
- The return type is shorter and easier to pass forward.
- If a user destructures to variables with correct names but in incorrect positions, a warning will be displayed.

If you don't want this class in a wider scope, you can restrict its visibility. It can even be private if you need to use it for some local processing only in a single file or class. It is worth using data classes instead of tuples. Classes are cheap in Kotlin, so don't be afraid to use them in your projects.

## Summary

- Use `data` modifier for classes that are used to represent a bundle of data.
- Be careful with destructuring, and when you do that, prefer matching the variable name with the property name.
- Prefer data classes instead of tuples. Defining a data class costs little, and it makes the code more readable and less error-prone.

## Item 38: Use function types or functional interfaces to pass operations and actions

Many languages do not have the concept of a function type. Instead, they use interfaces with a single method. Such interfaces are known as SAMs (Single-Abstract Method). Here is an example SAM that is used to pass information about what should happen when a view is clicked:

```
interface OnClickListener {  
    fun onClick(view: View)  
}
```

When a function expects a SAM, we must pass an instance of an object that implements this interface<sup>62</sup>.

```
fun setOnClickListener(listener: OnClickListener) {  
    // ...  
}  
  
setOnClickListener(object : OnClickListener {  
    override fun onClick(view: View) {  
        // ...  
    }  
})
```

However, Kotlin supports two mechanisms that give us more freedom:

- the function type,
- the functional interface.

---

<sup>62</sup>Unless it is a Java SAM: in such cases, there is special support, and we can pass a function type instead.

```
// Function type usage
fun setOnClickListener(listener: (View) -> Unit) {
    //...
}

// Functional interface declaration
fun interface OnClickListener {
    fun onClick(view: View)
}

// Functional interface usage
fun setOnClickListener(listener: OnClickListener) {
    //...
}
```

When we use either of these two, the argument can be defined as:

- A lambda expression or an anonymous function.

```
setOnClickListener { /*...*/ }
setOnClickListener(fun(view) { /*...*/ })
```

- A function reference or a bounded function reference.

```
setOnClickListener(::println)
setOnClickListener(this::showUsers)
```

- Objects that implement the declared function type or a functional interface.

```
class ClickListener : (View) -> Unit {
    override fun invoke(view: View) {
        // ...
    }
}

setOnClickListener(ClickListener())
```

Both function types and functional interfaces allow the above usages, but in general we consider function types as the standard way to represent operations and actions as objects.

## Using function types with type aliases

As we said already, function types are the standard way to represent operations and actions as objects. If you want to name a concrete function type, you can use a type alias.

```
typealias OnClick = (View) -> Unit
```

A type alias provides another name for a type that is like a nickname. No matter whether you use someone's full name or their nickname, you still mean the same person. It's the same with type aliases: during compilation they are replaced with the type they represent.

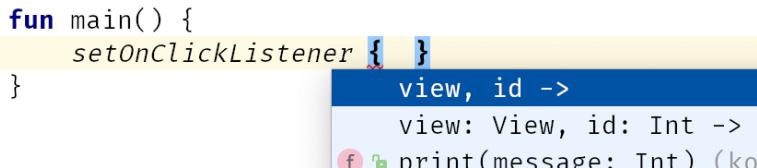
Type aliases can also be generic:

```
typealias OnClick<T> = (T) -> Unit
```

Function type parameters can be named. The advantage of naming them is that these names can then be suggested by default by the IDE. However, when we start naming parameters, the types tend to get longer. This is why we often use this feature together with type aliases.

```
typealias OnClick = (view: View) -> Unit

fun setOnClickListener(listener: OnClick) { /*...*/ }
```



```
fun main() {
    setOnClickListener { }
}
```

## Reasons to use functional interfaces

A functional interface is a heavier solution. Such interfaces need to be defined, but in return:

- they define a new named type,
- handler functions can be named differently (in a function type, handler name is always `invoke`),
- interoperability with other languages is better.

```
interface SwipeListener {
    fun onSwipe()
}

fun interface FlingListener {
    fun onFling()
}

fun setOnClickListener(listener: SwipeListener) {
    // when swipe happens
    listener.onSwipe()
}

fun main() {
    val onSwipe = SwipeListener { println("Swiped") }
    setOnClickListener(onSwipe) // Swiped

    val onFling = FlingListener { println("Touched") }
    setOnClickListener(onFling) // Error: Type mismatch
}
```

Functional interfaces also allow non-abstract functions to be added and other interfaces to be implemented.

```
interface ElementListener<T> {
    fun invoke(element: T)
}

fun interface OnClickListener : ElementListener<View> {
    fun onClick(view: View)

    override fun invoke(element: View) {
        onClick(element)
    }
}
```

When we design a class to be used from a language other than Kotlin, interfaces are cleaner and better supported. These other languages cannot see type aliases or have name suggestions. What is more, Kotlin function types need to return something, at least `Unit`, and returning `Unit` must be explicit in Java:

```
// Kotlin
class CalendarView() {
    var onDateClicked: ((date: Date) -> Unit)? = null
    var onPageChanged: OnDateClicked? = null
}

fun interface OnDateClicked {
    fun onClick(date: Date)
}

// Java
CalendarView c=new CalendarView();
c.setOnDateClicked(date->Unit.INSTANCE);
c.setOnPageChanged(date->{});
```

Another advantage of functional interfaces is that they are not wrapping types. Since a function type is a generic type under the hood, primitives cannot be used. This means that a parameter of type `Int` in Java will be interpreted as `Integer` instead of `int`. This can sometimes make a difference, as explained in Item 47: *Avoid unnecessary object creation*. Functional interfaces do not have this problem.

Overall, the main reasons to prefer functional interfaces are:

- Java interoperability,
- optimization for primitive types,
- to help us represent not merely a function but an interface with a contract.

When you don't need any of these, use function types instead of functional interfaces.

## Avoid expressing actions using interfaces with multiple abstract methods

Another practice that can be observed among developers who switched to Kotlin from Java is expressing actions using interfaces with multiple abstract methods:

```
class CalendarView {  
    var listener: Listener? = null  
  
    interface Listener {  
        fun onDateClicked(date: Date)  
        fun onPageChanged(date: Date)  
    }  
}
```

This pattern was popular in Java when functional interfaces weren't supported. I believe this is largely a result of laziness. From an API consumer's point of view, it is better to define them as separate properties containing either function types or functional interfaces:

```
// Using functional interfaces  
class CalendarView {  
    var onDateClicked: OnDateClicked? = null  
    var onPageChanged: OnPageClicked? = null  
}  
  
// Using function types  
class CalendarView {  
    var onDateClicked: ((date: Date) -> Unit)? = null  
    var onPageChanged: ((date: Date) -> Unit)? = null  
}
```

In this way, the implementations of `onDateClicked` and `onPageChanged` do not need to be tied together in an interface. Now, these functions may be changed independently, and we can use function literals (e.g., lambda expressions) to set them.

## Summary

- To express behavior, prefer function types or functional interfaces instead of standard interfaces or abstract classes.
- Function types are used more often. When they are used multiple times or are getting too long, we hide them behind type aliases.
- Functional interfaces are preferred primarily for Java (or other languages) interoperability and for more complicated cases when what we want to express is more than just an arbitrary function.

## Item 39: Use sealed classes and interfaces to express restricted hierarchies

Sealed classes and interfaces represent restricted hierarchies, which means hierarchies with a concrete set of classes that is known in advance. Great examples are `BinaryTree` (which is either `Leaf` or `Node`) and `Either` (which is either `Left` or `Right`). Such classes or interfaces have a concrete subset of children that will never change.

```
sealed interface BinaryTree
class Leaf(val value: Any?) : BinaryTree
class Node(val left: Tree, val right: Tree) : BinaryTree

sealed interface Either<out L, out R>
class Left<L>(val value: L) : Either<L, Nothing>
class Right<R>(val value: R) : Either<Nothing, R>
```

Sealed interfaces were introduced in Kotlin 1.5. Before that version, we had to use sealed classes whose subclasses had to be defined in the same file.

Sealed classes are also used to represent hierarchies that might change in the future but are considered final now, such as a set of operations we support, or the messages an actor accepts.

```
sealed class ValueChange<out T>
object Keep: ValueChange<Nothing>()
object SetDefault: ValueChange<Nothing>()
object SetEmpty: ValueChange<Nothing>()
class Set<out T>(val value: T): ValueChange<T>()

sealed class ManagerMessage
class CodeProduced(val code: String): ManagerMessage()
object ProductionStopped: ManagerMessage()

sealed interface AdView
object FacebookAd: AdView
object GoogleAd: AdView
class OwnAd(val text: String, val imgUrl: String): AdView
```

Notice that when a class does not hold any data, we use an object declaration. This is an optimization as it means we have one instance for all usages. Both creation and comparison are easier.

One might say that a sealed class or interface is a Kotlin way of expressing union types (sum types or coproducts) — a set of alternatives. For instance, `Either` is either `Left` or `Right`, but it is never both.

Sealed classes are abstract classes, but because of their characteristics they also have some additional restrictions on their subclasses:

- they need to be defined in the same package and module where the sealed class is defined,
- they can't be local or anonymous objects.

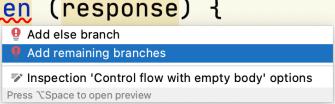
This means that when you use the `sealed` modifier, you control which subclasses a class or interface has. The clients of your library or module cannot add their own. No one can quietly add a local class or object expression that extends this class. The hierarchy of subclasses is restricted.

## Sealed classes and `when` expressions

Using `when` as an expression forces us to return a value for every branch. In most cases, the only way to cover all possibilities is to specify an `else` clause. The power of having a finite type as an argument makes it possible to have an exhaustive `when` with a branch for every possible value, which with sealed classes means `is` checks for all possible subtypes. Also, IntelliJ automatically suggests adding the remaining branches. These two make sealed classes very convenient to check for all possible types.

```
sealed class Result<out V>
data class Success<V>(val value: V) : Result<V>()
data class Failure(val error: Throwable) : Result<Nothing>()

fun handle(response: Result<String>) {
    val text = when (response) {
        }
```



```
sealed class Response<out R>
class Success<R>(val value: R): Response<R>()
class Failure(val error: Throwable): Response<Nothing>()

fun handle(response: Response<String>) {
    val text = when (response) {
        is Success -> "Success with ${response.value}"
        is Failure -> "Error"
        // else is not needed here
    }
    print(text)
}
```

Notice that when the `else` clause is not used, and when we add another subclass of this sealed class, the usage needs to be adjusted by including this new type. This is convenient in local code as it forces us to handle this new class in all exhaustive `when` expressions (so, typically in most cases where we need to). The inconvenient part is that when this sealed class is part of the public API of some library or shared module, adding a subtype is a breaking change.

## Summary

Sealed classes and interfaces should be used to represent restricted hierarchies. They make it easier to handle each possible value and, as a result, to add new methods using extension functions. Abstract classes leave space for new classes to join this hierarchy. If we want to control what the subclasses of a class are, we should use the `sealed` modifier. We use `abstract` mainly when we design for inheritance.

## Item 40: Prefer class hierarchies instead of tagged classes

It is not uncommon in large projects to find classes with a constant “mode” that specifies how this class should behave. We call such classes “tagged” as they contain a tag that specifies their mode of operation. However, there are many problems with them, most of which stem from the fact that different responsibilities from disparate modes fight for space in the same class even though they are generally distinguishable from each other. For instance, in the following snippet we can see a class that is used in tests to check if a value fulfills some criteria. This example is simplified, but it is a real sample from a large project<sup>63</sup>.

```
class ValueMatcher<T> private constructor(
    private val value: T? = null,
    private val matcher: Matcher
) {

    fun match(value: T?) = when (matcher) {
        Matcher.EQUAL -> value == this.value
        Matcher.NOT_EQUAL -> value != this.value
        Matcher.LIST_EMPTY ->
            value is List<*> && value.isEmpty()
        Matcher.LIST_NOT_EMPTY ->
            value is List<*> && value.isNotEmpty()
    }
}

enum class Matcher {
    EQUAL,
    NOT_EQUAL,
    LIST_EMPTY,
    LIST_NOT_EMPTY
}

companion object {
    fun <T> equal(value: T) =
        ValueMatcher<T>(
            value = value,
            matcher = Matcher.EQUAL
        )
}
```

---

<sup>63</sup>The full version contained many more modes.

```
fun <T> notEqual(value: T) =
    ValueMatcher<T>(
        value = value,
        matcher = Matcher.NOT_EQUAL
    )

fun <T> emptyList() =
    ValueMatcher<T>(
        matcher = Matcher.LIST_EMPTY
    )

fun <T> notEmptyList() =
    ValueMatcher<T>(
        matcher = Matcher.LIST_NOT_EMPTY
    )
}
```

There are many downsides to this approach:

- Additional boilerplate code is needed because multiple modes are handled in a single class.
- Properties are inconsistently used as they are used for different purposes. An object generally has more properties than most modes need as these properties might be required by other modes. For instance, in the example above, `value` is not used when the mode is `LIST_EMPTY` or `LIST_NOT_EMPTY`.
- It is hard to protect state consistency and correctness when elements have multiple purposes and can be defined in a few ways.
- The use of a factory method is often required because otherwise it is hard to ensure that objects are created correctly.

Instead of tagged classes, we have a better alternative in Kotlin: sealed classes. Instead of accumulating multiple modes in a single class, we should define multiple classes for each mode and use the type system to allow their polymorphic use. Then, the additional `sealed` modifier seals these classes as a set of alternatives. Here is how this could have been implemented:

```

sealed class ValueMatcher<T> {
    abstract fun match(value: T): Boolean

    class Equal<T>(val value: T) : ValueMatcher<T>() {
        override fun match(value: T): Boolean =
            value == this.value
    }

    class NotEqual<T>(val value: T) : ValueMatcher<T>() {
        override fun match(value: T): Boolean =
            value != this.value
    }

    class EmptyList<T>() : ValueMatcher<T>() {
        override fun match(value: T) =
            value is List<*> && value.isEmpty()
    }

    class NotEmptyList<T>() : ValueMatcher<T>() {
        override fun match(value: T) =
            value is List<*> && value.isNotEmpty()
    }
}

```

This implementation is much cleaner as no multiple responsibilities are tangled with each other. Each object has only the required data and can define which parameters it expects. Using a type hierarchy makes it possible to overcome all the shortcomings of tagged classes. We can also easily add methods as extension functions using `when`.

```

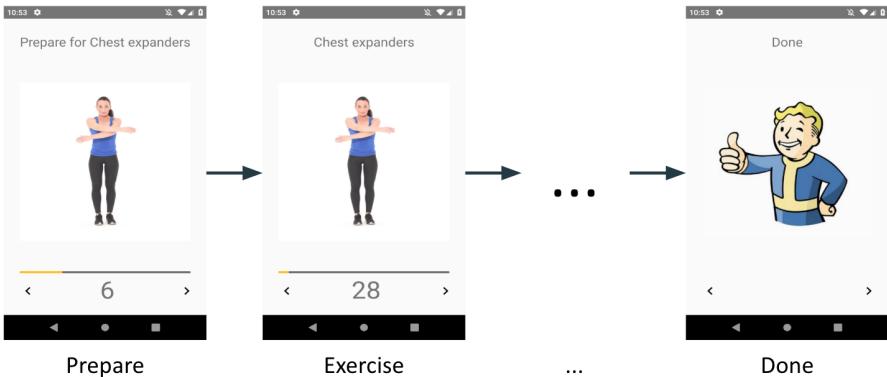
fun <T> ValueMatcher<T>.reversed(): ValueMatcher<T> =
    when (this) {
        is EmptyList -> NotEmptyList<T>()
        is NotEmptyList -> EmptyList<T>()
        is Equal -> NotEqual(value)
        is NotEqual -> Equal(value)
    }

```

## Tagged classes are not the same as classes using the state pattern

There is a popular pattern that is often confused with tagged classes. State pattern is a behavioral software design pattern that allows an object to alter its behavior

when its internal state changes. This pattern is often used in front-end controllers, presenters, or view models (from MVC, MVP, and MVVM architectures). For instance, let's say that you write an application that guides the user through morning exercises. Before every exercise, there is preparation time; at the end, there is a screen that states that the exercises are finished.



When we use the state pattern, we have a hierarchy of classes that represents different states, and we have a read-write property that we need in order to represent which state is the current one:

```
sealed class WorkoutState

class PrepareState(
    val exercise: Exercise
) : WorkoutState()

class ExerciseState(
    val exercise: Exercise
) : WorkoutState()

object DoneState : WorkoutState()

fun List<Exercise>.toStates(): List<WorkoutState> =
    flatMap { exercise ->
        listOf(
            PrepareState(exercise),
            ExerciseState(exercise)
        )
    } + DoneState
```

```
class WorkoutPresenter( /*...*/ ) {
    private var state: WorkoutState = states.first()

    //...
}
```

The essential difference is that the classes that have state are mutable, and keep their state in other classes. Those other classes can be tagged classes, but we prefer to use sealed classes instead. As you can see, when we use the state pattern, this item still applies; we prefer to use sealed classes instead of tagged classes. The class that keep state typically has more functionalities, and their behavior often depends on this state.

## Summary

In Kotlin, we use type hierarchies instead of tagged classes. We most often represent these type hierarchies as sealed classes. This practice does not collide with the state pattern, which is a popular and useful pattern in Kotlin. Both patterns actually cooperate as we prefer to use sealed hierarchies instead of tagged classes when we implement a state pattern. This is especially visible when we implement complex yet separable states in a single view.

## Item 41: Use enum to represent a list of values

When we have to represent a constant set of possible options, a classic choice is to use an enum. For instance, if our website offers a concrete set of payment methods, we can represent them in our service using the following enum class:

```
enum class PaymentOption {  
    CASH,  
    CARD,  
    TRANSFER  
}
```

Since each enum is an instance of the enum class, it can hold values that are always item-specific:

```
import java.math.BigDecimal  
  
enum class PaymentOption {  
    CASH,  
    CARD,  
    TRANSFER;  
  
    // Do not define mutable state in enum values  
    var commission: BigDecimal = BigDecimal.ZERO  
}  
  
fun main() {  
    val c1 = PaymentOption.CARD  
    val c2 = PaymentOption.CARD  
    print(c1 == c2) // true,  
    // because it is the same object  
  
    c1.commission = BigDecimal.TEN  
    print(c2.commission) // 10  
    // because c1 and c2 point to the same item  
  
    val t = PaymentOption.TRANSFER  
    print(t.commission) // 0,  
    // because commission is per-item  
}
```

It is a bad practice to define mutable variables in enum classes, because they are static for each item, so this way we create a global static mutable state (see Item 1: Limit mutability). However, this functionality is often used to attach some constant values to each item. These constant values can be attached during the creation of each item using the primary constructor:

```
import java.math.BigDecimal

enum class PaymentOption(val commission: BigDecimal) {
    CASH(BigDecimal.ONE),
    CARD(BigDecimal.TEN),
    TRANSFER(BigDecimal.ZERO)
}

fun main() {
    println(PaymentOption.CARD.commission) // 10
    println(PaymentOption.TRANSFER.commission) // 0

    val paymentOption: PaymentOption =
        PaymentOption.values().random()
    println(paymentOption.commission) // 0, 1 or 10
}
```

Kotlin enums can even have methods whose implementations are also item-specific. When we define them, the enum class itself needs to define an abstract method, and each item must override it:

```
enum class PaymentOption {
    CASH {
        override fun startPayment(
            transaction: Transaction
        ) {
            showCashPaymentInfo(transaction)
        }
    },
    CARD {
        override fun startPayment(
            transaction: Transaction
        ) {
            moveToCardPaymentPage(transaction)
        }
    },
}
```

```
TRANSFER {
    override fun startPayment(  
        transaction: Transaction  
    ) {  
        showMoneyTransferInfo()  
        setupPaymentWatcher(transaction)  
    }  
};  
  
abstract fun startPayment(transaction: Transaction)  
}
```

However, this option is rarely used because it is more convenient to use a primary constructor parameter of functional type:

```
enum class PaymentOption(  
    val startPayment: (Transaction) -> Unit  
) {  
    CASH(::showCashPaymentInfo),  
    CARD(::moveToCardPaymentPage),  
    TRANSFER{  
        showMoneyTransferInfo()  
        setupPaymentWatcher(it)  
    })  
}
```

An even more convenient option is to define an extension function. Notice that when we use enum values in a when expression, we do not need to add an else clause when we cover all the values.

```
enum class PaymentOption {  
    CASH,  
    CARD,  
    TRANSFER  
}  
  
fun PaymentOption.startPayment(transaction: Transaction) {  
    when (this) {  
        CASH -> showCashPaymentInfo(transaction)  
        CARD -> moveToCardPaymentPage(transaction)  
        TRANSFER -> {  
    }  
}
```

```
        showMoneyTransferInfo()
        setupPaymentWatcher(transaction)
    }
}
```

The power of enum is that its items are specific and constant. We can get all the items using this enum's companion object's `values()` function or the top-level `enumValueOf` function. We can also read enum from a `String` using its companion object's `valueOf(String)` or top-level `enumValueOf(String)`. All enums are a subtype of `Enum<T>`.

```
enum class PaymentOption {
    CASH,
    CARD,
    TRANSFER
}

inline fun <reified T : Enum<T>> printEnumValues() {
    for (value in enumValues<T>()) {
        println(value)
    }
}

fun main() {
    val options = PaymentOption.values()
    println(options.map { it.name })
    // [CASH, CARD, TRANSFER]

    val options2: Array<PaymentOption> =
        enumValues<PaymentOption>()
    println(options2.map { it.name })
    // [CASH, CARD, TRANSFER]

    val option: PaymentOption =
        PaymentOption.valueOf("CARD")
    println(option) // CARD

    val option2: PaymentOption =
        enumValueOf<PaymentOption>("CARD")
    println(option2) // CARD
```

```
printEnumValues<PaymentOption>()
// CASH
// CARD
// TRANSFER
}
```

Iterating over enum values is easy. All enums have the `ordinal` property and implement the `Comparable` interface. Enums also automatically implement `toString`, `hashCode` and `equals`. Their serialization and deserialization are simple (they are represented just by name), efficient, and automatically supported by most libraries for serialization (like Moshi, Gson, Jackson, Kotlinx Serialization, etc). As a result, enums are perfect for representing a concrete set of constant values. But how do they compare to sealed classes?

## Enum or a sealed class?

As shown in Item 39: Use sealed classes and interfaces to express restricted hierarchies; sealed classes and interfaces can also represent a set of values, but all their subclasses have to be object declarations.

```
sealed class PaymentOption
object Cash : PaymentOption()
object Card : PaymentOption()
object Transfer : PaymentOption()
```

For just a set of values, enum should be preferred: sealed classes cannot be automatically serialized or deserialized; they are not so easy to iterate over (although we can do it with reflection); and they do not have a natural order. However, there are some cases when we might choose sealed classes with object subclasses anyway: classes can keep values, so if we think we might need to transform objects into classes at some point, we should prefer sealed classes. For instance, if we define messages to some actor, we should use sealed classes even if all the messages are now object declarations. This is because it is likely we will need to transform some of these objects into classes in the future.

```
sealed class ManagerMessage()
object ProductionStarted : ManagerMessage()
object ProductionStopped : ManagerMessage()

// In the future we might add something like this:
class Alert(val message: String) : ManagerMessage()

// or we might need to add data to an existing message
class ProductionStarted(
    val time: DateTime
) : ManagerMessage()
```

When messages need to hold data, it is a clear-cut case: we need classes, not merely an enum. This is how our payment types, which hold the data required to make transactions, could be represented:

```
sealed class Payment

data class CashPayment(
    val amount: BigDecimal,
    val pointId: Int
) : Payment()

data class CardPayment(
    val amount: BigDecimal,
    val pointId: Int
) : Payment()

data class BankTransfer(
    val amount: BigDecimal,
    val pointId: Int
) : Payment()

fun process(payment: Payment) {
    when (payment) {
        is CashPayment -> {
            showPaymentInfo(
                payment.amount,
                payment.pointId
            )
        }
    }
}
```

```
        }
    }

    is CardPayment -> {
    moveToCardPaiment(
        payment.amount,
        payment.orderId
    )
}

is BankTransfer -> {
    val transferRepo = BankTransferRepo()
    val transferDetails = transferRepo.getDetails()
    displayTransferInfo(
        payment.amount,
        transferDetails
    )
    transferRepo.setUpPaymentWathcher(
        payment.orderId,
        payment.amount,
        transferDetails
    )
}
}

}
```

## Summary

- The advantage of enum classes is that they can be serialized and deserialized out of the box. They have the companion object methods `values()` and `valueOf(String)`. We can also get enum values by the type using the `enumValues()` and `enumValueOf(String)` functions. Each enum value has `ordinal`, and we can hold per-item data. They are perfect for representing a constant set of possible values.
- Enum classes represent a concrete set of values, while sealed classes represent a concrete set of classes. Since these classes can be object declarations, we can use sealed classes to a certain degree instead of enums, but not the other way around.

## Item 42: Respect the contract of equals

In Kotlin, every object extends `Any`, which has a few methods with well-established contracts. These methods are:

- `equals`
- `hashCode`
- `toString`

Their contract is described in their comments and is elaborated in the official documentation. As I described in Item 31: Respect abstraction contracts, every subtype of a type with a contract should respect this contract. The aforementioned methods have an important position in Kotlin as they have been defined since the beginning of Java, therefore many objects and functions depend on their contracts. Breaking these contracts will often cause some objects or functions to not work properly; therefore, in this and the next items we will talk about overriding these functions and about their contracts. Let's start with `equals`.

### Equality

In Kotlin, there are two types of equality:

- Structural equality - checked with the `equals` method or the `==` operator (and its negated counterpart `!=`). `a == b` translates to `a.equals(b)` when `a` is not nullable, otherwise it translates to `a?.equals(b) ?: (b === null)`.
- Referential equality - checked with the `===` operator (and its negated counterpart `!==`); returns `true` when both sides point to the same object.

Since `equals` is implemented in `Any`, which is the superclass of every class, we can check the equality of any two objects. However, using operators to check equality is not allowed when objects are not of the same type:

```
open class Animal
class Book
Animal() == Book() // Error: Operator == cannot be
// applied to Animal and Book
Animal() === Book() // Error: Operator === cannot be
// applied to Animal and Book
```

Objects either need to have the same type, or one needs to be a subtype of another:

```
class Cat : Animal()
Animal() == Cat() // OK, because Cat is a subclass of
// Animal
Animal() === Cat() // OK, because Cat is a subclass of
// Animal
```

This limitation originates in the fact that it does not make sense to check the equality of two objects of different types, as will become clear when we explain the contract of equals.

## Why do we need equals?

The default implementation of equals checks if another object is exactly the same instance, just like the referential equality (==). It means that every object is unique by default:

```
class Name(val name: String)

val name1 = Name("Marcin")
val name2 = Name("Marcin")
val name1Ref = name1

name1 == name1 // true
name1 == name2 // false
name1 == name1Ref // true

name1 === name1 // true
name1 === name2 // false
name1 === name1Ref // true
```

Such behavior is useful for many objects. It is perfect for active elements, like a database connection, a repository, or a thread. However, there are objects for which we need to represent equality differently. A popular alternative is a data class equality that checks if all primary constructor properties are equal:

```
data class FullName(val name: String, val surname: String)

val name1 = FullName("Marcin", "Moskała")
val name2 = FullName("Marcin", "Moskała")
val name3 = FullName("Maja", "Moskała")

name1 == name1 // true
name1 == name2 // true, because data are the same
name1 == name3 // false

name1 === name1 // true
name1 === name2 // false
name1 === name3 // false
```

Such behavior is perfect for classes that are represented by the data they hold, so we often use the data modifier in data model classes or in other data holders.

Notice that data class equality also helps when we need to compare some but not all properties, e.g., when we want to compare everything except for a cache or other redundant properties. Here is an example of an object that represents date and time and has the properties `asStringCache` and `changed`, which should not be compared by equality checking:

```
class DateTime(
    /** The millis from 1970-01-01T00:00:00Z */
    private var millis: Long = 0L,
    private var timeZone: TimeZone? = null
) {
    private var asStringCache = ""
    private var changed = false

    override fun equals(other: Any?): Boolean =
        other is DateTime &&
            other.millis == millis &&
            other.timeZone == timeZone

    //...
}
```

The same can be achieved using a data modifier:

```
data class DateTime(
    private var millis: Long = 0L,
    private var timeZone: TimeZone? = null
) {
    private var asStringCache = ""
    private var changed = false

    // ...
}
```

Just notice that copy in such a case will not copy properties that are not declared in the primary constructor. Such behavior is correct only when these additional properties are truly redundant (the object will behave correctly if they are lost).

Thanks to these two alternatives, namely default and data class equality, **we rarely need to implement equality ourselves in Kotlin**.

An example of when we might need to implement equality is when we would like to compare only one property. For instance, a User class might have an assumption that two users are equal when their id is identical.

```
class User(
    val id: Int,
    val name: String,
    val surname: String
) {
    override fun equals(other: Any?): Boolean =
        other is User && other.id == id

    override fun hashCode(): Int = id
}
```

As you can see, we implement equals ourselves when:

- We need its logic to differ from the default logic.
- We need to compare only a subset of properties.
- We do not want our object to be a data class, or the properties we need to compare are not in the primary constructor.

## The contract of equals

This is how equals is described in its comments (Kotlin 1.9.0, formatted):

Indicates whether some other object is “equal to” this one. Implementations must fulfil the following requirements:

- **Reflexive:** for any non-null value  $x$ ,  $x.equals(x)$  should return true.
- **Symmetric:** for any non-null values  $x$  and  $y$ ,  $x.equals(y)$  should return true if and only if  $y.equals(x)$  returns true.
- **Transitive:** for any non-null values  $x$ ,  $y$ , and  $z$ , if  $x.equals(y)$  returns true and  $y.equals(z)$  returns true, then  $x.equals(z)$  should return true.
- **Consistent:** for any non-null values  $x$  and  $y$ , multiple invocations of  $x.equals(y)$  consistently return true or consistently return false, provided no information used in  $equals$  comparisons on the objects is modified.
- **Never equal to null:** for any non-null value  $x$ ,  $x.equals(null)$  should return false.

Additionally, we expect  $equals$ ,  $toString$  and  $hashCode$  to be fast. This is not a part of the official contract, but it would be highly unexpected to wait a few seconds to check if two elements are equal.

All these requirements are important. They have been assumed since the beginning of the JVM platform, so now many objects depend on these assumptions. To understand this contract well, let’s discuss each of these requirements separately.

**Object equality should be reflexive**, meaning that  $x.equals(x)$  returns true. It sounds obvious, but this can be violated. For instance, someone might want to make a `Time` object that can compare milliseconds as well as represent the current time:

```
// DO NOT DO THIS!
class Time(
    val millisArg: Long = -1,
    val isNow: Boolean = false
) {
    val millis: Long
        get() =
            if (isNow) System.currentTimeMillis()
            else millisArg

    override fun equals(other: Any?): Boolean =
        other is Time && millis == other.millis
}

val now = Time(isNow = true)
now == now // Sometimes true, sometimes false
```

```
List(100000) { now }.all { it == now }  
// Most likely false
```

Notice that here the result is inconsistent, so it also violates the last principle.

When an object is not equal to itself, it might not be found in most collections even if it is there when we check using the `contains` method. Such an object will not work correctly in most unit test assertions either.

```
val now1 = Time(isNow = true)  
val now2 = Time(isNow = true)  
assertEquals(now1, now2)  
// Sometimes passes, sometimes not
```

**When a result is not consistent, we cannot trust it.** We can never be sure if a result is correct or is just a result of inconsistency.

How should we improve our `Time` class? A simple solution is checking separately if the object represents the current time; if it doesn't, we should check if it has the same timestamp. Although this is a typical example of a tagged class, as described in Item 40: Prefer class hierarchies instead of tagged classes, it would be even better to use class hierarchy instead:

```
sealed class Time  
data class TimePoint(val millis: Long) : Time()  
object Now : Time()
```

**Object equality should be symmetric**, meaning that the result of `x == y` and `y == x` should always be the same. This can easily be violated when we accept objects of a different type in our equality. For instance, let's say that we implemented a class to represent complex numbers and made its equality accept `Double`:

```
class Complex(  
    val real: Double,  
    val imaginary: Double  
) {  
    // DO NOT DO THIS, violates symmetry  
    override fun equals(other: Any?): Boolean {  
        if (other is Double) {  
            return imaginary == 0.0 && real == other  
        }  
        return other is Complex &&  
            real == other.real &&
```

```
    imaginary == other.imaginary  
}  
}
```

The problem is that `Double` does not accept equality with `Complex`. Therefore, the result depends on the order of the elements:

```
Complex(1.0, 0.0).equals(1.0) // true  
1.0.equals(Complex(1.0, 0.0)) // false
```

Lack of symmetry means, for instance, unexpected results on `contains` collections or on unit tests' assertions.

```
val list = listOf<Any>(Complex(1.0, 0.0))  
list.contains(1.0) // Currently on the JVM this is false,  
// but it depends on the collection's implementation  
// and should not be trusted to stay the same
```

**When equality is not symmetric, and it is used by another object, we cannot trust the result because it depends on whether this object compares x to y or y to x.** This fact is not documented, and it is not a part of the contract as object creators assume that both should work in the same way (they assume symmetry). Also, creators might do some refactoring at any time, thus accidentally changing the order of these values. If your object is not symmetric, it might lead to unexpected and really hard-to-debug errors in your implementation. This is why when we implement `equals`, we should always consider symmetry.

The general solution is that we should not accept equality between different classes. I've never seen a case in which it would be reasonable. Notice that similar classes are not equal to each other in Kotlin. `1` is not equal to `1.0`, and `1.0` is not equal to `1.0F`. These are different types, and they are not even comparable. In Kotlin we cannot use the `==` operator between two different types that do not have a common superclass other than `Any`:

```
Complex(1.0, 0.0) == 1.0 // ERROR
```

**Object equality should be transitive**, meaning that for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`. The biggest problem with transitivity is when we implement different kinds of equality that check a different subtype of properties. For instance, let's say that we have `Date` and `DateTime` defined this way:

```
open class Date(
    val year: Int,
    val month: Int,
    val day: Int
) {
    // DO NOT DO THIS, symmetric but not transitive
    override fun equals(o: Any?): Boolean = when (o) {
        is DateTime -> this == o.date
        is Date -> o.day == day &&
            o.month == month &&
            o.year == year
        else -> false
    }
}

// ...
}

class DateTime(
    val date: Date,
    val hour: Int,
    val minute: Int,
    val second: Int
) : Date(date.year, date.month, date.day) {
    // DO NOT DO THIS, symmetric but not transitive
    override fun equals(o: Any?): Boolean = when (o) {
        is DateTime -> o.date == date &&
            o.hour == hour &&
            o.minute == minute &&
            o.second == second
        is Date -> date == o
        else -> false
    }
}

// ...
}
```

The problem with the above implementation is that when we compare two `DateTime` instances, we check more properties than when we compare `DateTime` and `Date`. Therefore, two `DateTime` instances with the same day but a different time will not be equal to each other, but they'll both be equal to the same `Date`. As a result, their relation is not transitive:

```
val o1 = DateTime(Date(1992, 10, 20), 12, 30, 0)
val o2 = Date(1992, 10, 20)
val o3 = DateTime(Date(1992, 10, 20), 14, 45, 30)

o1 == o2 // true
o2 == o3 // true
o1 == o3 // false, so equality is not transitive

setOf(o2, o1, o3).size // 1 or 2?
// Depends on the collection's implementation
```

Notice that the restriction to compare only objects of the same type doesn't help here because we've used inheritance. Such inheritance violates the Liskov substitution principle and should not be used. In this case, use composition instead of inheritance (Item 36: Prefer composition over inheritance). When you do use composition instead of inheritance, do not compare two objects of different types. These classes are perfect examples of objects that hold data, so representing them this way is a good choice:

```
data class Date(
    val year: Int,
    val month: Int,
    val day: Int
)

data class DateTime(
    val date: Date,
    val hour: Int,
    val minute: Int,
    val second: Int
)

val o1 = DateTime(Date(1992, 10, 20), 12, 30, 0)
val o2 = Date(1992, 10, 20)
val o3 = DateTime(Date(1992, 10, 20), 14, 45, 30)

o1.equals(o2) // false
o2.equals(o3) // false
o1 == o3 // false

o1.date.equals(o2) // true
o2.equals(o3.date) // true
```

```
o1.date == o3.date // true
```

**Equality should be consistent**, meaning that the method invoked on two objects should always return the same result unless one of these objects was modified. For immutable objects, the result should always be the same. In other words, we expect `equals` to be a pure function (it should not modify the state of an object) whose result always depends only on the input and the state of its receiver. We've seen a `Time` class which violated this principle. This rule was also famously violated in `java.net.URL.equals()`, what will be explained soon.

**An object other than null should never be equal to null:** for any non-null value `x`, `x.equals(null)` must return `false`. This is important because `null` should be unique, and no object should be equal to it.

## The problem with `equals` in `java.net.URL`

One example of a really poorly designed `equals` is the one from `java.net.URL`. The equality of two `java.net.URL` objects depends on a network operation as two hosts are considered equivalent if both hostnames can be resolved to the same IP addresses. Take a look at the following example:

```
import java.net.URL

fun main() {
    val enWiki = URL("https://en.wikipedia.org/")
    val wiki = URL("https://wikipedia.org/")
    println(enWiki == wiki)
}
```

Should it return `true` or `false`? According to the contract, it should be `true`, but the result is inconsistent. In normal conditions, it should print `true` because the IP address for both URLs is resolved as the same; however, if you have the internet disconnected, it will print `false`. You can check this yourself. This is a big mistake! Equality should not be network-dependent.

Here are the most important problems with this solution:

- **This behavior is inconsistent.** For instance, two URLs could be equal when the internet connection is available but unequal when it is not. Also, IP addresses resolved by a URL can change over time, so the result might be inconsistent.
- **The network may be slow, and we expect `equals` and `hashCode` to be fast.** A typical problem is when we check if a URL is present in a list. Such an

operation would require a network call for each element in the list. Also, on some platforms, like Android, network operations are prohibited on the main thread. As a result, even adding a URL to a set needs to be started on a separate thread.

- **The defined behavior is known to be inconsistent with virtual hosting in HTTP.** Equal IP addresses do not imply equal content. Virtual hosting allows unrelated sites to share an IP address. This method could report two otherwise unrelated URLs to be equal because they're hosted on the same server.

In Android, this problem was fixed in Android 4.0 (Ice Cream Sandwich). Since that release, URLs are only equal if their hostnames are equal. When we use Kotlin/JVM on other platforms, it is recommended to use `java.net.URI` instead of `java.net.URL`.

## Implementing equals

I recommend against implementing `equals` yourself unless you have a good reason. Instead, use the default implementation or data class equality. If you do need custom equality, always consider whether your implementation is reflexive, symmetric, transitive, and consistent. The typical implementation of `equals` looks like this:

```
override fun equals(other: Any?): Boolean {
    if (this === other) return true
    if (other !is MyClass) return false
    return field1 == other.field1 &&
           field2 == other.field2 &&
           field3 == other.field3
}
// or
override fun equals(other: Any?) =
    this === other ||
    (other is MyClass &&
     field1 == other.field1 &&
     field2 == other.field2 &&
     field3 == other.field3)
```

Make classes with custom `equals` final, or beware that subclasses should not change how equality behaves. It is hard to make custom equality while inheri-

tance at the same time. Some even say it is impossible<sup>64</sup>. This is one of the reasons why data classes are final.

## Summary

- The == operator translates to equals and checks structural equality. The === operator checks referential equality, i.e., if two values are exactly the same object.
- Equality is reflexive, symmetric, transitive, and consistent. If you implement equals yourself, make sure it follows these rules.
- To fulfill the contract of equals, only classes of the same type should be considered equal. equals should be fast and should not require an internet connection. A famous example of a poor implementation from Java stdlib is java.net.URL equality.

---

<sup>64</sup>As Effective Java by Joshua Bloch (third edition) claims in Item 10: Obey the general contract when overriding equals: “There is no way to extend an instantiable class and add a value component while preserving the equals contract, unless you’re willing to forgo the benefits of object-oriented abstraction”. I have a feeling it is true, but I cannot prove it, so I avoid definitive statements.

## Item 43: Respect the contract of hashCode

Another method from `Any` that we can override is `hashCode`. First, let's explain why we need it. The `hashCode` function is used in a popular data structure called a *hash table*, which is used under the hood in a variety of different collections or algorithms.

### Hash table

Let's start with the problem the hash table was invented to solve. Let's say that we need a collection that quickly both adds and finds elements. An example of this type of collection is a set or a map, neither of which allows duplicates. So, whenever we add an element, we first need to look for an equal element.

A collection based on an array or on linked elements is not fast enough to check if it contains an element because to do this we need to compare this element with all elements in this list, one after another. Imagine that you have an array with millions of pieces of text, and now you need to check if it contains a certain one. It would be very time-consuming to compare your text against all these millions.

A popular solution to this problem is a hash table. All you need is a function that will assign a number to each element. Such a function is called a hash function, and it must always return the same value for equal elements. Additionally, it is good if our hash function:

- Is fast.
- Returns different values for unequal elements (or at least has enough variation to limit collisions to a minimum).

Such a function categorizes elements into different buckets by assigning a number to each one. What is more, based on our requirement for the hash function, all elements equal to each other will always be placed in the same bucket. These buckets are kept in a structure called a hash table, which is an array whose size is equal to the number of buckets. Every time we add an element, we use our hash function to calculate where it should be placed, and we add it there. Notice that this process is very fast because calculating the hash should be fast, and then we just use the result of the hash function as an index in the array to find our bucket. When we search for an element, we find its bucket in the same way, and then we only need to check if it is equal to any element in this bucket. We don't need to check any other bucket because the hash function must return the same value for equal elements. In this way, at a low cost, it divides the number of operations needed to find an element by the number of buckets. For instance, if we have 1,000,000 elements and 100,000 buckets, searching for duplicates only requires

the comparison of about 10 elements on average, and the performance cost of this improvement is tiny.

To see a more concrete example, let's say that we have the following strings and a hash function that splits elements into 4 buckets:

| Text   | Hash code |
|--|-----------|
| “How much wood would a woodchuck chuck”        | 3         |
| “Peter Piper picked a peck of pickled peppers” | 2         |
| “Betty bought a bit of butter”                 | 1         |
| “She sells seashells by the seashore”          | 2         |

Based on those numbers, we will have built the following hash table:

| Index | The object to which the hash table points   |
|-------|---|
| 0     | []  |
| 1     | [“Betty bought a bit of butter”]  |
| 2     | [“Peter Piper picked a peck of pickled peppers”, “She sells seashells by the seashore”] |
| 3     | [“How much wood would a woodchuck chuck”]   |

Now, when we are checking if a new text is in this hash table, we are calculating its hash code. If it is equal to 0, then we know that it is not in this list. If it is either 1 or 3, we need to compare it with a single text. If it is 2, we need to compare it with two pieces of text.

This concept is very popular in technology. It is used in databases, in many internet protocols, and in standard library collections in many languages. In Kotlin/JVM, both the default set (`LinkedHashSet`) and the default map (`LinkedHashMap`) use it (they both create more buckets than there are elements, so the complexity of checking if elements is a set or a key in a map is  $O(1)$  if hash code is implemented well). To produce a hash code, we use the `hashCode` function<sup>65</sup>.

## The problem with mutability

Notice that a hash is calculated for an element only when this element is added. An element is not moved when it mutates because the collection does not know it has changed. So, after an element is changed, it might be in the wrong bucket.

---

<sup>65</sup>Often, after some transformations, `hashCode` returns `Int`, which is a 32-bit signed integer (i.e., 4, 294, 967, 296 buckets), which is too much for a set that might contain only one element. To solve this problem, there is a transformation that makes this number much smaller. When it is needed, the algorithm transforms the previous hash table into a new one with new buckets.

This is why both `LinkedHashSet` and `LinkedHashMap` will not behave properly when an object mutates after it has been added:

```
data class FullName(  
    var name: String,  
    var surname: String  
)  
  
val person = FullName("Maja", "Markiewicz")  
val s = mutableSetOf<FullName>()  
s.add(person)  
person.surname = "Moskała"  
print(person) // FullName(name=Maja, surname=Moskała)  
print(s.contains(person)) // false  
print(person in s) // false  
print(s.first() == person) // true
```

This problem was already noted in Item 1: *Limit mutability*: Mutable objects should not be used in data structures based on hashes or in any other data structure that organizes elements based on their mutable properties. We should not use mutable elements for sets or as keys for maps, or at least we should not mutate elements that are in such collections. This is also a great reason to use immutable objects in general.

## The contract of hashCode

Now that we know what we need `hashCode` for, it should be clear how we expect it to behave. The formal contract is as follows (Kotlin 1.9.0):

- Whenever it is invoked on the same object more than once, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified.
- If two objects are equal according to the `equals()` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

Notice that the first requirement is that we need `hashCode` to be **consistent**. The second is the one that developers often forget about and which therefore needs to be highlighted is that **hashCode always needs to be consistent with equals, and equal elements must have the same hash code**. If they don't, elements will be lost in collections that use a hash table under the hood:

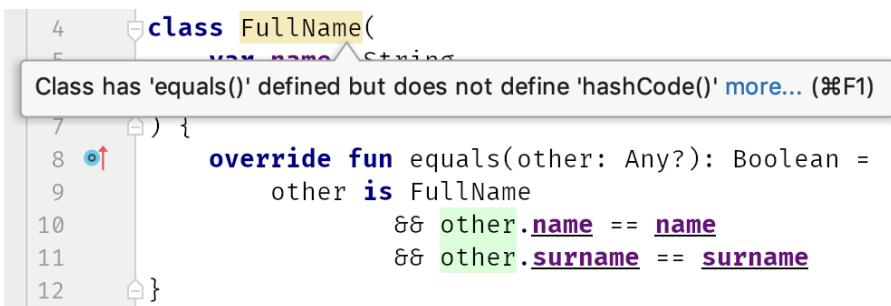
```

class FullName(
    var name: String,
    var surname: String
) {
    override fun equals(other: Any?): Boolean =
        other is FullName
        && other.name == name
        && other.surname == surname
}

val s = mutableSetOf<FullName>()
s.add(FullName("Marcin", "Moskała"))
val p = FullName("Marcin", "Moskała")
print(p in s) // false
print(p == s.first()) // true

```

This is why Kotlin suggests overriding hashCode when you have a custom equals implementation.



There is also a requirement that is not strictly required but is very important if we want hashCode to be useful: hashCode should spread elements as widely and variously as possible as different elements should have the highest possible probability of having different hash values.

Think about what happens when many elements are placed in the same bucket: there is no advantage in using a hash table! An extreme example would be making hashCode always return the same number. Such a function would always place all elements into the same bucket. This fulfills the formal contract, but it is completely useless. There is no advantage to using a hash table when hashCode always returns the same value. Just take a look at the examples below, where you can see a properly implemented hashCode and one that always returns 0. For each equals we add, a counter counts how many times it was used. You can see this when we operate on sets with values of both types: the second class, named Terrible, requires many more comparisons:

```
class Proper(val name: String) {  
  
    override fun equals(other: Any?): Boolean {  
        equalsCounter++  
        return other is Proper && name == other.name  
    }  
  
    override fun hashCode(): Int {  
        return name.hashCode()  
    }  
  
    companion object {  
        var equalsCounter = 0  
    }  
}  
  
class Terrible(val name: String) {  
    override fun equals(other: Any?): Boolean {  
        equalsCounter++  
        return other is Terrible && name == other.name  
    }  
  
    // Terrible choice, DO NOT DO THAT  
    override fun hashCode() = 0  
  
    companion object {  
        var equalsCounter = 0  
    }  
}  
  
val properSet = List(10000) { Proper("$it") }.toSet()  
println(Proper.equalsCounter) // 0  
val terribleSet = List(10000) { Terrible("$it") }.toSet()  
println(Terrible.equalsCounter) // 50116683  
  
Proper.equalsCounter = 0  
println(Proper("9999") in properSet) // true  
println(Proper.equalsCounter) // 1  
  
Proper.equalsCounter = 0  
println(Proper("A") in properSet) // false  
println(Proper.equalsCounter) // 0
```

```
Terrible.equalsCounter = 0
println(Terrible("9999") in terribleSet) // true
println(Terrible.equalsCounter) // 4324

Terrible.equalsCounter = 0
println(Terrible("A") in terribleSet) // false
println(Terrible.equalsCounter) // 10001
```

## Implementing hashCode

We define hashCode in Kotlin only when we define a custom equals. When we use the data modifier, it generates both equals and a consistent hashCode. When you do not have a custom equals method, do not define a custom hashCode unless you are sure you know what you are doing and you have a good reason for it. When you have a custom equals, implement a hashCode that always returns the same value for equal elements.

If you implemented a typical equals that checks the equality of significant properties, then a typical hashCode should be calculated using the hash codes of these properties. How can we make a single hash code out of this many hash codes? A typical way is to accumulate them all in a result, and every time we add the next one we multiply the result by the number 31. It doesn't need to be exactly 31, but this number's characteristics make it good for this purpose. It is used this way so often that now we can treat it as a convention. Hash codes generated by the data modifier are consistent with this convention. Here is an example implementation of a typical hashCode, together with its equals:

```
class DateTime(
    private var millis: Long = 0L,
    private var timeZone: TimeZone? = null
) {
    private var asStringCache = ""
    private var changed = false

    override fun equals(other: Any?): Boolean =
        other is DateTime &&
        other.millis == millis &&
        other.timeZone == timeZone

    override fun hashCode(): Int {
        var result = millis.hashCode()
        result = result * 31 + timeZone.hashCode()
    }
}
```

```
        return result
    }
}
```

One helpful function in Kotlin/JVM is `Objects.hash`, which calculates the hash of multiple objects using the same algorithm as presented above:

```
override fun hashCode(): Int =
    Objects.hash(timeZone, millis)
```

There is no such function in the Kotlin stdlib, but you can implement it yourself if you need it on other platforms:

```
override fun hashCode(): Int =
    hashCodeFrom(timeZone, millis)

inline fun hashCodeOf(vararg values: Any?) =
    values.fold(0) { acc, value ->
        (acc * 31) + value.hashCode()
    }
```

The reason why such a function is not in the stdlib is that we rarely need to implement `hashCode` ourselves. For instance, in the `DateTime` class presented above, instead of implementing `equals` and `hashCode` ourselves, we can just use the `data` modifier:

```
data class DateTime2(
    private var millis: Long = 0L,
    private var timeZone: TimeZone? = null
) {
    private var asStringCache = ""
    private var changed = false
}
```

When you do implement `hashCode`, remember that the most important rule is that it always needs to be consistent with `equals`, and it should always return the same value for elements that are equal.

## Summary

- `hashCode` is used to calculate a hash code for an object. It is used by hash tables to place elements in buckets, so we can find them faster.

- `hashCode` should be consistent with `equals`. This means that if two objects are equal, they should have the same hash code. So whenever you override `equals`, you should also override `hashCode`.
- `hashCode` should be fast and should spread elements as widely as possible.
- We rarely need to implement `hashCode` ourselves. When we do, we can use `Objects.hash` function on Kotlin/JVM. On other platforms, we can implement a similar function ourselves.

## Item 44: Respect the contract of `compareTo`

The `compareTo` method is not in the `Any` class. It is an operator in Kotlin that translates into the mathematical comparison signs:

```
obj1 > obj2 // translates to obj1.compareTo(obj2) > 0  
obj1 < obj2 // translates to obj1.compareTo(obj2) < 0  
obj1 >= obj2 // translates to obj1.compareTo(obj2) >= 0  
obj1 <= obj2 // translates to obj1.compareTo(obj2) <= 0
```

It is also located in the `Comparable<T>` interface. When an object implements this interface, or when it has an operator method named `compareTo` with one parameter, this means that this object has a natural order. Such an order needs to be:

- **Antisymmetric**, meaning if  $a \geq b$  and  $b \geq a$ , then  $a == b$ . Therefore, there is a relation between comparison and equality, and they need to be consistent with each other.
- **Transitive**, meaning if  $a \geq b$  and  $b \geq c$ , then  $a \geq c$ . Similarly, when  $a > b$  and  $b > c$ , then  $a > c$ . This property is important because sorting elements without it might take literally forever in some sorting algorithms.
- **Connex**, meaning there must be a relationship between every two elements: either  $a \geq b$  or  $b \geq a$ . In Kotlin, this relationship is guaranteed by the type system, because `compareTo` returns `Int`, and every `Int` is either positive, negative, or zero. This property is important because if there is no relationship between two elements, we cannot use classic sorting algorithms like quicksort or insertion sort. Instead, we need to use one of the special algorithms for partial orders, like topological sorting.

### Do we need a `compareTo`?

In Kotlin we rarely implement `compareTo` ourselves. We get more freedom by specifying the order on a case-by-case basis than by assuming one global natural order. For instance, we can sort a collection using `sortedBy` and provide a key that is comparable. So, in the example below, we sort users by their surnames:

```
class User(val name: String, val surname: String)  
  
val names = listOf<User>(/*...*/)  
  
val sorted = names.sortedBy { it.surname }
```

What if we need a more complex comparison than just by a key? For that, we can use the `sortedWith` function, which sorts elements using a comparator. This comparator can be produced using the `compareBy` function. So, in the following example, we sort users by comparing them by their `surname`; if they match, we compare them by their `name`:

```
val sorted = names
    .sortedWith(compareBy({ it.surname }, { it.name }))
```

Surely, we might make `User` implement `Comparable<User>`, but what order should it choose? Is any order truly natural for this type? When this is not absolutely clear, it is better to not make such objects comparable, and the objects' order should be specified for each sorting.

The natural order of `String` is alphanumeric, therefore it implements `Comparable<String>`. This is very useful because we often do need to sort text alphabetically; however, it also has its downsides: for instance, we can compare two strings using a comparison sign, which seems highly unintuitive. Most people who see a comparison sign between two strings will be rather confused.

```
// DON'T DO THIS!
print("Kotlin" > "Java") // true
```

Surely there are objects with a clear natural order? Units of measure, date, and time are all perfect examples. However, if you are not sure about whether your object has a natural order, it is better to use comparators instead. If you use a few of them often, you can place them in the companion object of your class:

```
class User(val name: String, val surname: String) {
    // ...

    companion object {
        val DISPLAY_ORDER =
            compareBy(User::surname, User::name)
    }
}

val sorted = names.sortedWith(User.DISPLAY_ORDER)
```

## Implementing `compareTo`

When we do need to implement `compareTo` ourselves, we have top-level functions that can help us. If all you need is to compare two values, you can use the `compareValues` function:

```
class User(  
    val name: String,  
    val surname: String  
) : Comparable<User> {  
    override fun compareTo(other: User): Int =  
        compareValues(surname, other.surname)  
}
```

If you need to use more values, or if you need to compare them using selectors, use `compareValuesBy`:

```
class User(  
    val name: String,  
    val surname: String  
) : Comparable<User> {  
    override fun compareTo(other: User): Int =  
        compareValuesBy(this, other,  
            { it.surname },  
            { it.name })  
}
```

This function helps us create most of the comparators we might need. If you need to implement one with a special logic, remember that it should return:

- 0 if the receiver and other are equal
- a positive number if the receiver is greater than other
- a negative number if the receiver is smaller than other

If you do this, don't forget to verify that your comparison is antisymmetric, transitive, and connex.

## Summary

- Classes with a clear natural order should implement `Comparable<T>`, that is, they should have a `compareTo` method. If you are not sure whether your class has a natural order, it is better to not implement `Comparable<T>`.
- The `compareTo` method is used to decide which of two objects is considered greater. We can use it with comparison operators (`>`, `<`, `>=`, `<=`). `compareTo` should be antisymmetric, transitive, and connex.
- When you implement custom `compareTo`, we often use helper methods from Kotlin stdlib. If you need to compare two comparable values, use the `compareValues` function. If you need to compare values by a few of their comparable properties, use `compareValuesBy`.

## Item 45: Consider extracting non-essential parts of your API into extensions

When we define final methods in a class, we need to decide whether we want to define them as members or as extension functions.

```
// Defining methods as members
class Workshop(/*...*/) {
    //...

    fun makeEvent(date: DateTime): Event = //...

    val permalink
        get() = "/workshop/$name"
}

// Defining methods as extensions
class Workshop(/*...*/) {
    //...

    fun Workshop.makeEvent(date: DateTime): Event = //...

    val Workshop_permalink
        get() = "/workshop/$name"
```

Both approaches are similar in many ways. Their use and even referencing them via reflection is very similar:

```
fun useWorkshop(workshop: Workshop) {
    val event = workshop.makeEvent(date)
    val permalink = workshop_permalink

    val makeEventRef = Workshop::makeEvent
    val permalinkPropRef = Workshop::permalink
}
```

However, there are some significant differences between those two options. They both have their pros and cons, thus one way does not dominate over the other. Therefore, I suggest **considering** extracting non-essential parts of your API into

extensions, not necessarily doing it. The point is to make smart decisions, and to do this we need to understand the differences between those two options.

The biggest difference between members and extensions in terms of use is that **extensions need to be imported separately**. For this reason, they can be located in a different package. This fact is used when we cannot add a member ourselves. It is also used in projects designed to separate data and behavior. Properties with fields need to be located in a class, but methods can be located separately as long as they only access the class's public API.

Thanks to the fact that extensions need to be imported, **we can have many extensions with the same name on the same type**. This is good because different libraries can provide extra methods, therefore we won't have a conflict. On the other hand, it would be dangerous to have two extensions with the same name but with different behavior. For such cases, we can cut the Gordian knot by making a member function. The compiler always chooses member functions over extensions<sup>66</sup>.

Another significant difference is that **extensions are not virtual**, meaning they cannot be redefined in derived classes. The extension function to call is selected statically during compilation. This is different behavior than member elements that are virtual in Kotlin. Therefore, we should not use extensions for elements that are designed for inheritance.

```
open class C
class D : C()

fun C.foo() = "c"
fun D.foo() = "d"

fun main() {
    val d = D()
    print(d.foo()) // d
    val c: C = d
    print(c.foo()) // c

    print(D().foo()) // d
    print((D() as C).foo()) // c
}
```

This behavior is the result of the fact that extension functions under the hood are compiled into normal functions, where the extension's receiver is placed as the first argument:

---

<sup>66</sup>The only exception is when an extension in the Kotlin stdlib has `kotlin.internal.HidesMembers` internal annotation.

```
fun foo(`this$receiver`: C) = "c"
fun foo(`this$receiver`: D) = "d"

fun main() {
    val d = D()
    print(foo(d)) // d
    val c: C = d
    print(foo(c)) // c

    print(foo(D())) // d
    print(foo(D() as C)) // c
}
```

Another consequence of this fact is that **we define extensions on types, not on classes**. This gives us more freedom. For instance, we can define an extension on a nullable or generic type:

```
inline fun CharSequence?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }

    return this == null || this.isBlank()
}

public fun Iterable<Int>.sum(): Int {
    var sum: Int = 0
    for (element in this) {
        sum += element
    }
    return sum
}
```

The last important difference is that **extensions are not listed as members in the class reference**. This is why they are not considered by annotation processors, and we cannot extract elements that should be processed into extensions when we process a class using annotation processing. On the other hand, if we extract non-essential elements into extensions, we don't need to worry about them being seen by these processors. We don't need to hide them because they are not in the class anyway.

Let me show you two examples where defining extensions makes more sense than defining members. The first one is the `Iterable` interface and its many extensions

like `map` or `filter`. These methods could be defined inside `Iterable` as members, but that would be a bad idea. They do not define the interface's essential behavior but rather some utils that can be used on iterable objects. Thanks to the fact that these methods are extensions, the `Iterable` interface is clean and easy to understand.

```
// Kotlin stdlib
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}

public inline fun <T, R> Iterable<T>.map(
    transform: (T) -> R
): List<R> {
    // ...
}

public inline fun <T> Iterable<T>.filter(
    predicate: (T) -> Boolean
): List<T> {
    // ...
}
```

Another example is a conversion function between two classes that represent a similar abstraction but on different layers of our application, such as a domain class `Product` and a data layer class `ProductJson`. The conversion functions `toProduct` and `toProductJson` could be members, but we generally prefer to define them as extensions. This way, we can keep our domain classes clean and free of data layer dependencies. This also lets us keep both these conversion functions next to each other, which makes them easier to maintain.

```
fun ProductJson.toProduct() = Product(
    id = this.id,
    title = this.title,
    imgSrc = this.img,
    description = this.desc,
    price = BigDecimal(this.price),
    type = enumValueOf<ProductType>(this.type)
)

fun Product.toProductJson() = ProductJson(
    id = this.id,
```

```
    title = this.title,  
    img = this.imgSrc,  
    desc = this.description,  
    price = this.price.toString(),  
    type = this.type.name  
)
```

## Summary

The most important differences between members and extensions are:

- Extensions need to be imported
- Extensions are not virtual
- Members have higher priority
- Extensions are on a type, not on a class
- Extensions are not listed in the class reference

To summarize this, extensions give us more freedom and flexibility. However, they do not support inheritance or annotation processing, and it might be confusing that they are not present in the class they are called on. The essential parts of our API should generally be members, but there are good reasons to extract non-essential parts of your API as extensions.

## Item 46: Avoid member extensions

When we define an extension function to some class, it is not added to this class as a member. An extension function is just a different kind of function that we call on the first argument that is there, which is called a receiver. Under the hood, extension functions are compiled to normal functions, and the receiver is placed as the first parameter. For instance, the following function:

```
fun String.isPhoneNumber(): Boolean =  
    length == 7 && all { it.isDigit() }
```

Under the hood is compiled to a function similar to this one:

```
fun isPhoneNumber(`$this`: String): Boolean =  
    `$this`.length == 7 && `$this`.all { it.isDigit() }
```

One of the consequences of how extension functions are implemented is that we can have member extensions or even define extensions in interfaces:

```
interface PhoneBook {  
    fun String.isPhoneNumber(): Boolean  
}  
  
class Fizz : PhoneBook {  
    override fun String.isPhoneNumber(): Boolean =  
        this.length == 7 && this.all { it.isDigit() }  
}
```

Even though it is possible, there are good reasons to avoid defining member extensions (except for DSLs). **Especially, do not define extensions as members just to restrict visibility.**

```
class PhoneBookIncorrect {  
  
    fun verify(number: String): Boolean {  
        require(number.isPhoneNumber())  
        // ...  
    }  
  
    // Bad practice, do not do this  
    fun String.isPhoneNumber(): Boolean =  
        this.length == 7 && this.all { it.isDigit() }  
}
```

One big reason is that member extension functions do not really restrict visibility. They only make it more complicated to use the extension function, since the user would need to provide both the extension and the dispatch receivers:

```
PhoneBookIncorrect().apply {  
    "1234567890".isPhoneNumber()  
}
```

You should restrict an extension's visibility by using a visibility modifier, not by making it a member.

```
class PhoneBook {  
  
    fun verify(number: String): Boolean {  
        require(number.isPhoneNumber())  
        // ...  
    }  
  
    // ...  
}  
  
// This is how we limit extension functions visibility  
private fun String.isPhoneNumber(): Boolean =  
    this.length == 7 && this.all { it.isDigit() }
```

If you need a function to be a member, for instance when it needs to use a class state, and you want to call it like an extension, consider using `let`.

```
class PhoneBook(  
    private val phoneNumberVerifier: PhoneNumberVerifier  
) {  
  
    fun verify(number: String): Boolean {  
        require(number.let(::isPhoneNumber))  
        // ...  
    }  
  
    // ...  
  
    private fun isPhoneNumber(number: String): Boolean =  
        phoneNumberVerifier.verify(number)  
}
```

## Why to avoid extension functions

There are a few good reasons why we prefer to avoid member extensions:

- Reference is not supported:

```
val ref = String::isPhoneNumber
val str = "1234567890"
val boundedRef = str::isPhoneNumber

val refX = PhoneBookIncorrect::isPhoneNumber // ERROR
val book = PhoneBookIncorrect()
val boundedRefX = book::isPhoneNumber // ERROR
```

- Implicit access to both receivers might be confusing:

```
class A {
    val a = 10
}
class B {
    val a = 20
    val b = 30

    fun A.test() = a + b // Is it 40 or 50?
}
```

- When we expect an extension to modify or reference a receiver, it is not clear if we modify the extension receiver or the dispatch receiver (the class in which the extension is defined):

```
class A {
    //...
}
class B {
    //...

    fun A.update() ... // Does it update A or B?
}
```

- For less experienced developers, it might be counterintuitive or scary to see member extensions.

## Avoid, not prohibit

This rule does not apply everywhere. The most obvious situation in which member extensions need to be used is when we define DSL builders (as presented in *Item 34: Consider defining a DSL for complex object creation*). Member extensions are also useful when we need to define a function that is called on some object representing a scope. One example might be a member function that produces a Channel using the produce function. Another might be an integration test function that calls an endpoint on TestApplicationEngine and is defined in an interface (as I explained in my article [Traits for testing in Kotlin<sup>67</sup>](#), this is a popular pattern used in backend integration tests). In both cases, defining this function as a scope is not our whim but serves a concrete purpose.

```
class OrderUseCase(
    // ...
) {
    // ...

    private fun CoroutineScope.produceOrders() =
        produce<Order> {
            var page = 0
            do {
                val orders = api
                    .requestOrders(page = page++)
                    .orEmpty()
                for (order in orders) send(order)
            } while (orders.isNotEmpty())
        }
}

interface UserApiTrait {

    fun TestApplicationEngine.requestRegisterUser(
        token: String,
        request: RegisterUserRequest
    ): UserJson? = ...

    fun TestApplicationEngine.request GetUserSelf(
        token: String
    ): UserJson? = ...
}
```

---

<sup>67</sup><https://kt.academy/article/traits-testing>

```
// ...
}
```

We prefer to avoid member extensions, but we use them if they are the best option we have.

## Summary

To summarize, if there is a good reason to use a member extension, it is fine. Just be aware of the downsides and generally try to avoid it. To restrict visibility, use visibility modifiers. Just placing an extension in a class does not limit its use from outside.

## Part 3: Efficiency



# Chapter 7: Make it cheap

Code efficiency today is often treated leniently. To a certain degree, this is reasonable as memory is cheap and developers are expensive. However, efficiency should not be ignored. Maybe your application is running on millions of devices, so it consumes a lot of energy, and some battery use optimization might save enough energy to power a small city. Maybe your company is paying a lot of money for servers and their maintenance, and some optimization might make this significantly cheaper. Or maybe your application works well for a small number of requests, but it does not scale well and is therefore unusable when there are unexpected peaks. Customers remember such situations.

Efficiency is important in the long term, but optimization is not easy. Premature optimization often does more harm than good. Instead, there are some rules that can help you make more efficient programs nearly painlessly. These are the cheap wins: they cost nearly nothing, but they can still help us improve performance significantly. When they are not sufficient, we should use a profiler and optimize performance-critical parts. This is more difficult to achieve because it requires a deeper understanding of what is expensive and how some optimizations can be done.

This and the next chapter are about performance:

- *Chapter 7: Make it cheap* - more general suggestions for performance.
- *Chapter 8: Efficient collection processing* - concentrates on collection processing.

These chapters focus on general rules that can be applied to optimize everyday development cheaply. They also give some Kotlin-specific suggestions on how performance might be optimized in the critical parts of your program. They should also extend your understanding of thinking about performance in general.

Please remember that when there is a tradeoff between readability and performance, you need to ask yourself what is more important in the components you develop. I have included some suggestions, but there is no universal answer.

## Item 47: Avoid unnecessary object creation

Object creation always costs something and can sometimes be expensive. This is why avoiding unnecessary object creation can be an important optimization. It can be done on many levels. For instance, in JVM it is guaranteed that a string object will be reused by other code running in the same virtual machine that happens to contain the same string literal<sup>68</sup>:

```
val str1 = "Lorem ipsum dolor sit amet"  
val str2 = "Lorem ipsum dolor sit amet"  
print(str1 == str2) // true  
print(str1 === str2) // true
```

Boxed primitives (Integer, Long) are also reused in JVM when they are small (by default, the Integer Cache holds numbers in the range from -128 to 127).

```
val i1: Int? = 1  
val i2: Int? = 1  
print(i1 == i2) // true  
print(i1 === i2) // true, because i2 was taken from cache
```

Reference equality (==) shows that this is the same object. However, if we use a number that is either smaller than -128 or bigger than 127, different objects will be created:

```
val j1: Int? = 1234  
val j2: Int? = 1234  
print(j1 == j2) // true  
print(j1 === j2) // false
```

Notice that a nullable type is used to force Integer instead of int under the hood. When we use Int, it is generally compiled to the primitive int, but if we make it nullable or when we use it as a type argument, Integer is used instead. This is because a primitive cannot be null and cannot be used as a type argument.

Knowing that such mechanisms are available in Kotlin, you might wonder how significant they are. Is object creation expensive?

---

<sup>68</sup>Java Language Specification, Java SE 8 edition, 3.10.5

## Is object creation expensive?

Wrapping something into an object has 3 costs:

- **Objects take additional space.** In a modern 64-bit JDK, an object has a 12-byte header that is padded to a multiple of 8 bytes, so the minimum object size is 16 bytes. For 32-bit JVMs, the overhead is 8 bytes. Additionally, object references also take space. Typically, references are 4 bytes on 32-bit or 64-bit platforms up to -Xmx32G, and they are 8 bytes for memory allocation pool set above 32Gb (-Xmx32G). These are relatively small numbers, but they can add up to a significant cost. When we think about small elements like integers, they make a difference. Int as a primitive fit in 4 bytes, but when it is a wrapped type on the 64-bit JDK we mainly use today, it requires 16 bytes (it fits in the 4 bytes after the header), and its reference requires 4 or 8 bytes. In the end, it takes 5 or 6 times more space<sup>69</sup>. This is why an array of primitive integers (IntArray) takes 5 times less space than an array of wrapped integers (Array<Int>), as explained in the Item 58: Consider Arrays with primitives for performance-critical processing.
- **Access requires an additional function call when elements are encapsulated.** Again, this is a small cost as function use is very fast, but it can add up when we need to operate on a huge pool of objects. We will see how this cost can be eliminated in Item 51: Use the inline modifier for functions with parameters of functional types and Item 49: Consider using inline classes.
- **Objects need to be created and allocated in memory, references need to be created, etc.** These are small numbers, but they can rapidly accumulate when there are many objects. In the snippet below, you can see the cost of object creation.

```
class A

private val a = A()

// Benchmark result: 2.698 ns/op
fun accessA(blackhole: Blackhole) {
    blackhole.consume(a)
}

// Benchmark result: 3.814 ns/op
fun createA(blackhole: Blackhole) {
    blackhole.consume(A())
}
```

---

<sup>69</sup>To measure the size of concrete fields in JVM objects, use Java Object Layout.

```
// Benchmark result: 3828.540 ns/op
fun createListAccessA(blackhole: Blackhole) {
    blackhole.consume(List(1000) { a })
}

// Benchmark result: 5322.857 ns/op
fun createListCreateA(blackhole: Blackhole) {
    blackhole.consume(List(1000) { A() })
}
```

By eliminating objects, we can avoid all three of these costs. By reusing objects, we can eliminate the first and the third ones. If we know the costs of objects, we can start considering how we can minimize these costs in our applications by limiting the number of unnecessary objects. In the next few items, we will see different ways to eliminate or reduce the number of objects. In this item, I will only present one technique, that is designing classes to use primitives instead of wrapped types.

## Using primitives

In JVM, we have a special built-in type to represent basic elements like numbers or characters. These are called primitives and are used by the Kotlin/JVM compiler under the hood wherever possible. However, there are some cases where a wrapped class (an object instance containing a primitive) needs to be used instead. The two main cases are:

1. When we operate on a nullable type (primitives cannot be `null`).
2. When we use a type as a generic type argument.

So, in short:

| Kotlin type | Java type     |
|-------------|---------------|
| Int         | int           |
| Int?        | Integer       |
| List<Int>   | List<Integer> |

Now you know that you can optimize your code to have primitives under the hood instead of wrapped types. Such optimization makes sense mainly on Kotlin/JVM and on some flavors of Kotlin/Native, but it doesn't make any sense on Kotlin/JS. Access to both primitive and wrapped types is relatively fast compared to other operations. The difference manifests itself when we deal with bigger collections

(we will discuss this in Item 58: Consider Arrays with primitives for performance-critical processing) or when we operate on an object intensively. Also, remember that forced changes might lead to less-readable code. **This is why I suggest this optimization only for performance-critical parts of code and in libraries.** You can identify the performance-critical parts of your code using a profiler.

To consider a concrete example, let's imagine that you implement a financial application in which you need to represent a stock snapshot. A snapshot is a set of values that are updated twice a second. It contains the following information:

```
class Snapshot(  
    val afterHours: SessionDetails,  
    val preMarket: SessionDetails,  
    val regularHours: SessionDetails,  
)  
  
data class SessionDetails(  
    val open: Double? = null,  
    val high: Double? = null,  
    val low: Double? = null,  
    val close: Double? = null,  
    val volume: Long? = null,  
    val dollarVolume: Double? = null,  
    val trades: Int? = null,  
    val last: Double? = null,  
    val time: Int? = null,  
)
```

Since you are tracking tens of thousands of stocks, and the snapshot for each of them is updated twice a second, your application will create instances of SessionDetails many times per second, which will require a lot of effort from the garbage collector. To avoid this, you can change the SessionDetails class to use primitives instead of wrapped types by eliminating nullability.

```
data class SessionDetails(  
    val open: Double = Double.NaN,  
    val high: Double = Double.NaN,  
    val low: Double = Double.NaN,  
    val close: Double = Double.NaN,  
    val volume: Long = -1L,  
    val dollarVolume: Double = Double.NaN,  
    val trades: Int = -1,  
    val last: Double = Double.NaN,
```

```
    val time: Int = -1,  
)
```

Note that **this change harms readability and makes this class harder to use** because `null` is a better way to represent the lack of a value than a special value like `NAN` or `-1`. However, in this case we decided to make this change because we are dealing with a performance-critical part of the application. By eliminating nullability, we've made our object allocate far fewer objects and much less memory. On a typical machine, the first version of `SessionDetails` allocates 192 bytes and needs to create 10 objects; in contrast, the second version allocates only 80 bytes and needs to create only one object. This is a significant difference that might be worth the trouble when we are dealing with tens of thousands of objects.

If such interventions are not enough in your application, you can consider using a very powerful but also very dangerous pattern object pool. Its core idea is to make objects mutable and to store and reuse unused objects. This pattern is hard to implement correctly, and it is easy to introduce synchronization issues, which is why I don't recommend using it unless you're sure that you need it.

## Summary

In this chapter, we learned about the costs of object creation and allocation. We also learned that we can reduce these costs by eliminating objects or reusing them, or by designing our objects to use primitives. The next items present other ways to reduce the number of unnecessary objects in our applications.

## Item 48: Consider using object declarations

When you have a class without any instance-specific state, you can turn it into an **object declaration** to define a singleton. This means not defining a constructor and using the `object` keyword instead of `class`. We reference the singleton object using the name of the object declaration.

```
object Singleton {
    fun doSomething() {
        // ...
    }
}

fun main() {
    val obj = Singleton
    obj.doSomething()

    Singleton.doSomething()
}
```

Object declarations are useful to limit the number of created objects. This is especially useful for classes that are created many times in your projects, like events or markers. Thanks to object declarations, we can be sure that they have only one instance.

```
sealed class ValueChange<out T>
data object Keep : ValueChange<Nothing>()
data object SetDefault : ValueChange<Nothing>()
data object SetEmpty : ValueChange<Nothing>()
data class Set<out T>(val value: T) : ValueChange<T>()

sealed class ManagerMessage
data class CodeProduced(val code: String) : ManagerMessage()
data object ProductionStopped : ManagerMessage()

sealed interface AdView
data object FacebookAd : AdView
data object GoogleAd : AdView
data class OwnAd(val text: String, val imgUrl: String) : AdView
```

It's a bit more challenging when you want to turn a class that has some generic parameter types, like `DeleteAll` in the following example, into an object declaration:

```
sealed interface StoreMessage<T>
data class Save<T>(val data: T) : StoreMessage<T>
data class DeleteAll<T> : StoreMessage<T>
```

In such cases, it is popular to use the pattern called Covariant Nothing Object<sup>70</sup>. To use it, we need to make the type parameter of the supertype class covariant (so, use the `out` modifier next to `T` in the `StoreMessage` declaration), then use `Nothing` as a type argument for the object declaration:

```
sealed interface StoreMessage<out T>
data class Save<T>(val data: T) : StoreMessage<T>
data object DeleteAll : StoreMessage<Nothing>
```

Since `Nothing` is a subtype of all types in Kotlin, and `T` in `StoreMessage` is covariant, `StoreMessage<Nothing>` is a subtype of all `StoreMessage<T>` types. This means that `DeleteAll` is a subtype of `StoreMessage<T>` for all `T` types.

```
val deleteAllInt: StoreMessage<Int> = DeleteAll
val deleteAllString: StoreMessage<String> = DeleteAll
```

This pattern is used in many projects and libraries, including Kotlin stdlib. For instance, `EmptyList` is an object declaration that is a subtype of `List<Nothing>`; as a result, it is a subtype of all `List<T>` types. This way, there is only one instance of an empty list in the whole application.

```
internal object EmptyList : List<Nothing> {
    // ...
}

val emptyListInt: List<Int> = EmptyList
val emptyListString: List<String> = EmptyList
```

## Summary

- To define singletons, turn classes without an instance-specific state into object declarations.
- Use the covariant nothing object pattern to turn classes with generic type parameters into object declarations.

---

<sup>70</sup>I defined and described this in detail in the book Advanced Kotlin.

## Item 49: Use caching when possible

If there is one performance optimization you should know, it is certainly caching. It is used in many different ways on many different levels. Your computer has a CPU cache and a disk cache. Your browser has a cache for web pages. Our network providers have caches for different types of data. JVM has many different caches that are used to speed up our applications. We define our own caches in both backend and Android applications. Caching is everywhere because it's a very powerful technique. I would even say that **caching is the most powerful way to speed up applications.**

The idea behind caching is simple. A cache contains a redundant copy of data that is stored such that we can access it quickly. Let me show you a typical example. Imagine that your application needs to fetch users by their id from a web service. You can implement this in the following way:

```
class WebUserRepository(  
    val userClient: UserClient  
) : UserRepository {  
    override suspend fun getUser(id: Int): User =  
        userClient.fetchUser(id)  
}
```

The problem is that for each user we want to get, we need to send a network request and wait for the response. One way to improve this is by storing the results of previous requests so we don't need to send a network request every time we want to get a user with a certain id. We can do this using a map:

```
class CachedWebUserRepository(  
    val userClient: UserClient  
) : UserRepository {  
    private val users = ConcurrentHashMap<Int, User>()  
  
    override suspend fun getUser(id: Int): User =  
        users.getOrPut(id) { userClient.fetchUser(id) }  
}
```

This is a simple implementation of a cache. Our map represents redundant memory because if we cleared it, no data would really be lost and our repository would just need to fetch it again. We can also see that we can access data from a cache quickly as we don't need to send a network request. However, there are two problems with caches. Firstly, if we cache data that changes, our cache can

become stale. For instance, if we cache users and a user's name changes, the data in our cache will be outdated. The standard solution to this problem is to use a caching library (like Caffeine or Ehcache) that allows us to specify how long a certain record should be considered valid. For instance, you can specify that our user record should be considered valid for one minute. After that, we will need to fetch it again. Such a solution makes a lot of sense on backend applications where users often fetch the same data repeatedly but the data rarely change. This is a typical example of using a cache on the backend:

```
class CachedWebUserRepository(
    val userClient: UserClient
) : UserRepository {

    private val users = Caffeine.newBuilder()
        .expireAfterWrite(1, TimeUnit.MINUTES)
        .buildSuspend<Int, User>()

    override suspend fun getUser(id: Int): User =
        users.get(id) { userClient.fetchUser(id) }
}
```

Using time-based expiration is far less popular on Android, where we tend to use caching for data that does not change, like configurations or database connections, or data that are only changed in an application, where we can update cached objects when they change.

```
private val connections =
    ConcurrentHashMap<String, Connection>()

fun getConnection(host: String) =
    connections.getOrPut(host) { createConnection(host) }
```

The second problem with caching is that it essentially means we are buying performance in exchange for memory. If we cache too much data, we can run out of memory, but there are a couple of tricks that can help us with this problem. One is to expire cache entries that are used less often (expire after write) or to set a cache size limit. However, the most powerful technique is making your cache use soft or even weak references. Let me explain this.

In Kotlin, when a variable references a value, it is a strong reference, so the existence of this reference prevents the garbage collector from cleaning up the value. However, JVM also offers two other kinds of references:

- A weak reference does not prevent the Garbage Collector from cleaning up a value. So, if no other reference is using this value, it will be cleaned up.
- A soft reference does not guarantee that a value won't be cleaned up by the GC either, but in most JVM implementations this value won't be cleaned up unless memory is needed.

If you are concerned about memory usage, the simplest way is to use a soft reference cache as this will not be limited when there is enough memory, but it will be cleaned up when memory is needed.

```
class CachedWebUserRepository(  
    val userClient: UserClient  
) : UserRepository {  
  
    private val users = Caffeine.newBuilder()  
        .maximumSize(10_000)  
        // When size is reached, less used entries are removed  
        .expireAfterAccess(10, TimeUnit.MINUTES)  
        //When entry is not used for 10 minutes, it is removed  
        .softValues() // Using soft references  
        .buildSuspended<Int, User>()  
  
    override suspend fun getUser(id: Int): User =  
        users.get(id) { userClient.fetchUser(id) }  
}
```

## Summary

- Use caching to speed up data access and reduce the number of heavy requests (like web and file system requests).
- Use a caching library to avoid common pitfalls and to get more features.
- Use time-based expiration to avoid stale data and to limit the size of the cache.
- Use a cache size limit and soft references to avoid running out of memory.

## Item 50: Extract objects that can be reused

A very useful trick for performance is lifting heavy operations to an outer scope to allow reusing them. For instance, here is a function that counts the number of values equal to the maximum value:

```
fun <T : Comparable<T>> Iterable<T>.countMax(): Int =
    count { it == this.maxOrNull() }
```

A better solution is to extract the biggest element to the level of the `countMax` function:

```
fun <T : Comparable<T>> Iterable<T>.countMax(): Int {
    val max = this.maxOrNull()
    return count { it == max }
}
```

This solution is better for performance because we don't need to find the biggest element on the receiver in every iteration. Notice that it also improves readability by making it clear that `max` is called on the extension receiver, therefore it is the same throughout all iterations.

Extracting a value to an outer scope so as to not recalculate it unnecessarily is an important practice. This might sound obvious, but it is not always very clear. Just take a look at this function, where we use a regex to determine whether a string contains a valid IP address:

```
fun String.isValidIpAddress(): Boolean {
    return this.matches(
        ("\\A(?:25[0-5]|2[0-4][0-9]|" +
         "[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|" +
         "[01]?[0-9][0-9]?)\\z").toRegex()
    )
}

// Usage
print("5.173.80.254".isValidIpAddress()) // true
```

The problem with this function is that the `Regex` object needs to be created every time we use it. This is a serious disadvantage since regex pattern compilation is a complex operation. This is why this function is not suitable for repeated use in performance-constrained parts of our code. However, we can improve it by lifting the regex up to the top level:

```
private val IS_VALID_IP_REGEX = "\\\A(?:(?:25[0-5]|2[0-4])" +
    "[0-9] | [01]?[0-9][0-9]?)\\\.){3}(?:25[0-5]|2[0-4][0-9] |"+
    "[01]?[0-9][0-9]?)\\z".toRegex()

fun String.isValidIpAddress(): Boolean =
    matches(IS_VALID_IP_REGEX)
```

If this function is in a file together with some other functions and we don't want to create this object unless it is used, we can even initialize the regex lazily:

```
private val IS_VALID_IP_REGEX by lazy {
    ("\\\\A(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\\\.){3}" +
    "(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\\z").toRegex()
}
```

Making properties lazy is also useful when we are dealing with classes.

## Lazy initialization

Often, when we need to create a heavy class, it is better to do so lazily. For instance, imagine that class A needs instances of B, C, and D, all of which are heavy. If we just create them during class creation, the creation of A will be very heavy because it will need to create B, C, and D and then the rest of its body. Therefore, the heaviness of object creation will just accumulate.

```
class A {
    val b = B()
    val c = C()
    val d = D()

    //...
}
```

There is a cure though. We can just initialize these heavy objects lazily:

```
class A {  
    val b by lazy { B() }  
    val c by lazy { C() }  
    val d by lazy { D() }  
  
    // ...  
}
```

Each object will then be initialized just before its first usage. The cost of creating these objects will be spread instead of accumulated.

## Summary

- Extracting objects to an outer scope can improve performance by avoiding unnecessary recalculation.
- Lazy initialization can be used to avoid heavy object creation during class initialization.

## Item 51: Use the inline modifier for functions with parameters of functional types

You might have noticed that nearly all Kotlin higher-order stdlib functions have an inline modifier.

```
public inline fun repeat(times: Int, action: (Int) -> Unit) {  
    for (index in 0 until times) {  
        action(index)  
    }  
}  
  
public inline fun <T, R> Iterable<T>.map(  
    transform: (T) -> R  
) : List<R> {  
    return mapTo(  
        ArrayList<R>(collectionSizeOrDefault(10)),  
        transform  
    )  
}  
  
public inline fun <T> Iterable<T>.filter(  
    predicate: (T) -> Boolean  
) : List<T> {  
    return filterTo(ArrayList<T>(), predicate)  
}
```

This `inline` modifier makes the compiler replace all uses of this function with its body during compilation. Also, all calls of function arguments inside `repeat` are replaced with these functions' bodies. So, the following `repeat` function call:

```
repeat(10) {  
    print(it)  
}
```

Will be replaced with the following code during compilation:

```
for (index in 0 until 10) {  
    print(index)  
}
```

This is a significant change compared to how functions are executed normally. In a normal function, execution jumps into this function body, invokes all statements, then jumps back to the place from where the function was invoked. Replacing calls with bodies is a significantly different behavior.

There are a few advantages of this behavior:

1. A type argument can be reified
2. Functions with functional parameters are faster when they are inline
3. Non-local return is allowed

There are also some costs to using this modifier. Let's review all the advantages and costs of the `inline` modifier.

## A type argument can be reified

Older versions of Java do not have generics. They were added to the Java programming language in 2004 in version J2SE 5.0, but they are still not present in the JVM bytecode, therefore generic types are erased during compilation. For instance, `List<Int>` compiles to `List`. This is why we cannot check if an object is `List<Int>`. We can only check if it is a `List`.

```
any is List<Int> // Error  
any is List<*> // OK
```

```
if(any is List<Int>) {
```

Cannot check for instance of erased type: List<Int>

For the same reason, we cannot operate on a type argument:

```
fun <T> printTypeName() {  
    print(T::class.simpleName) // ERROR  
}
```

We can overcome this limitation by making a function inline. Function calls are replaced with this function's body, so uses of type parameters can be replaced with type arguments using the `reified` modifier:

```
inline fun <reified T> printTypeName() {
    print(T::class.simpleName)
}

// Usage
printTypeName<Int>()      // Int
printTypeName<Char>()      // Char
printTypeName<String>()    // String
```

During compilation, the body of `printTypeName` replaces usages, and the reified type argument replaces the type parameter:

```
print(Int::class.simpleName) // Int
print(Char::class.simpleName) // Char
print(String::class.simpleName) // String
```

`reified` is a useful modifier. For instance, it is used in `filterIsInstance` from the stdlib to filter only elements of a certain type:

```
class Worker
class Manager

val employees: List<Any> =
    listOf(Worker(), Manager(), Worker())

val workers: List<Worker> =
    employees.filterIsInstance<Worker>()
```

`reified` modifier is also used in many libraries and util functions we define ourselves. The example below presents a common implementation of `fromJsonOrNull` that uses the Gson library. It also presents how the Koin library uses this kind of function to simplify both dependency injection and module declaration.

```

inline fun <reified T : Any> String.fromJsonOrNull() : T? =
    try {
        gson.fromJson(json, T::class.java)
    } catch (e: JsonSyntaxException) {
        null
    }

// usage
val user: User? = userAsText.fromJsonOrNull()

// Koin module declaration
val myModule = module {
    single { Controller(get()) } // get is reified
    single { BusinessService() }
}

// Koin injection
val service: BusinessService by inject()
// inject is reified

```

## Functions with functional parameters are faster when they are inlined

To be more concrete, all functions are slightly faster when they are inlined. There is no need to jump with execution and track the back-stack. This is why small functions that are used very often in the stdlib are often inlined:

```

inline fun print(message: Any?) {
    System.out.print(message)
}

```

However, this difference is most likely insignificant when a function does not have any functional parameters. This is why IntelliJ gives this warning:



91 **inline fun** printThree() {  
 **System.out.print**(message)  
}  
Expected performance impact of inlining 'public inline fun printThree(): Unit defined in org.kotlinacademy in file InlineRepeatBenchmark.kt' is insignificant. Inlining works best for functions with parameters of functional types

To understand why functions with functional parameters are typically faster when marked as inline, we first need to understand what the problem is with operating on functions as objects. These kinds of objects, which are created using function literals, need to be held somehow. In Kotlin/JS, this is simple since JavaScript treats functions as first-class citizens, so there are either functions or

function references under functional parameters. In Kotlin/JVM, an object needs to be created using either an anonymous JVM class or a normal class. Therefore, the following lambda expression will be compiled to a class.

```
// kotlin
val lambda: () -> Unit = {
    // code
}

// compiled to JVM equivalent of
Function0<Unit> lambda = new Function0<Unit>() {
    public Unit invoke() {
        // code
    }
};
```

Notice that this function type is translated to the `Function0` type as this is what a function type with no arguments is compiled to in JVM. Functions with more arguments compile to `Function1`, `Function2`, `Function3`, etc.

- `() -> Unit` compiles to `Function0<Unit>`
- `() -> Int` compiles to `Function0<Int>`
- `(Int) -> Int` compiles to `Function1<Int, Int>`
- `(Int, Int) -> Int` compiles to `Function2<Int, Int, Int>`

All these interfaces are generated by the Kotlin compiler. You cannot use them explicitly in Kotlin though because they are generated on demand, so we should use function types instead. However, knowing that function types are just interfaces opens your eyes to some new possibilities. You can, for instance, implement a function type:

```
class OnClickListener : () -> Unit {
    override fun invoke() {
        // ...
    }
}
```

As illustrated in Item 47: Avoid unnecessary object creation, wrapping the body of a function into an object will slow down the code. This is why the first of the two functions below will be faster:

```
inline fun repeat(times: Int, action: (Int) -> Unit) {
    for (index in 0 until times) {
        action(index)
    }
}

fun repeatNoinline(times: Int, action: (Int) -> Unit) {
    for (index in 0 until times) {
        action(index)
    }
}
```

The difference is visible but it is rarely significant in real-life examples. However, if we design our test well, you can see this difference clearly:

```
@Benchmark
// On average 189 ms
fun nothingInline(blackhole: Blackhole) {
    repeat(100_000_000) {
        blackhole.consume(it)
    }
}

@Benchmark
// On average 447 ms
fun nothingNoninline(blackhole: Blackhole) {
    noinlineRepeat(100_000_000) {
        blackhole.consume(it)
    }
}
```

On my computer, the first one takes 189 ms on average, while the second one takes 447 ms on average. This difference stems from the fact that in the first function we only iterate over numbers and call `consume` function (which is empty). In the second function, we call a method that iterates over numbers and calls an object, and this object calls `consume` function. All this difference is due to the fact that we use an extra object (Item 47: Avoid unnecessary object creation).

To show a more typical example, let's say that we have 5,000 products, and we need to sum up the prices of the ones that have been bought. We can do this simply by:

```
users.filter { it.bought }.sumByDouble { it.price }
```

On my machine, it takes 38 ms to calculate on average. How much would it be if the `filter` and `sumByDouble` functions were not inline? 42 ms on average on my machine. This doesn't look like a lot, but this is around 10% difference every time you use these methods for collection processing.

A more significant difference between inline and non-inline functions manifests itself when we capture local variables in function literals. A captured value needs to be wrapped into some object, and whenever it is used, this needs to happen through this object. For instance, in the following code:

```
var l = 1L
noinlineRepeat(100_000_000) {
    l += it
}
```

A local variable cannot be used directly in a non-inline lambda. This is why the value of `a` will be wrapped into a reference object during compilation:

```
val a = Ref.LongRef()
a.element = 1L
noinlineRepeat(100_000_000) {
    a.element = a.element + it
}
```

This is a more significant difference because such objects might be used many times: every time we use a function created by a function literal. For instance, in the above example, we use `a` twice, therefore the extra object will be used  $2 * 100,000,000$  times. To see this difference, let's compare the following functions:

```
@Benchmark
// On average 30 ms
fun nothingInline(blackhole: Blackhole) {
    var l = 0L
    repeat(100_000_000) {
        l += it
    }
    blackhole.consume(l)
}
```

```
@Benchmark
```

```
// On average 274 ms
fun nothingNoninline(blackhole: Blackhole) {
    var l = 0L
    noinlineRepeat(100_000_000) {
        l += it
    }
    blackhole.consume(l)
}
```

On my machine, the first one takes 30 ms, while the second takes 274 ms. This is due to the accumulated effects of the fact that a function is an object and the local variable needs to be wrapped. These objects make tiny barriers that need to be overcome many times, again and again, and this makes a significant difference in the end. Since in most cases we don't know how functions with parameters of functional types will be used, when we define a utility function with such parameters, for instance for collection processing, it is good practice to make it inline. This is why most extension functions with parameters of functional types in the stdlib are `inline`.

## Non-local return is allowed

The previously defined `noinlineRepeat` looks much like a control structure. Just compare it with an `if` expression or a `for` loop:

```
if (value != null) {
    print(value)
}

for (i in 1..10) {
    print(i)
}

repeatNoninline(10) {
    print(it)
}
```

One significant difference is that a return is not allowed inside:

```
fun main() {
    noinlineRepeat(10) {
        print(it)
        return // ERROR: Not allowed
    }
}
```

This is the result of what function literals are compiled to. We cannot return from `main` if our code is located in another class. There is no such limitation when a function literal is inlined as the code will be located in the `main` function anyway.

```
fun main() {
    repeat(10) {
        print(it)
        return // OK
    }
}
```

Thanks to that, functions can look and behave more like control structures:

```
fun getSomeMoney(): Money? {
    repeat(100) {
        val money = searchForMoney()
        if (money != null) return money
    }
    return null
}
```

## Costs of inline modifiers

Inline is a useful modifier, but it should not be used everywhere due to its costs and limitations. Let's review them.

**Inline functions cannot use elements that have restricted visibility.** We cannot use private or internal functions or properties in public inline functions. We cannot use private properties in public or inline functions.

```
internal inline fun read() {  
    val reader = Reader() // Error  
    // ...  
}  
  
private class Reader {  
    // ...  
}
```

This is why they cannot be used to hide implementation, so they are rarely used in classes. This is why inline functions are mostly utility functions.

**Inline functions cannot be recursive.** Otherwise, they would replace their calls infinitely. Recurrent cycles are especially dangerous because, at the moment, they do not show an error in IntelliJ:

```
inline fun a() {  
    b()  
}  
inline fun b() {  
    c()  
}  
inline fun c() {  
    a()  
}
```

**Inline functions make our code grow.** To see the scale of this growth, let's say that I really like printing 3. I first defined the following function:

```
inline fun printThree() {  
    print(3)  
}
```

I wanted to call it 3 times, so I added this function:

```
inline fun threePrintThree() {  
    printThree()  
    printThree()  
    printThree()  
}
```

I still wasn't satisfied, so I defined the following functions:

```
inline fun threeThreePrintThree() {  
    threePrintThree()  
    threePrintThree()  
    threePrintThree()  
}  
  
inline fun threeThreeThreePrintThree() {  
    threeThreePrintThree()  
    threeThreePrintThree()  
    threeThreePrintThree()  
}
```

What are they all compiled to? The first two are very readable:

```
inline fun printThree() {  
    print(3)  
}  
  
inline fun threePrintThree() {  
    print(3)  
    print(3)  
    print(3)  
}
```

The next two were compiled to the following functions:

```
inline fun threeThreePrintThree() {  
    print(3)  
    print(3)  
    print(3)  
    print(3)  
    print(3)  
    print(3)  
    print(3)  
    print(3)  
}  
  
inline fun threeThreeThreePrintThree() {  
    print(3)  
    print(3)
```

```
print(3)
}
}
```

This is an abstract example, but it shows a big problem with inline functions: code grows really quickly when we overuse them. I have actually encountered this problem in a real-life project. Having too many inline functions calling each other is dangerous because our code might start growing exponentially.

## Crossinline and noinline

There are cases in which we want to inline a function, but for some reason, we cannot inline all functions used as arguments. In such cases, we can use the following modifiers:

- `crossinline` - this means that the function should be inlined but non-local return is not allowed. We use it when this function is used in another scope where non-local return is not allowed; for instance, in another lambda that is not inlined.

- `noinline` - this means that this argument should not be inlined at all. It is used mainly when we use this function as an argument to another function that is not inlined.

```

inline fun requestNewToken(
    hasToken: Boolean,
    crossinline onRefresh: () -> Unit,
    noinline onGenerate: () -> Unit
) {
    if (hasToken) {
        httpCall("get-token", onGenerate) // We must use
        // noinline to pass function as an argument to a
        // function that is not inlined
    } else {
        httpCall("refresh-token") {
            onRefresh() // We must use crossinline to
            // inline function in a context where
            // non-local return is not allowed
            onGenerate()
        }
    }
}

fun httpCall(url: String, callback: () -> Unit) {
    /*...*/
}

```

It is good to know what the meaning of both modifiers is, but we can live without remembering them as IntelliJ IDEA suggests them when they are needed:

The screenshot shows a code editor with Java code. Line 26 contains the code `onRefresh()`. A yellow circle with a question mark icon is positioned over the opening parenthesis of the method call. A tooltip message appears below the code: "Can't inline 'onRefresh' here: it may contain non-local returns. Add 'crossinline' modifier to parameter declaration 'onRefresh'".

```

17 inline fun requestNewToken(
18     hasToken: Boolean,
19     onRefresh: ()->Unit,
20     onGenerate: ()->Unit
21 ) {
22     if (hasToken) {
23         httpCall("get-token", onGenerate)
24     } else {
25         httpCall("refresh-token") {
26             onRefresh()
27         }
28     }
29 }

```

## Summary

The main cases in which we use inline functions are:

- Very frequently used functions, like `print`.
- Functions that need to have a reified type passed as a type argument, like `filterIsInstance`.
- When we define top-level functions with parameters of functional types, especially helper functions, like collection processing functions (like `map`, `filter`, `flatMap`, `joinToString`), scope functions (like `also`, `apply`, `let`), or top-level utility functions (like `repeat`, `run`, `with`).

We rarely use inline functions to define an API, and we should be careful when one inline function calls some other inline functions.

## Item 52: Consider using inline value classes

Not only functions can be inlined: objects holding a single value can also be replaced with this value. To do this, we need to define a class with a single read-only primary constructor property, a modifier `value`, and a `JvmInline` annotation.

```
@JvmInline
value class Name(private val value: String) {
    // ...
}
```

Value classes were introduced in Kotlin 1.5 due to Java's plans to introduce value classes. Before that (but since Kotlin 1.3), we could use an `inline` modifier to achieve a similar result.

Such a class will be replaced with the value it holds whenever possible:

```
// Code
val name: Name = Name("Marcin")

// During compilation replaced with code similar to:
val name: String = "Marcin"
```

Methods from such a class will be evaluated as static methods:

```
@JvmInline
value class Name(private val value: String) {
    // ...

    fun greet() {
        print("Hello, I am $value")
    }
}

// Code
val name: Name = Name("Marcin")
name.greet()

// During compilation replaced with code similar to:
val name: String = "Marcin"
Name.`greet-impl`(name)
```

We can use inline value classes to make a wrapper around some type (like `String` in the above example) with no performance overhead ([Item 47: Avoid unnecessary object creation](#)). Two especially popular uses of inline value classes are:

- To indicate a unit of measure.
- To use types to protect users from value misuse.
- To optimize for memory usage.

Let's discuss these separately.

## Indicate unit of measure

Imagine that you need to use a method to set up a timer:

```
interface Timer {  
    fun callAfter(time: Int, callback: () -> Unit)  
}
```

What is this `time`? It might be a time in milliseconds, seconds, or minutes; it is not clear at this point, so it is easy to make a mistake. A serious mistake. One famous example of such a mistake is the Mars Climate Orbiter, which plowed into the Martian atmosphere. The reason for this was that the software used to control it was developed by an external company, and it produced outputs in different measurement units than those expected by NASA. It produced results in pound-force seconds ( $\text{lbf}\cdot\text{s}$ ), while NASA expected newton-seconds ( $\text{N}\cdot\text{s}$ ). The total cost of the mission was 327.6 million USD, and it was a complete failure. As you can see, confusion of measurement units can be really expensive.

One common way for developers to suggest a measurement unit is by including it in the parameter name:

```
interface Timer {  
    fun callAfter(timeMillis: Int, callback: () -> Unit)  
}
```

This is better, but it still leaves some space for mistakes. For example, the property name is often not visible when a function is used. Another problem is that indicating the type in this way is harder when the type is returned. In the example below, the time is returned from `decideAboutTime` but its measurement unit is not indicated at all. It might return the time in minutes, thus we will not set the time correctly.

```
interface User {
    fun decideAboutTime(): Int
    fun wakeUp()
}

interface Timer {
    fun callAfter(timeMillis: Int, callback: () -> Unit)
}

fun setUpUserWakeUpUser(user: User, timer: Timer) {
    val time: Int = user.decideAboutTime()
    timer.callAfter(time) {
        user.wakeUp()
    }
}
```

We might introduce the measurement unit of the returned value in the function name, for instance by naming it `decideAboutTimeMillis`; however, this solution is not considered very good as it makes this function provide low-level information even when we don't need it. Moreover, it does not necessarily solve the problem as a developer still needs to ensure that the measurement units match.

A better way to solve this problem is to introduce stricter types that will protect us from misusing types, and to make them efficient we can use inline value classes:

```
@JvmInline
value class Minutes(val minutes: Int) {
    fun toMillis(): Millis = Millis(minutes * 60 * 1000)
    // ...
}

@JvmInline
value class Millis(val milliseconds: Int) {
    // ...
}

interface User {
    fun decideAboutTime(): Minutes
    fun wakeUp()
}

interface Timer {
    fun callAfter(timeMillis: Millis, callback: () -> Unit)
```

```

}

fun setUpUserWakeUpUser(user: User, timer: Timer) {
    val time: Minutes = user.decideAboutTime()
    timer.callAfter(time) { // ERROR: Type mismatch
        user.wakeUp()
    }
}

```

This would force us to use the correct type:

```

fun setUpUserWakeUpUser(user: User, timer: Timer) {
    val time = user.decideAboutTime()
    timer.callAfter(time.toMillis()) {
        user.wakeUp()
    }
}

```

This is especially useful for metric units. For instance, on the frontend we often use a variety of units like pixels, millimeters, dp, etc. To support object creation, we can define DSL-like extension properties (you can make them inline as well):

```

inline val Int.min
    get() = Minutes(this)

inline val Int.ms
    get() = Millis(this)

val timeMin: Minutes = 10.min

```

Regarding indicating the amount of time, we can also use the `Duration` class from the standard library, which is an inline value class, and that offers DSL-like extension properties:

```
val time: Duration = 10.minutes
```

## Protect us from value misuse

It is a popular practice in bigger projects to use a wrapper around primitive types to protect us from misusing them. For instance, let's say that you write an application for a university in which each student is identified by a unique ID but

is also associated with a class id. Both might be represented as raw strings, but it would be easy to make a mistake and use the class id instead of the student id. To avoid this, we should define wrappers over different kinds of ids, and we make these inline value classes to avoid performance overhead:

```
@JvmInline
value class StudentId(val value: String)

@JvmInline
value class ClassId(val value: String)

data class Student(val id: StudentId, val classId: ClassId)
```

## Optimize for memory usage

As we learned in Item 47: Avoid unnecessary object creation, using primitive types instead of wrapped types is an optimization. However, operating on primitives can be harder. To have your cake and eat it too, we can use inline value classes to wrap primitives and operate on them as if they were objects:

```
@JvmInline
value class OptionalDouble(val value: Double) {

    fun isNaN() = value.isNaN()

    companion object {
        const val UNDEFINED_VALUE = Double.NaN
        val Undefined = OptionalDouble(UNDEFINED_VALUE)
    }
}
```

## Inline value classes and interfaces

Inline value classes can implement interfaces. We could use this in the example presented above to avoid transforming from one type to another.

```
interface TimeUnit {
    val millis: Long
}

@JvmInline
value class Minutes(val minutes: Long) : TimeUnit {
    override val millis: Long get() = minutes * 60 * 1000
    // ...
}

@JvmInline
value class Millis(val milliseconds: Long) : TimeUnit {
    override val millis: Long get() = milliseconds
}

// the type under the hood is TimeUnit
fun setUpTimer(time: TimeUnit) {
    val millis = time.millis
    //...
}

setUpTimer(Minutes(123))
setUpTimer(Millis(456789))
```

The catch is that when an object is used through an interface, it cannot be inlined. Therefore, in the above example, there is no advantage to using inline value classes since wrapped objects need to be created to let us present a type through this interface. **When we present inline value classes through an interface, such classes are not inlined.**

Another situation in which a type will not be inlined is when it is nullable and the value class holds a primitive as a parameter. In the example below, when `Millis` is used as a parameter type, it will be replaced with `Long`. However, if `Millis?` is used, it cannot be replaced because `Long` cannot be `null`. But if `Millis` held a non-primitive type, like `String`, then its type nullability wouldn't influence inlining.

```
@JvmInline
value class Millis(val milliseconds: Long) {
    val millis: Long get() = milliseconds
}

// the type under the hood is @Nullable Millis
fun setUpTimer(time: Millis?) {
    val millis = time?.millis
    //...
}

// the type under the hood is long
fun setUpTimer(time: Millis) {
    val millis = time.millis
    //...
}

fun main() {
    setUpTimer(Millis(456789))
}
```

## Typealias

Kotlin's typealias lets us create another name for a type:

```
typealias NewName = Int

val n: NewName = 10
```

Naming types is a useful capability that is used especially when we deal with long and repeatable types. For instance, it is a popular practice to name repeatable function types:

```
typealias ClickListener =
    (view: View, event: Event) -> Unit

class View {
    fun addClickListener(listener: ClickListener) {}
    fun removeClickListener(listener: ClickListener) {}
    //...
}
```

What needs to be understood though is that **type aliases do not protect us in any way from type misuse**. They just add a new name for a type. If we named `Int` as both `Millis` and `Seconds`, we would create the illusion that the type system protects us, but it does not:

```
typealias Seconds = Int
typealias Millis = Int

fun getTime(): Millis = 10
fun setUpTimer(time: Seconds) {}

fun main() {
    val seconds: Seconds = 10
    val millis: Millis = seconds // No compiler error

    setUpTimer(getTime())
}
```

In the above example, it would be easier to find what is wrong without using type aliases. This is why they should not be used this way. To indicate a unit of measure, use a parameter name or classes: a name is cheaper, but classes give better safety. When we use inline value classes, we take the best from both options: they are both cheap and safe.

## Summary

Inline value classes let us wrap a type without a performance overhead. Therefore, we improve safety by making our type system protect us from value misuse. If you use a type whose meaning is unclear (like `Int` or `String`), especially a type that might have different units of measure, consider wrapping it with inline value classes.

## Item 53: Eliminate obsolete object references

Programmers who are used to languages with automatic memory management rarely think about freeing objects. In Java, for example, the Garbage Collector (GC) does this job. However, forgetting about memory management often leads to memory leaks (unnecessary memory consumption) and in some cases to the `OutOfMemoryError`. The single most important rule is that we should not keep a reference to an object that is not useful anymore, especially if such an object is big in terms of memory or if there might be a lot of instances of such objects.

In Android, there is a common mistake made by beginners: since a reference to `Activity` (a concept similar to a window in a desktop application) is needed in many Android functionalities, they store it in a companion object or a top-level property for convenience:

```
class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //...
        activity = this
    }

    //...

    companion object {
        // DON'T DO THIS! It is a huge memory leak
        var activity: MainActivity? = null
    }
}
```

Holding a reference to an activity in a companion object does not let the Garbage Collector release it if our application is running. Activities are heavy objects, so this is a huge memory leak. There are some ways to improve this situation, but it is best not to hold such resources statically at all. **Manage dependencies properly instead of storing them statically**. Also, notice that we might cause a memory leak when we hold an object that stores a reference to another one. In the example below, we hold a lambda function that captures a reference to the `MainActivity`:

```

class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //...

        // Be careful, we leak a reference to `this`
        logError = {
            Log.e(
                this::class.simpleName,
                it.message
            )
        }
    }

    //...
}

companion object {
    // DON'T DO THIS! A memory leak
    var logError: ((Throwable) -> Unit)? = null
}
}

```

However, problems with memory can be much more subtle. Take a look at the stack implementation below<sup>71</sup>:

```

class Stack {
    private var elements: Array<Any?> =
        arrayOfNulls(DEFAULT_INITIAL_CAPACITY)
    private var size = 0

    fun push(e: Any) {
        ensureCapacity()
        elements[size++] = e
    }

    fun pop(): Any? {
        if (size == 0) {
            throw EmptyStackException()
        }
    }
}

```

---

<sup>71</sup>Example inspired by the book Effective Java by Joshua Bloch.

```
        return elements[--size]
    }

    private fun ensureCapacity() {
        if (elements.size == size) {
            elements = elements.copyOf(2 * size + 1)
        }
    }

    companion object {
        private const val DEFAULT_INITIAL_CAPACITY = 16
    }
}
```

Can you spot a problem here? Take a minute to think about it.

The problem is that when we pop, we just decrement the size of this stack, but we don't free an element in the array. Let's say that we have 1,000 elements on the stack and we pop nearly all of them, one after another, until the size of the stack is 1. Now, we can access only one element, and so we should hold only one element. However, our stack still holds 1,000 elements and doesn't allow the GC to destroy them. All these objects are wasting our memory. This is why they are called memory leaks. If these leaks accumulate, we might face an `OutOfMemoryError`. How can we fix this implementation? A very simple solution is to set `null` in the array when an object is not needed anymore:

```
fun pop(): Any? {
    if (size == 0)
        throw EmptyStackException()
    val elem = elements[--size]
    elements[size] = null
    return elem
}
```

We should recognize and release the values that are not needed anymore. This rule applies in a surprising number of classes. To see another example, let's say that we need a `mutableLazy` property delegate. It should work just like `lazy`, but it should also allow property state mutation. I can define it using the following implementation:

```
fun <T> mutableLazy(
    initializer: () -> T
): ReadWriteProperty<Any?, T> =
    MutableLazy(initializer)

private class MutableLazy<T>(
    val initializer: () -> T
) : ReadWriteProperty<Any?, T> {

    private var value: T? = null
    private var initialized = false

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T {
        synchronized(this) {
            if (!initialized) {
                value = initializer()
                initialized = true
            }
            return value as T
        }
    }

    override fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: T
    ) {
        synchronized(this) {
            this.value = value
            initialized = true
        }
    }
}

// usage
var game: Game? by mutableLazy { readGameFromSave() }

fun setUpActions() {
    startNewGameButton.setOnClickListener {

```

```
        game = makeNewGame()
        startGame()

    }

resumeGameButton.setOnClickListener {
    startGame()
}

}
}
```

The above implementation of `mutableLazy` works correctly, but it has one flaw: the `initializer` is not cleaned after usage. This means that it is held as long as the reference to an instance of `MutableLazy` exists, even though it is not useful anymore. This is how `MutableLazy` implementation can be improved:

```
fun <T> mutableLazy(
    initializer: () -> T
): ReadWriteProperty<Any?, T> =
    MutableLazy(initializer)

private class MutableLazy<T>(
    var initializer: (() -> T)?
) : ReadWriteProperty<Any?, T> {

    private var value: T? = null

    override fun getValue(
        thisRef: Any?,
        property: KProperty<*>
    ): T {
        synchronized(this) {
            val initializer = initializer
            if (initializer != null) {
                value = initializer()
                this.initializer = null
            }
            return value as T
        }
    }

    override fun setValue(
        thisRef: Any?,
        property: KProperty<*>,
        value: T
    ) {
    }
}
```

```
) {  
    synchronized(this) {  
        this.value = value  
        this.initializer = null  
    }  
}  
}  
}
```

When we set the initializer to `null`, the previous value can be recycled by the GC.

How important is this optimization? Well, we don't know. It depends on what is captured by our lambda expression. It might be insignificant in most cases, but in others it might be a costly memory leak. That is why we should remember to replace unneeded references with `null`, especially in general-purpose tools. `Stack` and `MutableLazy` are such tools. For such tools, we should care more about optimization, especially if we're creating a library. For instance, in all 3 implementations of a lazy delegate from Kotlin stdlib, we can see that initializers are set to `null` after usage:

```
private class SynchronizedLazyImpl<out T>(  
    initializer: () -> T, lock: Any? = null  
) : Lazy<T>, Serializable {  
    private var initializer: (() -> T)? = initializer  
    private var _value: Any? = UNINITIALIZED_VALUE  
    private val lock = lock ?: this  
  
    override val value: T  
        get() {  
            val _v1 = _value  
            if (_v1 !== UNINITIALIZED_VALUE) {  
                @Suppress("UNCHECKED_CAST")  
                return _v1 as T  
            }  
  
            return synchronized(lock) {  
                val _v2 = _value  
                if (_v2 !== UNINITIALIZED_VALUE) {  
                    @Suppress("UNCHECKED_CAST") (_v2 as T)  
                } else {  
                    val typedValue = initializer!!()  
                    _value = typedValue  
                    initializer = null  
                    typedValue  
                }  
            }  
        }  
}
```

```
        }
    }
}

override fun isInitialized(): Boolean =
    _value != UNINITIALIZED_VALUE

override fun toString(): String =
    if (isInitialized()) value.toString()
    else "Lazy value not initialized yet."

private fun writeReplace(): Any =
    InitializedLazyImpl(value)
}
```

To free memory as soon as possible, it is useful to define your variables in a narrower scope and not capture them (Item 4: Minimize the scope of variables). A variable in a function (or lambda expression) is garbage collected when this function completes (or when this variable is not needed anymore). A property is garbage collected when a class instance is garbage collected. We should avoid keeping heavy values in top-level or object properties because these are never garbage collected.

It might also be useful to use weak references to reference views. This is a popular trick on Android. A weak reference does not stop the GC from collecting the object, so it can be collected as soon as this element is not visible anymore. This is very useful if we assume that we are not interested in references to views that are not displayed anymore.

```
class BaseActivity: Activity() {
    private val errorDialog = WeakReference<Dialog?>(null)
    ...
}
```

The big problem is that memory leaks are sometimes hard to predict and don't manifest themselves until the application crashes. This is why we should search for leaks using special tools, the most basic of which is the heap profiler. There are also some libraries that help in the search for data leaks. For instance, a popular library for Android is LeakCanary, which shows a notification (to the developer) whenever a memory leak is detected.

# Chapter 8: Efficient collection processing

Collections are one of the most important concepts in programming. In iOS, one of the most important view elements, `UICollectionView`, is designed to represent a collection. Similarly, in Android, it is hard to imagine an application without `RecyclerView` or `Listview`. For example, when you need to code a news portal, you will have a list of news. Each new item will probably have a list of authors and a list of tags. Or, when you make an online shop, you start with a list of products, each of which will most likely have a list of categories and a list of different variants. When a user buys something, they use a basket, which is probably a collection of products and amounts. Then, the user needs to choose from a list of delivery options and a list of payment methods. Collections are everywhere in programming! Just think about your application and you will quickly see lots of collections.

This is also reflected in programming languages. Most modern languages have some collection literals:

```
// Python  
primes = [2, 3, 5, 7, 13]  
// Swift  
let primes = [2, 3, 5, 7, 13]
```

Collection processing was one of the most important functionalities of functional programming languages. The name of the Lisp programming language<sup>72</sup> stands for “list processing”. Most modern languages have good support for collection processing. This includes Kotlin, which has one of the most powerful sets of tools for collection processing. Just think of the following collection processing:

---

<sup>72</sup>Lisp is one of the oldest programming languages still in widespread use today. It's often referred to as the father of all functional programming languages. Today, the best-known general-purpose Lisp dialects are Clojure, Common Lisp, and Scheme.

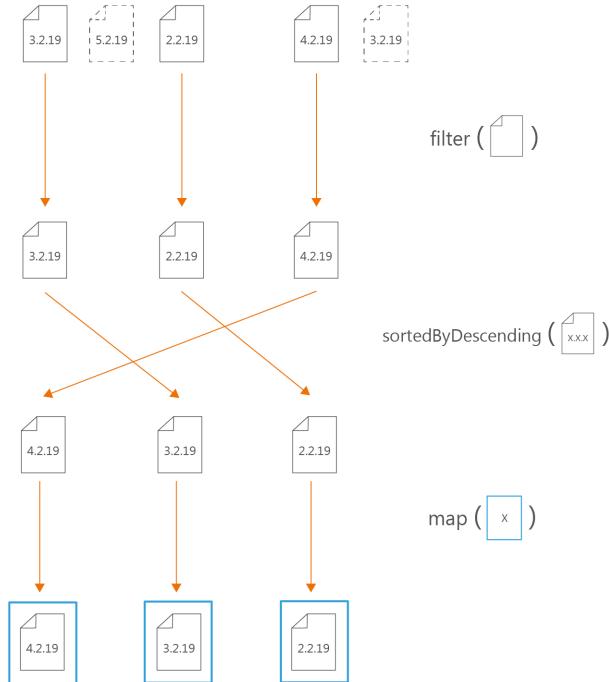
```
val visibleNews = mutableListOf<News>()
for (n in news) {
    if (n.visible) {
        visibleNews.add(n)
    }
}

Collections.sort(visibleNews,
    { n1, n2 -> n2.publishedAt - n1.publishedAt })
val newsItemAdapters = mutableListOf<NewsItemAdapter>()
for (n in visibleNews) {
    newsItemAdapters.add(NewsItemAdapter(n))
}
```

In Kotlin, it can be replaced with this:

```
val newsItemAdapters = news
    .filter { it.visible }
    .sortedByDescending { it.publishedAt }
    .map(::NewsItemAdapter)
```

Such notation is not only shorter but also more readable. Every step performs a concrete transformation on the list of elements. Here is a visualization of the above processing:



The performance of the above examples is very similar, but it is not always so simple. Kotlin has a lot of collection processing methods, so we can do the same processing in a lot of different ways. For instance, the processing implementations below have the same result but their performance is different:

```

fun productsListProcessing(): String =
    clientsList
        .filter { it.adult }
        .flatMap { it.products }
        .filter { it.bought }
        .map { it.price }
        .filterNotNull()
        .map { "$$it" }
        .joinToString(separator = " + ")
  
```

```
fun productsSequenceProcessing(): String =  
    clientsList.asSequence()  
        .filter { it.adult }  
        .flatMap { it.products.asSequence() }  
        .filter { it.bought }  
        .mapNotNull { it.price }  
        .joinToString(separator = " + ") { "$$it" }
```

Collection processing optimization is much more than just a brain puzzle. It is extremely important and is often performance-critical in big systems, which makes it important. As a consultant, I've seen a lot of projects, and my experience is that I see collection processing again and again in lots of different places. This is not something that can be easily ignored.

The good news is that collection processing optimization is not hard to master. There are some rules and a few things to remember, but actually, anyone can do it effectively. This is what we are going to learn in this chapter.

## Item 54: Prefer Sequences for big collections with more than one processing step

People often miss the difference between `Iterable` and `Sequence`. This is understandable since even their definitions are nearly identical:

```
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}

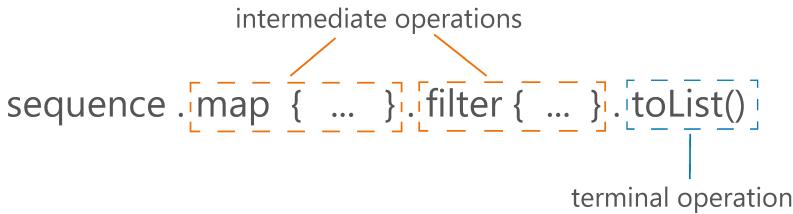
interface Sequence<out T> {
    operator fun iterator(): Iterator<T>
}
```

You can say that the only formal difference between them is their names. However, `Iterable` and `Sequence` are associated with totally different usages (have different contracts), so nearly all their processing functions work differently. Sequences are lazy, therefore intermediate functions for `Sequence` processing don't do any calculations. Instead, they return a new `Sequence` that decorates the previous one with the new operation. All these computations are evaluated during a terminal operation like `toList()` or `count()`. `Iterable` processing, on the other hand, returns a collection like `List` at every step.

```
public inline fun <T> Iterable<T>.filter(
    predicate: (T) -> Boolean
): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}

public fun <T> Sequence<T>.filter(
    predicate: (T) -> Boolean
): Sequence<T> {
    return FilteringSequence(this, true, predicate)
}
```

As a result, collection processing operations are invoked when they are used. `Sequence` processing functions are not invoked until the terminal operation (an operation that returns something different than `Sequence`). For instance, for `Sequence`, `filter` is an intermediate operation, so it doesn't do any calculations; instead, it decorates the sequence with the new processing step. Calculations are done in a terminal operation like `toList` or `sum`.



```

val list = listOf(1, 2, 3)
val listFiltered = list
    .filter { print("f$it "); it % 2 == 1 }
// f1 f2 f3
println(listFiltered) // [1, 3]

val seq = sequenceOf(1, 2, 3)
val filtered = seq.filter { print("f$it "); it % 2 == 1 }
println(filtered) // FilteringSequence@...

val asList = filtered.toList()
// f1 f2 f3
println(asList) // [1, 3]

```

There are a few important advantages to the fact that sequences are lazy in Kotlin:

- They keep the natural order of operations.
- They do a minimal number of operations.
- They can be infinite.
- They do not need to create collections at every step.

Let's talk about these advantages one by one.

## Order is important

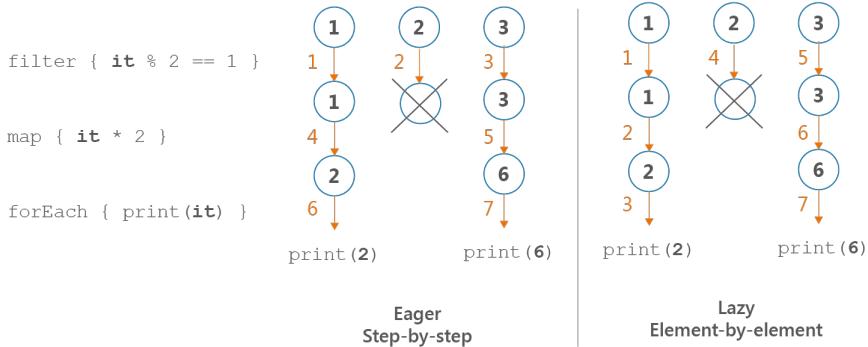
Because of how iterable and sequence processing is implemented, the ordering of their operations is different. In sequence processing, we take the first element and apply all the operations, then we take the next element, and so on. We call this an element-by-element or lazy order. In iterable processing, we take the first operation and apply it to the whole collection, then we move to the next operation, and so on. We call this a step-by-step or eager order.

```

listOf(1, 2, 3)
    .filter { print("F$it, "); it % 2 == 1 }
    .map { print("M$it, "); it * 2 }
    .foreach { print("E$it, ") }
// Prints: F1, F2, F3, M1, M3, E2, E6,

sequenceOf(1, 2, 3)
    .filter { print("F$it, "); it % 2 == 1 }
    .map { print("M$it, "); it * 2 }
    .foreach { print("E$it, ") }
// Prints: F1, M1, E2, F2, F3, M3, E6,

```



Notice that if we were to implement these operations without any collection processing functions and we used classic loops and conditions instead, we would have an element-by-element order, like in sequence processing:

```

for (e in listOf(1, 2, 3)) {
    print("F$e, ")
    if (e % 2 == 1) {
        print("M$e, ")
        val mapped = e * 2
        print("E$mapped, ")
    }
}
// Prints: F1, M1, E2, F2, F3, M3, E6,

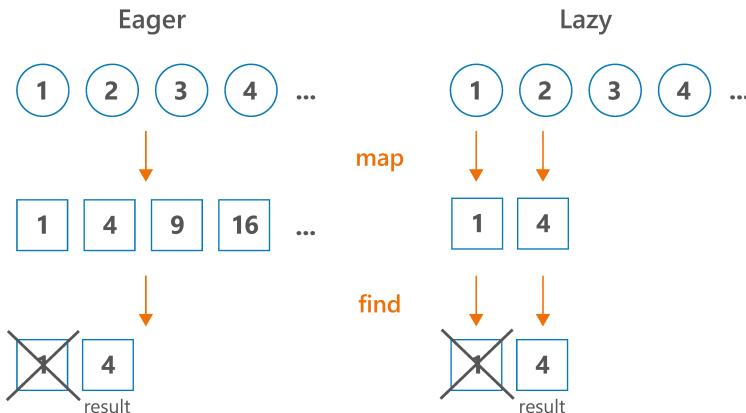
```

Therefore, the element-by-element order that is used in sequence processing is more natural. It also opens the door for low-level compiler optimizations as

sequence processing can be optimized to basic loops and conditions. Maybe this will happen one day.

## Sequences do the minimal number of operations

Often we do not need to process a whole collection at every step to produce the result. Let's say that we have a collection with millions of elements and, after processing, we only need to take the first 10. Why process all the other elements? Iterable processing doesn't have the concept of intermediate operations, so a processed collection is returned from every operation. Sequences do not need that, therefore they can do the minimal number of operations required to get the result.



Take a look at the following example, where we have a few processing steps and we end our processing with `find`:

```
(1..10)
    .filter { print("F$it, "); it % 2 == 1 }
    .map { print("M$it, "); it * 2 }
    .find { it > 5 }
// Prints: F1, F2, F3, F4, F5, F6, F7, F8, F9, F10,
// M1, M3, M5, M7, M9,

(1..10).asSequence()
    .filter { print("F$it, "); it % 2 == 1 }
```

```

    .map { print("M$it, "); it * 2 }
    .find { it > 5 }
// Prints: F1, M1, F2, F3, M3,

```

For this reason, when we have some intermediate processing steps and our terminal operation does not necessarily need to iterate over all elements, using a sequence will most likely be better for your processing performance and it looks nearly the same. Examples of operations that do not necessarily need to process all the elements are `first`, `find`, `take`, `any`, `all`, `none`, or `indexOf`.

## Sequences can be infinite

Thanks to the fact that sequences do processing on demand, we can have infinite sequences. A typical way to create an infinite sequence is using sequence generators like `generateSequence` or `sequence`.

`generateSequence` needs the first element and a function specifying how to calculate the next one:

```

generateSequence(1) { it + 1 }
    .map { it * 2 }
    .take(10)
    .foreach { print("$it, ") }
// Prints: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,

```

`sequence` uses a suspending function (coroutine<sup>73</sup>) that generates the next number on demand. Whenever we ask for the next number, the sequence builder runs until a value is yielded using `yield`. The execution then stops until we ask for another number. Here is an infinite list of Fibonacci numbers:

```

import java.math.BigDecimal

val fibonacci: Sequence<BigDecimal> = sequence {
    var current = 1.toBigDecimal()
    var prev = 1.toBigDecimal()
    yield(prev)
    while (true) {
        yield(current)
        val temp = prev

```

---

<sup>73</sup>These are sequential coroutines, as opposed to parallel/concurrent coroutines. They do not change thread, but they use the capability of suspended functions to stop in the middle of the function and resume whenever needed.

```
        prev = current
        current += temp
    }
}

fun main() {
    print(fibonacci.take(10).toList())
    // [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
}
```

Notice that infinite sequences cannot be processed unless we limit their number of elements. We cannot iterate infinitely.

```
print(fibonacci.toList()) // Runs forever
```

Therefore, we either need to limit them using an operation like `take`, or we need to use a terminal operation that will not need all elements, like `first`, `find` or `indexOf`. Basically, these are also the operations for which sequences are more efficient because they do not need to process all elements.

Notice, that `any`, `all`, and `none` should not be used without being limited first. `any` can only return `true` or run forever. Similarly, `all` and `none` can only return `false`.

## Sequences do not create collections at every processing step

Standard collection processing functions return a new collection at every step. Most often it is a `List`. This could be an advantage as we have something ready to be used or stored after every step, but this comes at a cost. Such collections need to be created and filled with data at every step.

```
numbers
    .filter { it % 10 == 0 } // 1 collection here
    .map { it * 2 } // 1 collection here
    .sum()
// In total, 2 collections created under the hood
```

```
numbers
    .asSequence()
    .filter { it % 10 == 0 }
    .map { it * 2 }
    .sum()
// No collections created
```

This is a problem, especially when we are dealing with big or heavy collections. Let's start from an extreme yet common case: file reading. Files can weigh gigabytes, therefore allocating all the data in a collection at every processing step could be a huge waste of memory. This is why we use sequences to process files by default.

As an example, let's analyze crimes in the city of Chicago. This city, like many others, shares the whole database of crimes that have taken place there since 2001 on the internet<sup>74</sup>. Currently, this dataset weighs over 1.53 GB. Let's say that our task is to find how many crimes had cannabis in their descriptions. This is what a naive solution using collection processing would look like (`readLines` returns `List<String>`):

```
// BAD SOLUTION, DO NOT USE COLLECTIONS FOR
// POSSIBLY BIG FILES
File("ChicagoCrimes.csv").readLines()
    .drop(1) // Drop descriptions of the columns
    .mapNotNull { it.split(",").getOrNull(6) }
    // Find description
    .filter { "CANNABIS" in it }
    .count()
    .let(::println)
```

The result on my computer is `OutOfMemoryError`.



Exception in thread “main” java.lang.OutOfMemoryError: Java heap space

No surprise! We create a collection, then we have 3 intermediate processing steps which add up to 4 collections. 3 of them contain the majority of this data file, which takes 1.53 GB, so in total, they consume more than 4.59 GB. This is a huge waste of memory. The correct implementation should involve a sequence, and we do this using the `useLines` function, which always operates on a single line:

---

<sup>74</sup>You can find this database at [data.cityofchicago.org](http://data.cityofchicago.org)

```
File("ChicagoCrimes.csv").useLines { lines ->
    // The type of `lines` is Sequence<String>
    lines.drop(1) // Drop descriptions of the columns
        .mapNotNull { it.split(",").getOrNull(6) }
        // Find description
        .filter { "CANNABIS" in it }
        .count()
        .let { println(it) } // 318185
}
```

On my computer, this took 8.3s. To compare the efficiency of both methods, I did another experiment: I reduced this dataset size by dropping the columns I didn't need. Thus, I achieved a `CrimeData.csv` file with the same crimes but with a size of only 728 MB. Then, I did the same processing. The first implementation, which uses collection processing, took around 13s; the second one, which uses sequences, took around 4.5s. As you can see, using sequences for big files is good not only for memory but also for performance.

A significant cost of collection processing is the creation of new collections. This cost is smaller when the size of the new collection is known in advance, like when we use `map`, or the cost is bigger when it is not, like when we use `filter`. Knowing the size of a new list is important because Kotlin generally uses `ArrayList` as the default list, which needs to copy its internal array when its number of elements is greater than internal capacity (whenever this copy operation is growing, internal capacity grows by half). Sequence processing only creates a collection when it ends with `toList`, `toSet`, etc. However, notice that `toList` on a sequence does not know the size of the new collection in advance.

Both creating collections and filling them with data represent a significant part of the cost of processing collections. This is why we should **prefer to use Sequences for big collections with more than one processing step**.

By “big collections” I mean both many elements and really heavy collections. It might be a list of integers with tens of thousands of elements. It might also be a list with just a few strings, each of which is so long that they all weigh many megabytes in total. These are not common situations, but they sometimes happen.

By one processing step, I mean more than a single function for collection processing. So, if you compare these two functions:

```
fun singleStepListProcessing(): List<Product> {
    return productsList.filter { it.bought }
}

fun singleStepSequenceProcessing(): List<Product> {
    return productsList.asSequence()
        .filter { it.bought }
        .toList()
}
```

You will notice that there is nearly no difference in performance (actually, simple list processing is faster because its `filter` function is inline). However, when you compare functions with more than one processing step, like the functions below which use `filter` and then `map`, the difference will be visible for bigger collections. To see the difference, let's compare typical processing with two and three processing steps for 5,000 products:

```
fun twoStepListProcessing(): List<Double> {
    return productsList
        .filter { it.bought }
        .map { it.price }
}

fun twoStepSequenceProcessing(): List<Double> {
    return productsList.asSequence()
        .filter { it.bought }
        .map { it.price }
        .toList()
}

fun threeStepListProcessing(): Double {
    return productsList
        .filter { it.bought }
        .map { it.price }
        .average()
}

fun threeStepSequenceProcessing(): Double {
    return productsList.asSequence()
        .filter { it.bought }
        .map { it.price }
```

```
    .average()  
}
```

Below, you can see the average results on a MacBook Pro (Retina, 15-inch, Late 2013)<sup>75</sup> for 5,000 products in the `productsList`:

|  |    |     |    |
|--|----|-----|----|
| twoStepListProcessing                  | 81 | 095 | ns |
| twoStepSequenceProcessing              | 55 | 685 | ns |
| twoStepListProcessingAndAccumulate     | 83 | 307 | ns |
| twoStepSequenceProcessingAndAccumulate | 6  | 928 | ns |

It is hard to predict what performance improvement we can expect. From my observations, in a typical collection processing example with more than one step, we can expect around a 20-40% performance improvement for at least a couple of thousand elements.

## When aren't sequences faster?

There are some operations where we don't profit from this sequence usage because we have to operate on the whole collection anyway. `sorted` is an example from Kotlin stdlib (currently, this is the only example). `sorted` uses an optimal implementation: it accumulates the Sequence into `List` and then uses `sort` from Java stdlib. The disadvantage is that this accumulation process takes longer than the same processing on a `Collection` (however, if `Iterable` is not a `Collection` or array, then the difference is not significant because it also needs to be accumulated).

The fact that `Sequence` has methods like `sorted` is controversial because sequences that contain a method that requires all elements to calculate the next one are only partially lazy (evaluated when we need to get the first element) and they don't work on infinite sequences. It was added because it is a popular function and is much easier to use in this way; however, Kotlin developers should remember its flaws, especially the fact that it cannot be used on infinite sequences.

```
generateSequence(0) { it + 1 }.take(10).sorted().toList()  
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
generateSequence(0) { it + 1 }.sorted().take(10).toList()  
// Infinite time. Does not return.
```

`sorted` is a rare example of a processing step that is faster on `Collection` than on `Sequence`. Still, when we do a few processing steps and a single `sorted` function (or another function that needs to work on the whole collection), we might expect some performance improvement using sequence processing.

---

<sup>75</sup>Processor 2.6 GHz Intel Core i7, Memory 16 GB 1600 MHz DDR3

```
productsList.asSequence()
    .filter { it.bought }
    .map { it.price }
    .sorted()
    .take(10)
    .sum()
```

## What about Java streams?

Java 8 introduced streams to allow collection processing. They act and look similar to Kotlin sequences.

```
productsList.asSequence()
    .filter { it.bought }
    .map { it.price }
    .average()

productsList.stream()
    .filter { it.bought }
    .mapToDouble { it.price }
    .average()
    .orElse(0.0)
```

Java 8 streams are lazy and are collected in the last (terminal) processing step. The three big differences between Java streams and Kotlin sequences are the following:

- Kotlin sequences have many more processing functions (because they are defined as extension functions), and they are generally easier to use (this is because Kotlin sequences were designed when Java streams were already in use; for instance, we can collect using `toList()` instead of `collect(Collectors.toList())`)
- Java stream processing can be started in parallel mode using a parallel function. This can give us a huge performance improvement in contexts in which we have a machine with multiple cores that are often unused (which is common nowadays). However, use this with caution as this feature has known pitfalls<sup>76</sup>.

---

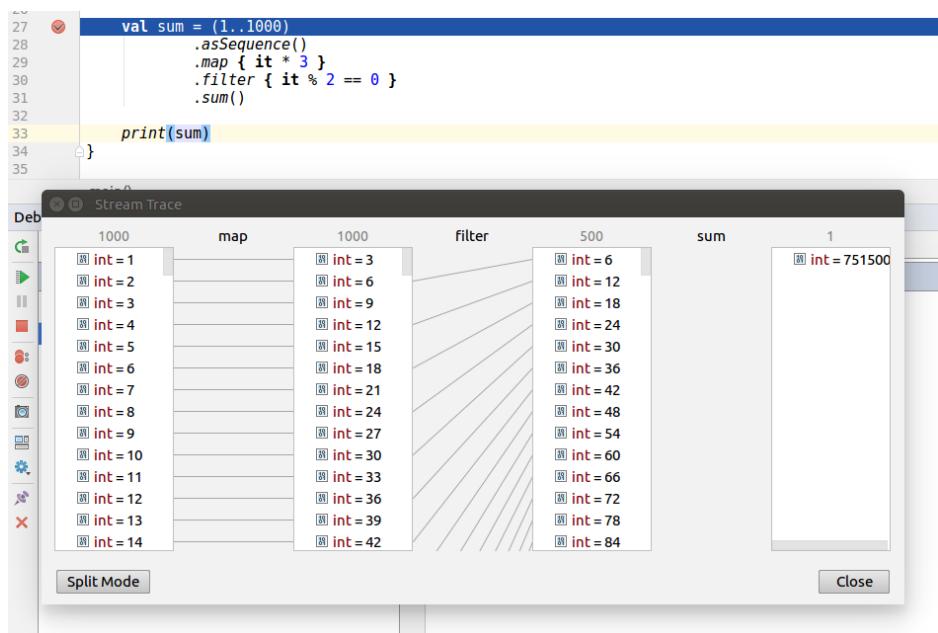
<sup>76</sup>The problems come from the common join-fork thread pool they use, which means that one process can block another. There's also a problem with the fact that single element processing can block other elements. Read more about it here: [kt.academy/l/java8-streams-problem](http://kt.academy/l/java8-streams-problem)

- Kotlin sequences can be used in common modules, Kotlin/JVM, Kotlin/JS, and Kotlin/Native modules. Java streams can be only used in Kotlin/JVM, and only when the JVM version is at least 8.

In general, when we don't use parallel mode, it is hard to say if Java stream or Kotlin sequence is more efficient. My suggestion is to use Java streams rarely: only for computationally heavy processing where you can profit from parallel mode. Otherwise, use Kotlin stdlib functions in order to achieve clean, homogeneous code that can be used on different platforms or in common modules.

## Kotlin Sequence debugging

Both Kotlin Sequence and Java Stream have the support that helps us debug how elements are transformed at every step. For Java Stream, this requires a plugin named “Java Stream Debugger”. Kotlin Sequences also used to require a plugin named “Kotlin Sequence Debugger”, though this functionality is now integrated into the Kotlin plugin. Here is a screen showing sequence processing at every step:



## Summary

Collection and sequence processing are very similar and both support nearly the same processing methods, yet there are important differences between the two.

Sequence processing is harder as we generally keep elements in collections, so changing to collections requires a transformation to a sequence and often also back to the desired collection. Sequences are lazy, which brings some important advantages:

- They keep the natural order of operations.
- They do a minimal number of operations.
- They can be infinite.
- They do not need to create collections at every step.

As a result, they are better for processing heavy objects or bigger collections with more than one processing step. Sequences also have their own debugger, which can help us by visualizing how elements are processed. Sequences are not intended to replace classic collection processing. You should use them only when there's a good reason to, and you'll be rewarded with better performance and fewer memory problems.

## Item 55: Consider associating elements to a map

It is not uncommon to have a big set of elements in which we need to find elements by their keys. This might be:

- A class that stores configurations that are loaded from one or more files.
- A network repository that stores downloaded data.
- An in-memory repository (such repositories are often used for tests).

This data might represent a list of users, ids, configurations, etc. It is generally fetched as a list, so it is tempting to represent it in memory in the same way:

```
class ConfigurationRepository(  
    private val configurations: List<Configuration>  
) {  
    fun getByName(name: String) = configurations  
        .firstOrNull { it.name == name }  
}  
  
class NetworkUserRepo(  
    private val userService: UserService  
) : UserRepo {  
  
    private var users: List<User>? = null  
  
    suspend fun loadUsers() {  
        users = userService.getUsers()  
    }  
  
    override fun getUser(id: UserId): User? = users  
        ?.firstOrNull { it.id == id }  
}  
  
class InMemoryUserRepo : UserRepo {  
  
    private val users: MutableList<User> = mutableListOf()  
  
    override fun getUser(id: UserId): User? = users  
        .firstOrNull { it.id == id }  
  
    fun addUser(user: User) {
```

```
        user.add(user)
    }
}
```

## Using Maps

However, this is rarely the best way to store these elements. Notice how the data we load is used: we most often access an element by an identifier or name (which is typically related to how we design our data so as to have unique keys in databases). Finding an element in a list has linear complexity ( $O(n)$ , where  $n$  is the size of the list; or, more concretely, it takes on average  $n/2$  comparisons to find an element in a list). This is especially problematic for big lists because finding each element requires comparing it with many other elements. A better solution is to use a Map instead of a List. Kotlin by default uses a hashmap (`LinkedHashMap`), which offers much better performance when finding an element, as is described in Item 43: Respect the contract of `hashCode`. On JVM, finding an element takes only one comparison because the size of the used hashmap is adjusted to the size of the map itself (given that the `hashCode` function is implemented properly).

For example, this is the `InMemoryRepo` from before, but its implementation uses a map instead of a list:

```
class InMemoryUserRepo : UserRepo {
    private val users: MutableMap<UserId, User> =
        mutableMapOf()

    override fun getUser(id: UserId): User? = users[id]

    fun addUser(user: User) {
        user.put(user.id, user)
    }
}
```

Most other operations, like modifying or iterating over data (likely using collection processing methods like `filter`, `map`, `flatMap`, `sorted`, `sum`, etc.), have more or less the same performance for the standard map and list, but finding an element by key is much faster.

## Associating elements with keys

The crucial question is how to transform from a list to a map and vice versa? To transform an iterable to a map, we use the `associate` method, in which we produce key-value pairs for each element.

```
data class User(val id: Int, val name: String)

val users = listOf(User(1, "Michał"), User(2, "Marek"))

val nameById: Map<Int, String> =
    users.associate { it.id to it.name }
    println(byId) // {1=Michał, 2=Marek}
```

There are other variants of the `associate` method. My favorite is `associateBy`, which keeps an iterable element as a value, and we only have to decide what key should represent it.

```
val byId: Map<Int, User> = users.associateBy { it.id }
println(byId)
// {1=User(id=1, name=Michał),
// 2=User(id=2, name=Marek)}

val byName: Map<String,User> = users.associateBy { it.name }
println(byName)
// {Michał=User(id=1, name=Michał),
// Marek=User(id=2, name=Marek)}
```

Note that keys in a map must be unique because duplicates are removed. This is why we should only associate elements using a unique identifier (to group by something that is not unique, use the `groupBy` function).

To transform from a map to a list, you can use the `values` property of `Map`:

```
fun main() {
    val users = listOf(User(1, "Michał"), User(2, "Michał"))
    val byId = users.associateBy { it.id }
    println(byId.values)
    // [User(id=1, name=Michał), User(id=2, name=Michał)]
}
```

By associating elements to a map, we can improve the performance of our code. This is especially important when we need to access elements frequently. For example, we can use this technique to improve the performance of the `ConfigurationRepository` and `NetworkUserRepo` from the beginning of this item:

```
class ConfigurationRepository(  
    configurations: List<Configuration>  
) {  
    private val configurations: Map<String, Configuration> =  
        configurations.associateBy { it.name }  
  
    fun getByName(name: String) = configurations[name]  
}  
  
class NetworkUserRepo(  
    private val userService: UserService  
) : UserRepo {  
  
    private var users: Map<UserId, User>? = null  
  
    suspend fun loadUsers() {  
        users = userService.getUsers()  
            .associateBy { it.id }  
    }  
  
    override fun getUser(id: UserId): User? = users?.get(id)  
}
```

Thanks to the changes we've made, finding a configuration or a user is an immediate operation that we can do as many times as we want without worrying about performance. Of course, the cost is that we need to associate our elements to a map, but this is an operation with linear complexity, just like finding an element in a list.

This technique can also be used for more complicated processing operations. For example, imagine that you need to fetch a list of articles from one service, and details of authors from another one. Then you need to associate each article with its author. This is how a naive implementation might look:

```
fun produceArticlesWithAuthors(  
    articles: List<Article>,  
    authors: List<Author>  
) : List<ArticleWithAuthor> {  
    return articles.map { article ->  
        val author = authors  
            .first { it.id == article.authorId }  
        ArticleWithAuthor(article, author)  
    }  
}
```

```
    }  
}
```

The complexity of this solution is  $O(n * m)$ , where  $n$  is the number of articles, and  $m$  is the number of authors. This is because we need to find each article's author, and finding an author is a linear operation. We can improve the performance of this code by associating authors to a map:

```
fun produceArticlesWithAuthors(  
    articles: List<Article>,  
    authors: List<Author>  
) : List<ArticleWithAuthor> {  
    val authorsById = authors.associateBy { it.id }  
    return articles.map { article ->  
        val author = authorsById[article.authorId]  
        ArticleWithAuthor(article, author)  
    }  
}
```

The complexity of this solution is  $O(n + m)$ , because we first associate the authors to a map, which needs one iteration over the authors list; then we iterate over the article list once, and finding the author of each is an extremely fast operation. This is a significant improvement if the number of articles and authors is not small.

## Summary

- Use `Map` instead of `List` when you need to find elements by key.
- Use `associate` or `associateBy` to transform from a list to a map.
- Associating elements to a map to find them by their keys is a powerful technique that can be used to improve the performance of your code.

## Item 56: Consider using `groupingBy` instead of `groupBy`

I've noticed that when people seek my help with collection processing, what they are often missing is the fact that elements can be grouped. Here are a few tasks that require this operation:

- Counting the number of users in a city, based on a list of users.
- Finding the number of points received by each team, based on a list of players.
- Finding the best option in each category, based on a list of options.

There are two ways to group elements from an iterable. The first one is easier, but the second one is faster. Let's discuss them both.

### groupBy

The easiest way to solve this problem is by using the `groupBy` function, which returns a `Map<K, List<V>>`, where `V` is the type of the elements in the collection we started from, and `K` is the type we are mapping to. So, if we have a `User` list that we group by an id of type `String`, then the returned map is `Map<String, List<User>>`. In other words, `groupBy` divides our collection into multiple small collections: one for each unique key. This is how this function can be used to solve the above problems:

```
// Count the number of users in each city
val usersCount: Map<City, Int> = users
    .groupBy { it.city }
    .mapValues { (_, users) -> users.size }

// Find the number of points received by each team
val pointsPerTeam: Map<Team, Int> = players
    .groupBy { it.team }
    .mapValues { (_, players) ->
        players.sumOf { it.points }
    }

// Find the best option in each category
val bestFormatPerQuality: Map<Quality, Resolution> =
    formats.groupBy { it.quality }
        .mapValues { (_, formats) ->
```

```

        formats.maxByOrNull { it.resolution }!!
        // it is fine to use !! here, because
        // this collection cannot be empty
    }
}

```

These are good solutions. When we use `groupBy`, we receive a `Map` as a result, and we can use all the different methods defined on it. This makes `groupBy` a really nice intermediate step.

## groupingBy

On the other hand, if we are dealing with some performance-critical parts of our code, `groupBy` is not the best choice because it takes some time to create a collection for each category we have, especially since these group sizes are not known in advance. Instead, we could use the `groupingBy` function, which does not do any additional operations: it just wraps the iterable together with the specified key selector.

```

public inline fun <T, K> Iterable<T>.groupingBy(
    crossinline keySelector: (T) -> K
): Grouping<T, K> {
    return object : Grouping<T, K> {
        override fun sourceIterator(): Iterator<T> =
            this@groupingBy.iterator()
        override fun keyOf(element: T): K =
            keySelector(element)
    }
}

```

The returned `Grouping` can be considered a bit like a map from a key to a list of elements, but it supports far fewer operations. However, since using it might be an important optimization, let's analyze the options.

The first problem (counting users per city) can be solved easily. The Kotlin Standard Library already has the `eachCount` function, which easily gives us a map from each city to its number of users.

```

val usersCount = users.groupingBy { it.city }
    .eachCount()

```

Finding the number of points received by each team is a bit harder. We can use the `fold` function, which is like a `fold` on an iterable, but it has a separate accumulator for each unique key. So, calculating the number of points per team is very similar to calculating the number of points in a collection.

```
val pointsPerTeam = players
    .groupingBy { it.team }
    .fold(0) { acc, elem -> acc + elem.points }
```

It would make sense to extract an extension function to calculate the sum of elements in each group. We might call it `eachSumBy`.

```
fun <T, K> Grouping<T, K>.eachSumBy(
    selector: (T) -> Int
): Map<K, Int> =
    fold(0) { acc, elem -> acc + selector(elem) }

val pointsPerTeam = players
    .groupingBy { it.team }
    .eachSumBy { it.points }
```

Finally, the last problem: we need to find the biggest element in the group. We might use `fold`, but this would require a “zero” value, which we don’t have. Instead, we can use `reduce`, which just starts from the first element. Its lambda has one additional parameter: the reference to the key of the group (we don’t use it in the example below, so there is `_` instead).

```
val bestFormatPerQuality = formats
    .groupingBy { it.quality }
    .reduce { _, acc, elem ->
        if (acc.resolution > elem.resolution) acc else elem
    }
```

Now, you might have noticed that we could also have used `reduce` in the previous problem. If so, you’re right and such a solution would be more efficient. I just wanted to present both options.

Again, we can extract an extension function.

```
// Could be optimized to keep accumulator selector
inline fun <T, K> Grouping<T, K>.eachMaxBy(
    selector: (T) -> Int
): Map<K, T> =
    reduce { _, acc, elem ->
        if (selector(acc) > selector(elem)) acc else elem
    }

val bestFormatPerQuality = formats
    .groupingBy { it.quality }
    .eachMaxBy { it.resolution }
```

The last important function from the stdlib that is defined on `Grouping` is `aggregate`, which is very similar to `fold` and `reduce`. It iterates over all the elements and aggregates for each key. Its operation has 4 parameters: the key of the current element; an accumulator (also per element) or `null` for the first element with this key; a reference to the element; and a boolean, which is true if this element is the first element for this key. This is how our last problem can be solved using `aggregate`:

```
val bestFormatPerQuality = formats
    .groupingBy { it.quality }
    .aggregate { _, acc: VideoFormat?, elem: VideoFormat, _ ->
        when {
            acc == null -> elem
            acc.resolution > elem.resolution -> acc
            else -> elem
        }
    }
```

## Summary

The `groupBy` function is part of many collection processing operations. It is convenient to use as it returns a `Map` that has plenty of useful functions. Its alternative is `groupingBy`, which is better for performance but is generally harder to use. It currently supports the following functions: `eachCount`, `fold`, `reduce`, and `aggregate`. Using them, we can define other functions we might need, just as we defined `eachSumBy` and `eachMaxBy` in this chapter.

## Item 57: Limit the number of operations

Every collection processing method is a cost. For standard collection processing, the cost is:

- additional iteration over the elements,
- a new collection created under the hood.

For sequence processing, the cost is:

- another object that wraps the whole sequence,
- lambda expression creation<sup>77</sup>.

All these costs are generally small, but they are proportional to the number of elements. So, they can become significant with big collections. One of the most basic ways of limiting this cost is using operations that are composites. For instance, instead of filtering for not null and then casting to non-nullable types, we use `filterNotNull`. Or, instead of mapping and then filtering out nulls, we can just use `mapNotNull`.

```
class Student(val name: String?)\n\n// Works\nfun List<Student>.getNames(): List<String> = this\n    .map { it.name }\n    .filter { it != null }\n    .map { it!! }\n\n// Better\nfun List<Student>.getNames(): List<String> = this\n    .map { it.name }\n    .filterNotNull()\n\n// Best\nfun List<Student>.getNames(): List<String> = this\n    .mapNotNull { it.name }
```

---

<sup>77</sup>We need to create a lambda expression as an object because the operation is passed to the sequence object, so it cannot be inlined.

The biggest problem is not misunderstanding the importance of such changes but a lack of knowledge about which collection processing functions we should use. This is another reason why it is good to learn them. Also, help comes from IDE warnings, which often suggest a better alternative.

```

19     fun makePassingStudentsListText(): String = studentsRepository
20         .getStudents()
21             .filter { it.pointsInSemester > 15 && it.result >= 50 }
22                 .sortedWith(compareBy{ it.surname }, { it.name })
23                     .map { "${it.name} ${it.surname}, ${it.result}" }
24                         .joinToString(separator = "\n")
25
26
27

```

Call chain on collection type may be simplified [more...](#) (#F1)

However, these suggestions are still limited. Here is a list of a few common function calls and alternative ways of limiting the number of operations:

| Instead of:   | Use:   |
|---|--|
| <code>.filter { it != null }</code><br><code>.map { it!! }</code>   | <code>.filterNotNull()</code>  |
| <code>.map { &lt;Transformation&gt; }</code><br><code>.filterNotNull()</code>   | <code>.mapNotNull { &lt;Transformation&gt; }</code>  |
| <code>.map { &lt;Transformation&gt; }</code><br><code>.joinToString()</code>  | <code>.joinToString { &lt;Transformation&gt; }</code>  |
| <code>.filter { &lt;Predicate 1&gt; }</code><br><code>.filter { &lt;Predicate 2&gt; }</code>  | <code>.filter { &lt;Predicate 1&gt; &amp;&amp; &lt;Predicate 2&gt; }</code>  |
| <code>.filter { it is Type }</code><br><code>.map { it as Type }</code>   | <code>.filterIsInstance&lt;Type&gt;()</code>   |
| <code>.sortedBy { &lt;Key 2&gt; }</code><br><code>.sortedBy { &lt;Key 1&gt; }</code>  | <code>.sortedWith(</code><br><code>compareBy{ &lt;Key 1&gt; }, { &lt;Key 2&gt; })</code>   |
| <code>listOf(...)</code><br><code>.filterNotNull()</code>   | <code>listOfNotNull(...)</code>  |
| <code>.withIndex()</code><br><code>.filter { (index, elem) -&gt;</code><br><code>&lt;Predicate using index&gt; }</code><br><code>.map { it.value }</code> | <code>.filterIndexed { index, elem -&gt;</code><br><code>&lt;Predicate using index&gt; }</code><br><br>(Similarly for map, foreach, reduce and fold) |

## Summary

Most collection processing steps require iteration over the whole collection and intermediate collection creation. This cost can be limited by using more suitable collection processing functions.

## Item 58: Consider Arrays with primitives for performance-critical processing

We cannot declare primitives in Kotlin, but they are used as an optimization under the hood. This is a significant optimization, as described already in Item 47: Avoid unnecessary object creation. Primitives are:

- Lighter, as every object adds additional weight.
- Faster, as accessing a value through accessors is an additional cost, and creating an object is also a cost.

Therefore, using primitives for a huge amount of data might be a significant optimization. One problem is that typical Kotlin collections like `List` or `Set` are generic. Primitives cannot be used as generic types, so we end up using wrapped types instead. This is a convenient solution that suits most cases as it is easier to do processing over standard collections. Having said that, in performance-critical parts of our code we should instead consider using arrays with primitives, like `IntArray` or `LongArray`, as they are lighter in terms of memory and their processing is more efficient.

| Kotlin type                   | Java type                        |
|-------------------------------|----------------------------------|
| <code>Int</code>              | <code>int</code>                 |
| <code>List&lt;Int&gt;</code>  | <code>List&lt;Integer&gt;</code> |
| <code>Array&lt;Int&gt;</code> | <code>Integer[]</code>           |
| <code>IntArray</code>         | <code>int[]</code>               |

How much lighter are arrays with primitives? Let's say that in Kotlin/JVM we need to hold 1,000,000 integers, and we can choose to keep them either in `IntArray` or in `List<Int>`. If you make some measurements, you will find that on a typical machine `IntArray` allocates 4,000,016 bytes, while `List<Int>` allocates 20,000,040 bytes, which is 5 times more. If it is possible to optimize for memory use, choose arrays with primitives.

```
import jdk.nashorn.internal.ir.debug.ObjectSizeCalculator
.getObjectSize

fun main() {
    val ints = List(1_000_000) { it }
    val array: Array<Int> = ints.toTypedArray()
    val intArray: IntArray = ints.toIntArray()
    println(getObjectSize(ints))      // 20 000 040
```

```
    println(getObjectSize(array))      // 20 000 016
    println(getObjectSize(intArray))   // 4 000 016
}
```

There is also a difference in performance. For the same collection of 1,000,000 numbers, calculating the average of these elements is around 25% faster when we use an array with primitives instead of a list with wrapped integers.

```
open class InlineFilterBenchmark {

    lateinit var list: List<Int>
    lateinit var array: IntArray

    @Setup
    fun init() {
        list = List(1_000_000) { it }
        array = IntArray(1_000_000) { it }
    }

    @Benchmark
    // On average 1 260 593 ns
    fun averageOnIntList(): Double {
        return list.average()
    }

    @Benchmark
    // On average 868 509 ns
    fun averageOnIntArray(): Double {
        return array.average()
    }
}
```

As you can see, primitives and arrays with primitives can be used as an optimization in performance-critical parts of your code. They allocate less memory, and their processing is faster. However, the improvement in most cases is not significant enough to use arrays with primitives by default instead of lists. Lists are more intuitive and we use them much more often, so in most cases, we should prefer them instead. Just keep this optimization in mind in case you need to optimize some performance-critical parts of code.

## Summary

In a typical case, `List` or `Set` should be preferred over arrays. However, if you hold big collections of values that can be represented as primitives, using arrays of primitives might significantly improve your performance and memory use. This is especially important for library creators or developers who write games or do advanced graphic processing.

## Item 59: Consider using mutable collections

The biggest advantage of using mutable collections instead of immutable collections is that their performance is faster. When we add an element to an immutable collection, we need to create a new collection and add all elements to it. Here is how this is currently implemented in Kotlin stdlib (Kotlin 1.2):

```
operator fun <T> Iterable<T>.plus(element: T): List<T> {
    if (this is Collection) return this.plus(element)
    val result = ArrayList<T>()
    result.addAll(this)
    result.add(element)
    return result
}
```

When we deal with bigger collections, adding multiple elements to another collection can be a costly process. This is why using mutable collections, especially if we often need to add elements, is a performance optimization. On the other hand, Item 1: Limit mutability taught us the advantages of using immutable collections for safety. Notice that these arguments rarely apply to local variables, where synchronization or encapsulation is rarely needed. This is why it generally makes more sense to use mutable collections for local processing. This is reflected in the standard library, where all collection processing functions are internally implemented using mutable collections:

```
inline fun <T, R> Iterable<T>.map(
    transform: (T) -> R
): List<R> {
    val size =
        if (this is Collection<*>) this.size else 10
    val destination = ArrayList<R>(size)
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

Instead of using immutable collections:

```
// This is not how map is implemented
inline fun <T, R> Iterable<T>.map(
    transform: (T) -> R
): List<R> {
    var destination = listOf<R>()
    for (item in this)
        destination += transform(item)
    return destination
}
```

## Summary

Adding mutable collections to elements is generally faster, but immutable collections give us more control over how they are changed. However, in the local scope we generally do not need this control, so mutable collections should be preferred, especially in util functions, where element insertion might happen many times.

## Item 60: Use appropriate collection types

Lists, sets, and maps are represented by interfaces. Each of these collection types has a specific contract:

- List represents an ordered collection of elements. The same elements can occur multiple times. A list's elements can be accessed by indices (zero-based integer numbers that reflect elements' positions).
- Set represents a collection of unique elements.
- Map represents a set of key-value pairs; each key must be unique and points to exactly one value.

These contracts still leave plenty of freedom for different implementations of these collections. The biggest differences between different implementations are:

- The data structure that is used under the hood, which determines the efficiency of different operations.
- Whether the collection is mutable or immutable; if it is mutable, whether it is thread-safe or not.

In Kotlin, most collections we use are mutable under the hood, therefore they're not thread-safe. This is the kind of collection we'll concentrate on in this item. We discussed thread-safe collections in Item 2: Eliminate critical sections.

Regarding the data structure used under the hood, this is something we might want to consider when we optimize the performance of our application. So, let's consider the most important types of collections and their performance characteristics.

### Array-based list

The default implementation of `List` in Kotlin is the array-based list. It is just a wrapper over an array, which is like a certain number of reserved places next to each other in memory. An array-based list keeps track of the number of elements and their inner array size. When the number of elements reaches the size of the inner array, the array is replaced with a new one which is bigger. The size of the new array is usually calculated as the size of the old array multiplied by a certain factor, which in JVM `ArrayList` is 1.5. This is how `ArrayList` grows. When the number of elements decreases, the size of the inner array is typically not decreased, but element references are replaced with `null` values. This is how `ArrayList` shrinks.

By knowing how `ArrayList` works, we can predict the time, we can predict the time complexity of different operations on it:

- Accessing an element by its index is very fast ( $O(1)$ ) because it's just a matter of accessing the corresponding element in the inner array.
- If there is still a place left in the internal array, adding an element to the end of the list is also very fast ( $O(1)$ ) because it is just a matter of adding an element to the inner array. If there is not enough space, the internal array needs to be copied; this is more demanding for the CPU, but it is still a linear operation ( $O(n)$ ). This is why it's a good idea to initialize `ArrayList` with a size that is the expected number of elements.
- Adding an element to the beginning or in the middle of the list is a linear operation ( $O(n)$ ) because all elements need to be shifted to the right.
- Changing an element by its index is a constant time operation ( $O(1)$ ) because it's just a matter of changing the corresponding element in the inner array.
- Removing an element from the end of the list is very fast ( $O(1)$ ) because it's just a matter of removing the last element from the inner array.
- Removing an element from the beginning or the middle of the list is a linear operation ( $O(n)$ ) because all elements need to be shifted to the left.
- Searching for an element by value is a linear operation ( $O(n)$ ) because all elements need to be checked one by one.

## Deque

Deque (usually pronounced “deck”) stands for “double ended queue” as it represents a queue in which we can effectively add or remove elements at both ends. Kotlin offers an `ArrayDeque` implementation of this data structure which implements the `MutableList<E>` interface. Its behavior is similar to `ArrayList`, but its internal array is circular. This means that the beginning of the array is not necessarily the first element of this collection. There is a separate property (`head`) that marks the beginning of the collection. This way, if there is enough space in the internal array, adding or removing elements at the beginning or end is a constant time operation ( $O(1)$ ) because it is just a matter of inserting an element into the array and changing the `head` property; otherwise, if the array is full, we need to additionally shift elements, which has linear complexity.

When you need to implement a stack or a first-in-last-out (FILO), `ArrayList` is a good choice. When you need to implement a queue, i.e., a first-in-first-out (FIFO), `ArrayDeque` is a good choice.

Deque is a good choice when we often need to add or remove elements at the beginning or end of a list, like when we implement a queue or stack.

```

class InvoiceService {
    private val invoices = ArrayDeque<Invoice>()

    fun addInvoice(invoice: Invoice) = synchronized(this) {
        invoices.addLast(invoice)
    }

    fun processInvoices() = synchronized(this) {
        while (invoices.isNotEmpty()) {
            val invoice = invoices.removeFirst()
            // ...
        }
    }
}

```

To implement a first-in-first-out queue, we most often use `Channel` from the Kotlin Coroutines library, which is synchronized and thread-safe.

## Linked lists

The most well-known alternative to an array-based list is a linked list. This is a data structure that consists of nodes, where each node contains a value and a reference to the next node. The last node in the list has a reference to `null`. The list itself has a reference to the first node. This is a simplified implementation of a linked list:

```

class LinkedList<T> : List<T> {
    private val head: Node<T>? = null

    private class Node<T>(
        val value: T,
        var next: Node<T>?
    )

    // ...
}

```

The biggest advantage of this data structure is that adding or removing elements from the beginning or the middle of the list is a constant time operation ( $O(1)$ ) because it is just a matter of changing references. Adding elements is always fast

and does not require the initial capacity to be set. The biggest disadvantage is that accessing an element by its index is a linear operation ( $O(n)$ ) because we need to traverse the list from the beginning to the element with the given index. Adding or removing an element from the middle does not necessitate shifting the elements, but it does necessitate iterating over elements anyway to find the position by index, so it's a linear operation ( $O(n)$ ). Linked lists also take twice as much space because they need to store not only the elements but also all the references to the next elements. Finding an element, changing an element, or iterating over a list has similar time complexity as in an array-based list ( $O(n)$ ).

Kotlin does not provide a `LinkedList` implementation. On JVM, you can use `LinkedList` from Java stdlib, which we might choose to use when we often need to add or remove elements from the middle. Generally, linked lists are rarely used because, in most cases, an array-based list or deque is a better choice. However, the linking algorithm is used in many other data structures, like the default set or map, both of which use a hash table and links between elements.

## Hash tables

We've already discussed hash tables in [Item 43: Respect the contract of `hashCode`](#). This is a very popular algorithm that is used when we need an efficient way to find a value in a collection. Since sets and maps need to keep their elements unique, the default implementations of these collections use hash tables.

The basic form of a hash table loses the order of elements. To keep the order, the default implementations of `Set` and `Map` in Kotlin use a hash table in combination with links between elements. The actual classes are named `LinkedHashMap` and `LinkedHashSet`. So, each element in a hash table is a node of a linked list. This way, even though the `Set` and `Map` contracts do not guarantee the order of elements, Koltins' default `Set` and `Map` guarantee to keep the order in which elements were added.

If you don't need to keep elements in order in a set or map, you can use an implementation that does not keep links between elements. The advantages of this are that it takes less memory and is faster, as maintaining these connections takes time. You can create such collections using the `HashSet` or `HashMap` classes, or the `hashSetOf` or `hashMapOf` functions.

```
fun deserialize(input: ByteReadPacket): LeadersMessage {
    val size = input.readInt()
    val leaders = HashMap<City, User>(size)
    repeat(size) {
        val city = deserializeCity(input)
        val user = deserializeUser(input)
        leaders[city] = user
    }
    return LeadersMessage(leaders)
}
```

These implementations of set and map keep the internal hash table size greater than the number of elements. Therefore, finding an element by key is a constant time operation ( $O(1)$ ). Adding an element is also a constant time operation ( $O(1)$ ) unless the hash table needs to be resized, which is a linear operation ( $O(n)$ ). Removing an element is also a constant time operation ( $O(1)$ ). Iterating over a set or map is a linear operation ( $O(n)$ ) because we need to traverse all the elements in the hash table.

## Sorted binary trees

In computer science, there are many different kinds of tree-based data structures, but one of these is especially important from Kotlin's perspective: the sorted binary tree. This is a tree where each node has a maximum of two children, and each node has a value that is greater than all the values in its left subtree and smaller than all the values in its right subtree. This is a simplified implementation of a sorted binary tree:

```
class BinaryTree<T : Comparable<T>> {
    private val root: Node<T>? = null

    private class Node<T>(
        val value: T,
        var left: Node<T>?,
        var right: Node<T>?
    )

    // ...
}
```

In Kotlin, we can create a sorted set or a sorted map by using the `sortedSetOf` and `sortedMapOf` functions. These are useful when we need a collection that keeps elements in a specific order.

```
val sortedSet = sortedSetOf(5, 1, 3, 2, 4)
println(sortedSet) // [1, 2, 3, 4, 5]

val sortedMap = sortedMapOf(
    5 to "five",
    1 to "one",
    3 to "three",
    2 to "two",
    4 to "four"
)
println(sortedMap) // {1=one, 2=two, 3=three, 4=four, 5=five}
```

Finding an element in such a collection is a logarithmic operation ( $O(\log n)$ ), which is better than for a regular array-based list but worse than for a hash table. Adding and removing elements is a logarithmic operation ( $O(\log n)$ ) because we must traverse the tree to find the right place for the new element. Iterating over a tree is a linear operation ( $O(n)$ ) because we need to traverse the tree in order.

```
class ArticlesListAdapter {
    private val articles = sortedSetOf(ARTICLES_COMPARATOR)

    fun add(article: Article) {
        articles.add(article)
        redrawView()
    }

    private fun redrawView() {
        // ...
    }

    companion object {
        val ARTICLES_COMPARATOR: Comparator<Article> =
            compareByDescending { it.publishedDate }
    }
}
```

## Summary

In this item, we've discussed the most popular collection types and data structures used in Kotlin, and we've learned that each has its own advantages and disadvantages. We've also learned that the time complexity of different operations on a collection is very important, especially when these collections contain a lot

of elements. In such cases, we should always consider what the best collection for this use case will be.

# Dictionary

Some technical terms are not well-understood and require explanation. This is especially problematic when a term is confused with another similar one. This is why, in this chapter, I present some important terms used in this book, together with the terms that they are often confused with.

## Function vs method

In Kotlin, basic functions start with the `fun` keyword, and they can be defined:

- At the top-level (top-level functions)
- In a class (member functions)
- In a function (local functions)

Here are some examples:

```
fun double(i: Int) = i * 2 // Top-level function

class A {
    fun triple(i: Int) = i * 3 // Member function

    fun twelveTimes(i: Int): Int { // Member function
        fun fourTimes() = // Local function
            double(double(i))
        return triple(fourTimes())
    }
}
```

Additionally, we can define anonymous functions using function literals:

```
val double = fun(i: Int) = i * 2 // Anonymous function
val triple = { i: Int -> i * 3 } // Lambda expression,
// which is a shorter notation for an anonymous function
```

A **method** is a function associated with a class. Member functions (functions defined in a class) are clearly methods as they are associated with the class in which they are defined. An instance of this class is required if we are to call it, and we need to use this class name to reference it. In the example below, `doubled` is a member and a method. It is also a function because every method is a function.

```
class IntWrapper(val i: Int) {  
    fun doubled(): IntWrapper = IntWrapper(i * 2)  
}  
  
// Usage  
val wrapper = IntWrapper(10)  
val doubledWrapper = wrapper.doubled()  
  
val doubledReference = IntWrapper::doubled
```

Extension functions are also methods, because those are functions associated with a type. In the example below, you can see tripled, which is an extension function and also a method.

```
fun IntWrapper.tripled() = IntWrapper(i * 3)  
  
// Use  
val wrapper = IntWrapper(10)  
val tripledWrapper = wrapper.tripled()  
  
val tripledReference = IntWrapper::tripled
```

## Extension vs member

A member is an element defined in a class. In the following example, 4 members are declared: the name, surname, and fullName properties, and the withSurname method.

```
class User(  
    val name: String,  
    val surname: String  
) {  
    val fullName: String  
        get() = "$name $surname"  
  
    fun withSurname(surname: String) =  
        User(this.name, surname)  
}
```

Extensions are elements defined outside a class, but they are called in the same way as members. In the example below, there are 2 extensions: the officialFullName extension property, and the withName extension function.

```
val User.officialFullName: String
    get() = "$surname, $name"

fun User.withName(name: String) =
    User(name, this.surname)
```

## Parameter vs argument

A **parameter** is a variable defined in a function declaration. The **argument** is the actual value of this variable, which gets passed to the function. Take a look at the next example, where `length` in the `randomStr` declaration is a parameter, and `10` in this function call is an argument.

```
fun randomStr(length: Int): String { // length is a parameter
    // ....
}

randomString(10) // 10 is an argument
randomString(10 + 20) // 30 is an argument
```

It is similar to generic types. The type parameter is a blueprint or placeholder for a type declared as generic. The type argument is an actual type used to parametrize the generic. Take a look at the next example. `T` in the `printName` declaration is a type parameter, and `String` in this function call is a type argument.

```
inline fun <reified T> printName() { // T is a type parameter
    print(T::class.simpleName)
}

fun main() {
    printName<String>() // String is a type argument
}
```

## Primary vs Secondary constructor

A constructor is a special type of function<sup>78</sup> that is called to create an object. In Kotlin, we treat constructors like a function that produces an object. A constructor can be declared in a class body:

---

<sup>78</sup>Formally a subroutine, but constructors are treated just like functions in Kotlin, and a constructor reference implements a function type.

```
class SomeObject {  
    val text: String  
  
    constructor(text: String) {  
        this.text = text  
        print("Creating object")  
    }  
}
```

However, as it is common to use a constructor to set up an object, there is a concept called **primary constructor**: this is a constructor that is defined just after the class name, and it has parameters that can be used to initialize properties:

```
class SomeObject(text: String) {  
    val text: String = text  
  
    init {  
        print("Creating object")  
    }  
}
```

As it is common to have a primary constructor parameter named the same as the property it initializes, it is possible to shorten the above code with **properties defined in the primary constructor**:

```
class SomeObject(val text: String) {  
    init {  
        print("Creating object")  
    }  
}
```

When we need to create another constructor, we need to use a so-called **secondary constructor**: a constructor that calls the primary constructor using the `this` keyword:

```
class SomeObject(val text: String) {  
    constructor(date: Date) : this(date.toString())  
  
    init {  
        print("Creating object")  
    }  
}
```

However, this situation is rare in Kotlin because we can use default arguments when we need to support a different subset of primary constructor arguments (described in detail in Item 33: Consider a primary constructor with named optional arguments). Or, we can use a factory method when we need a different kind of object creation (described in detail in Item 32: Consider factory functions instead of secondary constructors). If we need to support multiple subsets of primary constructor arguments for Java use, we can use the `@JvmOverloads` annotation, which is an instruction for the compiler to generate overloads for this function that substitute default parameter values.

```
class SomeObject @JvmOverloads constructor(  
    val text: String = "")  
) {  
    init {  
        print("Creating object")  
    }  
}
```