The background features a large, abstract graphic composed of numerous overlapping blue triangles of varying sizes and shades, creating a sense of depth and motion.

Marcin Moskala

Effective Kotlin

BEST PRACTICES



Effective Kotlin

Best practices

Marcin Moskala

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

This version was published on 2020-06-20



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 - 2020 Marcin Moskala

Tweet This Book!

Please help Marcin Moskala 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](#)

Contents

Introduction: Be pragmatic	1
Part 1: Good code	13
Chapter 1: Safety	14
Item 1: Limit mutability	15
Item 2: Minimize the scope of variables	35
Item 3: Eliminate platform types as soon as possible	42
Item 4: Do not expose inferred types	50
Item 5: Specify your expectations on arguments and state	53
Item 6: Prefer standard errors to custom ones	64
Item 7: Prefer <code>null</code> or <code>Failure</code> result when the lack of result is possible	66
Item 8: Handle nulls properly	70
Item 9: Close resources with <code>use</code>	81
Item 10: Write unit tests	84
Chapter 2: Readability	88
Item 11: Design for readability	90
Item 12: Operator meaning should be consistent with its function name	96
Item 13: Avoid returning or operating on <code>Unit?</code>	101
Item 14: Specify the variable type when it is not clear	103
Item 15: Consider referencing receivers explicitly	105
Item 16: Properties should represent state, not behavior	113
Item 17: Consider naming arguments	119

CONTENTS

Item 18: Respect coding conventions	126
Part 2: Code design	129
Chapter 3: Reusability	130
Item 19: Do not repeat knowledge	132
Item 20: Do not repeat common algorithms	141
Item 21: Use property delegation to extract common property patterns	146
Item 22: Use generics when implementing common al- gorithms	153
Item 23: Avoid shadowing type parameters	157
Item 24: Consider variance for generic types	159
Item 25: Reuse between different platforms by extracting common modules	172
Chapter 4: Abstraction design	177
Item 26: Each function should be written in terms of a single level of abstraction	182
Item 27: Use abstraction to protect code against changes	189
Item 28: Specify API stability	205
Item 29: Consider wrapping external API	209
Item 30: Minimize elements visibility	211
Item 31: Define contract with documentation	216
Item 32: Respect abstraction contracts	230
Chapter 5: Object creation	233
Item 33: Consider factory functions instead of constructors	234
Item 34: Consider a primary constructor with named optional arguments	250
Item 35: Consider defining a DSL for complex object creation	261
Chapter 6: Class design	273
Item 36: Prefer composition over inheritance	274

CONTENTS

Item 37: Use the data modifier to represent a bundle of data	288
Item 38: Use function types instead of interfaces to pass operations and actions	296
Item 39: Prefer class hierarchies to tagged classes	301
Item 40: Respect the contract of <code>equals</code>	308
Item 41: Respect the contract of <code>hashCode</code>	322
Item 42: Respect the contract of <code>compareTo</code>	332
Item 43: Consider extracting non-essential parts of your API into extensions	336
Item 44: Avoid member extensions	341
Part 3: Efficiency	345
Chapter 7: Make it cheap	346
Item 45: Avoid unnecessary object creation	348
Item 46: Use inline modifier for functions with parameters of functional types	363
Item 47: Consider using inline classes	379
Item 48: Eliminate obsolete object references	388
Chapter 8: Efficient collection processing	397
Item 49: Prefer Sequence for big collections with more than one processing step	401
Item 50: Limit the number of operations	419
Item 51: Consider Arrays with primitives for performance-critical processing	422
Item 52: Consider using mutable collections	425
Dictionary	427

Introduction: Be pragmatic

Stories of how popular programming languages originate are often fascinating.

The prototype of JavaScript (named Mocha back then) was created in just 10 days. Its creators first considered using Java, but they wanted to make a simpler language for Web designers¹.

Scala was created at a university by a scientist, Martin Odersky. He wanted to move some concepts from functional programming into the Java Object Oriented Programming world. It turned out that these worlds are connectable².

Java was originally designed in the early ‘90s for interactive television and set-top boxes by a team called “The Green Team” working at Sun. Eventually, this language was too advanced for the digital cable television industry at the time. It ended up revolutionising general programming instead³.

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

1. Since the beginning, it was created and designed to be a general-purpose programming language, suitable for large applications.

¹Read about it here: www.en.wikipedia.org/wiki/JavaScript and www.2ality.com/2011/03/javascript-how-it-all-began.html

²Read about it here: https://www.artima.com/scalazine/articles/origins_of_scala.html

³Read about it here: <https://bit.ly/36LMw2z>

2. The creators of Kotlin are taking their time. Development of Kotlin started in 2010, and the first officially stable version was released in February of 2016. During this period, a lot has 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 never had ambitions to experiment with new concepts. Kotlin introduced a few new concepts (like property delegation) but the Kotlin team is very conscientious and they prefer to analyze how existing concepts have cooperated and worked for other languages. They are always trying to understand the strengths and weaknesses of other languages and build on them. It is a good strategy for JetBrains since it is the biggest creator of integrated programming environments (IDE). They have a lot of data and knowledge about how various languages are used. They also have specialists that understand each of those 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 and more advanced smart casting and type inference without the necessity of IDE adjustments. The Kotlin team also supports developers by constantly improving IntelliJ warnings 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 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 - with application growth, 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 - the application runs fast and needs fewer resources (memory, processor, etc.).

We, as a programming community, have tried to satisfy those 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*⁴ helps us with the same problem. Throughout this book, we will mention many such rules.

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

- Safety
- 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:

⁴SOLID is a popular set of principles for OOP, introduced and popularized by Robert C. Martin.

- Efficiency

Those requirements were not something present only at the beginning. They have been with Kotlin until 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 on that.

The purpose of this book

To really unleash the advantages of Kotlin, we need to use it properly. Knowing 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 get better at writing 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 find influence from presentations and Kotlin forums. This book tries to compose as much knowledge about best practices in Kotlin as possible, no matter where it originated.

You can call it a collection of best practices; though it differs from classic “Effective X” books because of Kotlin 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 future⁵. 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

⁵KotlinConf 2018 keynote by Andrey Breslav.

give hints about using 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 left only a few such items. 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.

For whom is this book written?

This book is not teaching basics. It assumes that you have enough knowledge and skills to do Kotlin development. If you don't, I recommend starting first with some resources designed for beginners. I recommend *Kotlin in Action* by Dmitry Jemerov and Svetlana Isakova or the Coursera course *Kotlin for Java Developers* by Andrey Breslav and Svetlana Isakova. *Effective Kotlin* is for experienced Kotlin developers.

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

- Property
- Platform type
- Named arguments
- Property delegation
- DSL creation
- Inline classes and functions

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

Book design

Concepts in the book are grouped into three parts:

- Good code - more general rules about making 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 generally is of the highest priority. Readability is another chapter because the code is not only for a compiler but also for programmers. Even when we work alone, we want code that is readable and self-explanatory.
- Code design - this section is for developers creating a project together with other developers, or creating libraries. It is about conventions and setting contracts. It will, in the end, reflect on readability and safety, but all in terms of correct code design. This part is a bit more abstract at the beginning, but thanks to that it can explore topics that are often omitted in books about code quality. This section is also about preparing our code for growth. A lot of items are about being ready for changes in the future. Therefore especially important for developers creating large projects.
- Efficiency - this section is for developers that 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.

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: Design abstractions
- 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, which are like rules. 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. For instance, the first rule, “Limit mutability”, might be enigmatic for someone who meets it for the first time, but is clear enough to just write in a comment on a code review for someone who is familiar with this book. In the end, suggestions designed this way, with their explanations, should give a clear guideline on how to write good and idiomatic Kotlin code.

Chapters 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 11: Design for readability*, but it is also true for:

- *Chapter 7: Make it cheap*'s first item *Item 45: Avoid unnecessary object creation*
- *Chapter 3: Reusability*'s first item *Item 19: Do not repeat knowledge*
- *Chapter 1: Safety*'s first item *Item 1: Limit mutability*

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

- *Chapter 1: Safety*'s last item is *Item 10: Write unit tests*
- *Chapter 2: Readability*'s last item is *Item 18: Respect coding conventions*
- *Chapter 3: Reusability*'s last item is *Item 25: 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 in such a way that chapters should be understandable independently of others. Having said that, you should read chapters from the beginning as chapters are constructed to make one flow.

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

Labels

It is impossible to write a book for everyone. This book is written primarily for experienced Kotlin developers. Many of them are already familiar with general best practices for programming and they look 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 one are those, I added the following labels at the beginning of such items:

- Not Kotlin-specific - 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 (those are items 10, 19, 26, 27, 36).

- Basics - 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 the title seems clear to you, skip such item (those are items 10, 18, 19, 20, 25, 26, 27).
- Edu - item is less about best practices and it is dedicated to teaching advanced Kotlin features that are useful for experienced Kotlin developers. If you know those features, skip such items (those are items 21, 22, 24, 32)

Suggestions

If you have any suggestions or corrections regarding this book, send them to contact@marcinmoskala.com

Acknowledgments

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



Márton Braun - A Kotlin enthusiast since version 1.0 of the language, and an aspiring writer, speaker, educator. Android developer and self-proclaimed Kotlin evangelist at Aut-Soft. Android/Kotlin tech editor for RayWenderlich.com. University student and instructor at BME-VIK, studying computer engineering while teaching Kotlin and Android. Creator of the MaterialDrawerKt and Krate libraries. Occasionally gets addicted to StackOverflow.

Márton highly influenced especially chapters 1 to 6 with useful comments, suggestions and corrections. He suggested changes in names of chapters, supported book reorganization, and contributed many important ideas.



David Blanc - After graduating in Computer Science from the INSA (a French Engineering school), David started working as a Java engineer for 8 years in various French IT companies, and moved to mobile application development on iOS and Android in 2012.

In 2015, he decided to focus on Android and joined i-BP, an IT department of the banking group BPCE, as 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 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 since he graduated from the University of Utah. He started evaluating Kotlin as a viable language since version 0.6 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 organisation. Jordan loves playing tabletop games with his family.

Jordan highly influenced most of the book, left many corrections and suggestions also to snippets and titles. He suggested deeper explanation for DSL, and how item dedicated to unit testing can be shortened. He protected correct technical wording.

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

Kirill Bubochkin - Left perfectly on-point and well thought comments all around the book.

Fabio Collini - Great review, especially for the Chapter 4 and 5. Inspired point that factory method can be inline in opposition to constructors, and how configuration can be stored in data classes.

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

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

Danilo Herrera - Influenced chapter 3, and highly influenced Chapter 4: Abstractions design.

Allan Caine - Highly influenced Chapter 5: Objects creation.

Edward Smith - Highly influenced Chapter 6: Class design.

Juan Manuel Rivero - Reviewed Chapter 6, 7 and 8.

I would also like to thank:

- Marta Raźniewska, who made drawings starting 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++? Developers are often bought by conciseness or amazing Kotlin features. For business, as I found out, the truly convincing argument is Kotlin safety - how its design eliminates potential application errors. You don't need to have any experience with development to get upset when the application you use crashes, or when there is an error on a website that does not let you check out after you spent an hour collecting products into a basket. Having fewer crashes makes the lives of both users and developers better, and it provides 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 and each of them is composed of different kinds of elements such as classes, objects, functions, type aliases and top-level properties. Some of those elements can hold state, for instance by having read-write property `var` or by composing a mutable object:

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

When an element holds 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 that has some money balance:

```
1 class BankAccount {
2     var balance = 0.0
3     private set
4
5     fun deposit(depositAmount: Double) {
6         balance += depositAmount
7     }
8
9     @Throws(InsufficientFunds::class)
10    fun withdraw(withdrawAmount: Double) {
11        if (balance < withdrawAmount) {
12            throw InsufficientFunds()
13        }
14        balance -= withdrawAmount
15    }
16 }
17
18 class InsufficientFunds : Exception()
19
```

```
20 val account = BankAccount()  
21 println(account.balance) // 0.0  
22 account.deposit(100.0)  
23 println(account.balance) // 100.0  
24 account.withdraw(50.0)  
25 println(account.balance) // 50.0
```

Here `BankAccount` has a state that represents how much money is present on that account. Holding state is a double-edged sword. On one hand it is very useful because it makes it possible to represent elements changing over time, but 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 changed when there are more of them. A class with many mutating points that depend on each other is often really hard to understand and to modify. It is especially problematic in case of unexpected situations or errors.
2. Mutability makes it harder to reason about the code. State of immutable element is clear. Mutable state is much harder to comprehend. It is harder to reason what is its value as it might change at any point and just because we checked in some moment it doesn't mean it is still the same.
3. It requires proper synchronization in multithreaded programs. Every mutation is a potential conflict.
4. Mutable elements are harder to test. We need to test every possible state, and the more mutability, the more states there are to test. What is more, 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 test all different combinations of possible states.
5. When state mutates, often some other classes need to be notified about this change. For instance, when we add a

mutable element to a sorted list, once the element is changed, we need to sort this list again.

Problems with state consistency and the growing complexity of the project with more mutation points are familiar for developers working in bigger teams. Let's see an example of how hard it is to manage shared state. Take a look at the below snippet⁶. It shows multiple threads trying to modify the same property, however because of conflicts some of those operations will be lost.

```
1 var num = 0
2 for (i in 1..1000) {
3     thread {
4         Thread.sleep(10)
5         num += 1
6     }
7 }
8 Thread.sleep(5000)
9 print(num) // Very unlikely to be 1000
10 // Every time a different number
```

When we use Kotlin coroutines, there are less conflicts because less threads are involved, but they still occur:

⁶To test it, just place it in an entry point (main function) and run. Similarly with other snippets not having an entry point.

```
1 suspend fun main() {
2     var num = 0
3     coroutineScope {
4         for (i in 1..1000) {
5             launch {
6                 delay(10)
7                 num += 1
8             }
9         }
10    }
11    print(num) // Every time a different number
12 }
```

In real-life projects, we generally cannot just lose some operations, and so we need to implement proper synchronization like the one presented below. Although implementing proper synchronization is hard, and the more mutation points we have, the harder it is. Limiting mutability does help.

```
1 val lock = Any()
2 var num = 0
3 for (i in 1..1000) {
4     thread {
5         Thread.sleep(10)
6         synchronized(lock) {
7             num += 1
8         }
9     }
10 }
11 Thread.sleep(1000)
12 print(num) // 1000
```

The drawbacks of mutability are so numerous that there are languages that do not allow state mutation at all. These are purely functional languages. One well-known example is Haskell. Such

languages are rarely used for mainstream development though, since it's very hard to do programming with so limited mutability. Mutating state is a very useful way to represent the state of real-world systems. I recommend using mutability, but do it sparingly and wisely decide where our mutating points should be. The good news is that Kotlin supports limiting mutability well.

Limiting mutability in Kotlin

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

- Read-only properties `val`
- Separation between mutable and read-only collections
- `copy` in data classes

Let's discuss them one by one.

Read-only properties `val`

In Kotlin we can make each property read-only `val` (like value) or read-write `var` (like variable). Read-only properties `val` do not allow setting:

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

Notice though that read-only properties are not necessarily immutable nor final. A read-only property can hold a mutable object:

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

```
1 var name: String = "Marcin"
2 var surname: String = "Moskała"
3 val fullName
4     get() = "$name $surname"
5
6 fun main() {
7     println(fullName) // Marcin Moskała
8     name = "Maja"
9     println(fullName) // Maja Moskała
10 }
```

Notice that it is possible because when we define a custom getter, it will be called every time we ask for the value.

```
1 fun calculate(): Int {
2     print("Calculating... ")
3     return 42
4 }
5
6 val fizz = calculate() // Calculating...
7 val buzz
8     get() = calculate()
9
10 fun main() {
11     print(fizz) // 42
12     print(fizz) // 42
13     print(buzz) // Calculating... 42
```

```
14     print(buzz) // Calculating... 42
15 }
```

This trait, that properties in Kotlin are encapsulated by default and they can have custom accessors (getters and setters) is very important in Kotlin because it gives us flexibility when we change or define our API. It will be described in detail in Item 16: Properties should represent state, not behavior. The core idea though is that `val` do not offer mutation points because it is only a getter under the hood when `var` is both getter and setter. That's why we can override `val` with `var`:

```
1 interface Element {
2     val active: Boolean
3 }
4
5 class ActualElement: Element {
6     override var active: Boolean = false
7 }
```

Values of read-only properties `val` can change, but such properties do not offer a mutation point which 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`.

Although remember that `val` doesn't mean immutable. It can be defined by getter or delegate. This fact gives us more freedom to change. Though when we don't need that, final properties should be preferred. It is easier to reason about them as they have the state stated next to their definition. They are also better supported in Kotlin. For instance, they can be smart-casted:

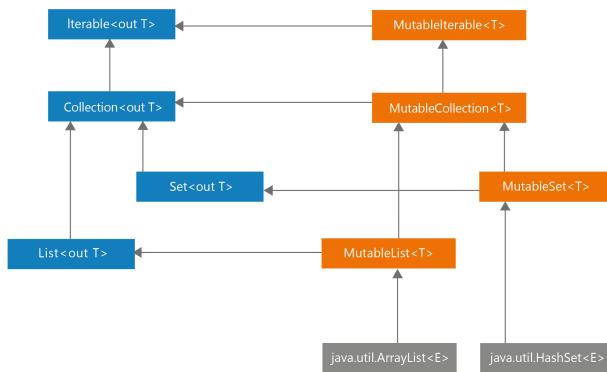
```
1 val name: String? = "Márton"
2 val surname: String = "Braun"
3
4 val fullName: String?
5     get() = name?.let { "$it $surname" }
6
7 val fullName2: String? = name?.let { "$it $surname" }
8
9 fun main() {
10     if (fullName != null) {
11         println(fullName.length) // ERROR
12     }
13
14     if (fullName2 != null) {
15         println(fullName2.length) // Márton Braun
16     }
17 }
```

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

Separation between mutable and read-only collections

Similarly, as Kotlin separates read-write and read-only properties, Kotlin separates read-write and read-only collections. This is achieved thanks to the way the hierarchy of collections was designed. Take a look at the diagram presenting collections hierarchy in Kotlin. On the left side, you can see the `Iterable`, `Collection`, `Set`, and `List` interfaces that 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 that 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 actual objects that can be used on Kotlin/JVM. On the left side interfaces are read-only. On the right side collections and interfaces are mutable.

Read-only collections are not necessarily immutable. Very often they are 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 below snippet you can see a simplified implementation of `Iterable<T>.map` from stdlib.

```
1 inline fun <T, R> Iterable<T>.map(  
2     transformation: (T) -> R  
3 ): List<R> {  
4     val list = ArrayList<R>()  
5     for (elem in this) {  
6         list.add(transformation(elem))  
7     }  
8     return list  
9 }
```

The design choice to make these collection interfaces read-only instead of truly immutable is very important. 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 the one achieved from having immutable collections. The only risk is when a developer tries to “hack the system” and performs 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 will be used only to read it. This is part of the contract. More about it on Part 2 of this book.

Down-casting collections is not only breaking their contract and depending 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:

```
1 val list = listOf(1,2,3)  
2  
3 // DON'T DO THIS!  
4 if (list is MutableList) {  
5     list.add(4)  
6 }
```

The result of this operation is platform-specific. On the JVM `listOf` returns an instance of `Arrays.ArrayList` that implements Java `List` interface. This Java `List` interface has methods like `add` or `set`, and so it translates to the Kotlin `MutableList` interface. However, `Arrays.ArrayList` does not implement some of those operations. This is why the result of the above code is the following:



```
Exception      in      thread      "main"
java.lang.UnsupportedOperationException
at java.util.AbstractList.add(AbstractList.java:148)
at java.util.AbstractList.add(AbstractList.java:108)
```

Though there is no guarantee how this will behave in a year from now. Underlying collections might change. They might be replaced with truly immutable collections implemented in Kotlin and not implementing `MutableList` at all. Nothing is guaranteed. This is why **down-casting read-only collections to mutable should never take place in Kotlin**. If you need to change from read-only to mutable, you should use `List.toMutableList` function, which creates a copy that you can then modify:

```
1 val list = listOf(1, 2, 3)
2
3 val mutableList = list.toMutableList()
4 mutableList.add(4)
```

This way does not break any contract, and it is also 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 already named 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 are created.
2. Immutability makes it easier to parallelize the program as there are no conflicts among shared objects.
3. References to immutable objects can be cached as they are not going to change.
4. We do not need to make defensive copies on immutable objects. When we do copy immutable objects, we do not need to make it a deep copy.
5. Immutable objects are the perfect material to construct other objects. Both mutable and immutable. We can still decide where mutability takes place, and it is easier to operate on immutable objects.
6. We can add them to set or use them as keys in maps, in opposition to mutable objects that shouldn't be used this way. This is because both those collections use hash table under the hood in Kotlin/JVM, and when we modify elements already classified to a hash table, its classification might not be correct anymore and we won't be able to find it. This problem will be described in detail in *Item 41: Respect the contract of hashCode*. We have a similar issue when a collection is sorted.

```
1 val names: SortedSet<FullName> = TreeSet()
2 val person = FullName("AAA", "AAA")
3 names.add(person)
4 names.add(FullName("Jordan", "Hansen"))
5 names.add(FullName("David", "Blanc"))
6
7 print(s) // [AAA AAA, David Blanc, Jordan Hansen]
8 print(person in names) // true
9
10 person.name = "ZZZ"
11 print(names) // [ZZZ AAA, David Blanc, Jordan Hansen]
12 print(person in names) // false
```

At the last check, collection returned false even though that person is in this set. It couldn't be found because it is in 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 the immutable objects should have methods that produce an object after some change. 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` after this 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 a `withSurname` method that produces a copy with a particular property changed:

```
1 class User(  
2     val name: String,  
3     val surname: String  
4 ) {  
5     fun withSurname(surname: String) = User(name, surname)  
6 }  
7  
8 var user = User("Maja", "Markiewicz")  
9 user = user.withSurname("Moskała")  
10 print(user) // User(name=Maja, surname=Moskała)
```

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

```
1 data class User(  
2     val name: String,  
3     val surname: String  
4 )  
5  
6 var user = User("Maja", "Markiewicz")  
7 user = user.copy(surname = "Moskała")  
8 print(user) // User(name=Maja, surname=Moskała)
```

This is an elegant and universal solution that supports making data model classes immutable. Surely, this way is less efficient than just using a mutable object instead, but it has all described advantages of immutable objects and 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 that. Either by using a mutable collection or by using read-write property `var`:

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

Both properties can be modified, but in different ways:

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

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

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

Both those ways are correct and they both have their pros and cons. They both have a single mutating point, but it is located in a different place. In the first one mutation takes place on the concrete list implementation. We might depend on the fact that it has proper synchronization in case of multithreading, but such an assumption is also dangerous since it is not really guaranteed. 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. Though, in case of a lack of synchronization, remember that we can still lose some elements:

```
1 var list = listOf<Int>()
2 for (i in 1..1000) {
3     thread {
4         list = list + i
5     }
6 }
7 Thread.sleep(1000)
8 print(list.size) // Very unlikely to be 1000,
9 // 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 using a delegate (which is using a custom setter). For instance, when we use an observable delegate, we can log every change of a list:

```
1 var names by Delegates.observable(listOf<String>()) {
2     _, old, new ->
3         println("Names changed from $old to $new")
4 }
5
6 names += "Fabio"
7 // Names changed from [] to [Fabio]
8 names += "Bill"
9 // 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 on mutable properties, it is also easier to control how they change - there is only a setter instead of multiple methods mutating this object, and we can make it private:

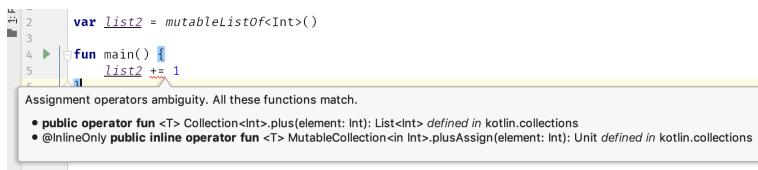
```
1 var announcements = listOf<Announcement>()
2     private set
```

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

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

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

We would need to synchronize both ways it can mutate (by property change and by internal state change). Also, changing it using plus-assign is impossible because of ambiguity:



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

Do not leak mutation points

It is an especially dangerous situation when we expose a mutable object that makes up state. Take a look at this example:

```
1 data class User(val name: String)  
2  
3 class UserRepository {  
4     private val storedUsers: MutableMap<Int, String> =  
5         mutableMapOf()  
6  
7     fun loadAll(): MutableMap<Int, String> {  
8         return storedUsers  
9     }  
10  
11    //...  
12 }
```

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

```
1 val userRepository = UserRepository()  
2  
3 val storedUsers = userRepository.loadAll()  
4 storedUsers[4] = "Kirill"  
5 //...  
6  
7 print(userRepository.loadAll()) // {4=Kirill}
```

It is especially dangerous when such modifications are accidental. There are two ways how we can deal with that. The first one is copying returned mutable objects. We call that *defensive copying*. This can be a useful technique when we deal with a standard objects and here copy generated by `data` modifier can be really helpful:

```
1 class UserHolder {
2     private val user: MutableUser()
3
4     fun get(): MutableUser {
5         return user.copy()
6     }
7
8     //...
9 }
```

Though whenever possible we prefer limiting mutability, and for collections we can do that by upcasting those objects to their read-only supertype:

```
1 data class User(val name: String)
2
3 class UserRepository {
4     private val storedUsers: MutableMap<Int, String> =
5         mutableMapOf()
6
7     fun loadAll(): Map<Int, String> {
8         return storedUsers
9     }
10
11    //...
12 }
```

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 a lot of tools that support limiting mutability. We should use them to limit mutation points. Simple rules are:

- Prefer `val` over `var`.

- Prefer an immutable property over a mutable one.
- Prefer objects and classes that are immutable over mutable.
- If you need them to change, consider making them immutable data classes, and using `copy`.
- When you hold state, prefer read-only over mutable collections.
- Design your mutation points wisely and do not produce unnecessary ones.
- Do not expose mutable objects.

There are some exceptions to these rules. Sometimes we prefer a mutable object because they are more efficient. Such optimizations should be preferred only in the performance critical parts of our code (Part 3: Efficiency) and 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: 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, so for instance, if a variable is used only inside a loop, defining it inside this loop

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

```
1 val a = 1
2 fun fizz() {
3     val b = 2
4     print(a + b)
5 }
6 val buzz = {
7     val c = 3
8     print(a + c)
9 }
10 // 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 variables defined in those functions. Here is an example of how limiting variable scope might look like:

```
1 // Bad
2 var user: User
3 for (i in users.indices) {
4     user = users[i]
5     print("User at $i is $user")
6 }
7
8 // Better
9 for (i in users.indices) {
10     val user = users[i]
11     print("User at $i is $user")
12 }
13
14 // Same variables scope, nicer syntax
15 for ((i, user) in users.withIndex()) {
16     print("User at $i is $user")
17 }
```

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), and it is better to define variables in as narrow 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 will be to break. This is a similar reason to why we prefer immutable properties or objects over their mutable counterparts.

Thinking about mutable properties, it is easier to track how

they change when they can only be modified in a smaller scope.
It is easier to reason about them and change their behavior.

Another problem is that **variables with a wider 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, like using this variable after the iteration to do something with the last element. It would be really bad because another developer 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 a variable to be initialized when it is defined. Do not force a developer to look where it was defined. This can be supported with control structures such as if, when, try-catch or the Elvis operator used as expressions:

```
1  // Bad
2  val user: User
3  if (hasValue) {
4      user = getValue()
5 } else {
6     user = User()
7 }
8
9 // Better
10 val user: User = if(hasValue) {
11     getValue()
12 } else {
13     User()
14 }
```

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

```
1 // Bad
2 fun updateWeather(degrees: Int) {
3     val description: String
4     val color: Int
5     if (degrees < 5) {
6         description = "cold"
7         color = Color.BLUE
8     } else if (degrees < 23) {
9         description = "mild"
10        color = Color.YELLOW
11    } else {
12        description = "hot"
13        color = Color.RED
14    }
15    // ...
16 }
17
18 // Better
19 fun updateWeather(degrees: Int) {
20     val (description, color) = when {
21         degrees < 5 -> "cold" to Color.BLUE
22         degrees < 23 -> "mild" to Color.YELLOW
23         else -> "hot" to Color.RED
24     }
25     // ...
26 }
```

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

Capturing

When I teach about Kotlin coroutines, one of my exercises is to implement 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 we filter out all the numbers that are divisible by this prime number.

A very simple implementation of this algorithm looks like this:

```
1 var numbers = (2..100).toList()
2 val primes = mutableListOf<Int>()
3 while (numbers.isNotEmpty()) {
4     val prime = numbers.first()
5     primes.add(prime)
6     numbers = numbers.filter { it % prime != 0 }
7 }
8 print(primes) // [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
9 // 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:

```
1 val primes: Sequence<Int> = sequence {
2     var numbers = generateSequence(2) { it + 1 }
3
4     while (true) {
5         val prime = numbers.first()
6         yield(prime)
7         numbers = numbers.drop(1)
8             .filter { it % prime != 0 }
9     }
10 }
```

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

Although in nearly every group there is a person who tries to “optimize” it, and to not create the variable in every loop he or she extracts `prime` as a mutable variable:

```
1 val primes: Sequence<Int> = sequence {
2     var numbers = generateSequence(2) { it + 1 }
3
4     var prime: Int
5     while (true) {
6         prime = numbers.first()
7         yield(prime)
8         numbers = numbers.drop(1)
9             .filter { it % prime != 0 }
10    }
11 }
```

The problem is that this implementation does not work correctly anymore. These are the first 10 yielded numbers:

```
1 print(primes.take(10).toList())
2 // [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” one 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 dropping works and 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 because such situations happen. To prevent them we should avoid mutability and prefer narrower scope for variables.

Summary

For many reasons, we should prefer to define variables for the closest possible scope. Also, we prefer `val` over `var` also for local variables. 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 3: Eliminate platform types as soon as possible

The null-safety introduced by Kotlin is amazing. Java was known in the community from Null-Pointer Exceptions (NPE), and Kotlin's safety mechanisms make them rare or eliminate them entirely. Although one thing that cannot be secured completely is a connection between a language that does not have solid null-safety - like Java or C - and Kotlin. 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 that it is nullable and we interpret it as a `String?`. If it is annotated with `@NotNull` then we trust this annotation and we type it as `String`. Though, what if this return type is not annotated with either of those annotations?

```
1 // Java
2 public class JavaTest {
3
4     public String giveName() {
5         // ...
6     }
7 }
```

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

The real problem would be when we would need to take generic types from Java. Imagine that a Java API returns a `List<User>` that is not annotated at all. If Kotlin would assume nullable types by default, and we would know that this list and those users are not

null, we would need to not only assert the whole list but also filter nulls:

```
1 // Java
2 public class UserRepo {
3
4     public List<User> getUsers() {
5         /**
6     }
7 }
8
9 // Kotlin
10 val users: List<User> = UserRepo().users!!.filterNotNull()
```

What if a function would return a `List<List<User>>` instead?
Gets complicated:

```
1 val users: List<List<User>> = UserRepo().groupedUsers!!
2     .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 denoted with a single exclamation mark ! after the type name, such as `String!`. Though this notation cannot be used in a code. Platform types are non-denotable, meaning that one cannot write them down explicitly in the language. When a platform value is assigned to a Kotlin variable or property, it can be inferred but it cannot be explicitly set. Instead, we can choose the type that we expect: Either a nullable or a non-null type.

```
1 // Java
2 public class UserRepo {
3     public User getUser() {
4         //...
5     }
6 }
7
8 // Kotlin
9 val repo = UserRepo()
10 val user1 = repo.user           // Type of user1 is User!
11 val user2: User = repo.user    // Type of user2 is User
12 val user3: User? = repo.user  // Type of user3 is User?
```

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

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

The problem is that is still dangerous because something we assumed to be not-null might be null. This is why for safety reasons I always suggest to be very conscientious of 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 by an annotation or by describing it in a comment, they can introduce this behavior without changing any contract.

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

```
1 // Java
2 import org.jetbrains.annotations.NotNull;
3
4 public class UserRepo {
5     public @NotNull User getUser() {
6         //...
7     }
8 }
```

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 into 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 by⁷:

- 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 support library `android.support.annotations`)
- 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`)

⁷You can see all currently supported annotations under this link:
<https://bit.ly/2XP5rb2>

Alternatively, you can specify in Java that all types should be Not-null by default using JSR 305's `@ParametersAreNonnullByDefault` annotation.

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

```
1 // Java
2 public class JavaClass {
3     public String getValue() {
4         return null;
5     }
6 }
7
8 // Kotlin
9 fun statedType() {
10     val value: String = JavaClass().value
11     //...
12     println(value.length)
13 }
14
15 fun platformType() {
16     val value = JavaClass().value
17     //...
18     println(value.length)
19 }
```

In both cases, a developer assumed that `getValue` will not return `null` and he or she was wrong. This results in an NPE in both cases, but there's a difference in where that 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 not-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 not-nullable. Possibly from a middle of some more complex expression. Variable typed as a platform type can be treated both as nullable and not-nullable. Such variable might be used few times safely, and then unsafely and throw NPE then. When we use such properties, typing system do not protect us. It is a similar situation as in Java, but in Kotlin we do not expect that we might have NPE just from using an object. It is very likely that sooner or later someone will use it unsafely, and then we will end up with a runtime exception and its cause might be not so easy to find.

```
1 // Java
2 public class JavaClass {
3     public String getValue() {
4         return null;
5     }
6 }
7
8 // Kotlin
9 fun platformType() {
10     val value = JavaClass().value
11     //...
12     println(value.length) // NPE
13 }
14
15 fun statedType() {
16     val value: String = JavaClass().value // NPE
17     //...
18     println(value.length)
19 }
```

What is even more dangerous, platform type might be propagated further. For instance, we might expose a platform type as a part of

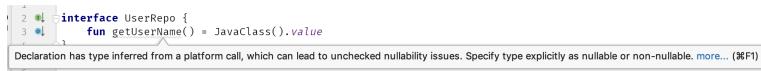
our interface:

```
1 interface UserRepo {  
2     fun getUsername() = JavaClass().value  
3 }
```

In this case, methods 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:

```
1 class RepoImpl: UserRepo {  
2     override fun getUsername(): String? {  
3         return null  
4     }  
5 }  
6  
7 fun main() {  
8     val repo: UserRepo = RepoImpl()  
9     val text: String = repo.getUsername() // NPE in runtime  
10    print("User name length is ${text.length}")  
11 }
```

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



Summary

Types that come from another language and has unknown nullability are known as platform types. Since they are dangerous,

we should eliminate them as soon as possible, and do not let them propagate. It is also good to specify types using annotations that specify nullability on exposed Java constructors, methods and fields. It is precious information both for Java and Kotlin developers using those elements.

Item 4: Do not expose inferred types

Kotlin's type inference is one of the most popular Kotlin features in the JVM world. So popular that Java 10 introduced type inference as well (limited comparing to Kotlin). Though, there are some dangers in using this feature. Above all, we need to remember that the inferred type of an assignment is the exact type of the right side, and not a superclass or interface:

```
1 open class Animal
2 class Zebra: Animal()
3
4 fun main() {
5     var animal = Zebra()
6     animal = Animal() // Error: Type mismatch
7 }
```

In most cases, this is not a problem. When we have too restrictive type inferred, we just need to specify it and our problem is solved:

```
1 open class Animal
2 class Zebra: Animal()
3
4 fun main() {
5     var animal: Animal = Zebra()
6     animal = Animal()
7 }
```

Though, there is no such comfort when we don't control a library or another module. In such cases, inferred type exposition can be really dangerous. Let's see an example.

Let's say that you have the following interface used to represent car factories:

```
1 interface CarFactory {  
2     fun produce(): Car  
3 }
```

There is also a default car used if nothing else was specified:

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

It is produced in most factories, so you made it default. You omitted the type because you decided that DEFAULT_CAR is a Car anyway:

```
1 interface CarFactory {  
2     fun produce() = DEFAULT_CAR  
3 }
```

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

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

Now, all your factories can only produce Fiat126P. Not good. If you defined this interface yourself, this problem will be probably caught soon and easily fixed. Though, if it is a part of the external API⁸, you might be informed first by angry users.

Except that, the return type is important information when someone does not know API well, and so for the sake of readability, we should make it explicit especially in parts of our API visible from outside (so exposed API).

⁸Elements (classes, functions, objects) that might be used from some outer module or some part of our code maintained by different developers. For instance, in libraries, those are all public and protected classes, functions and object declarations.

Summary

The general rule is that if we are not sure about the type, we should specify it. It is important information and we should not hide it (*Item 14: Specify the variable type when it is not clear*). Additionally for the sake of safety, in an external API, we should always specify types. We cannot let them be changed by accident. Inferred types can be too restrictive or can too easily change when our project evolves.

Item 5: Specify your expectations on arguments and state

When you have expectations, declare them as soon as possible. We do that in Kotlin mainly using:

- `require` block - a universal way to specify expectations on arguments.
- `check` block - a universal way to specify expectations on the state.
- `assert` block - a universal way to check if something is true. Such checks on the JVM will be evaluated only on the testing mode.
- Elvis operator with `return` or `throw`.

Here is an example using those mechanisms:

```
1 // Part of Stack<T>
2 fun pop(num: Int = 1): List<T> {
3     require(num <= size) {
4         "Cannot remove more elements than current size"
5     }
6     check(isOpen) { "Cannot pop from closed stack" }
7     val ret = collection.take(num)
8     collection = collection.drop(num)
9     assert(ret.size == num)
10    return ret
11 }
```

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

- Expectations are visible even to those programmers who are not reading the documentation.
- If they are not satisfied, the function throws an exception instead of leading to unexpected behavior. It is important that these exceptions are thrown before the state is modified and so we will not have a situation where only some modifications are applied and others are not. Such situations are dangerous and hard to manage⁹. Thanks to assertive checks, errors are harder to miss and our state is more stable.
- Code is to some degree self-checking. There is less of a need to be unit-tested when these conditions are checked in the code.
- All checks listed above work with smart-casting, so thanks to them there is less casting 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 to have some expectations on those arguments 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.

⁹I remember how in a Gameboy Pokemon game one could copy pokemon by making a transfer and disconnecting the cable at the right moment. After that, this pokemon was present on both Gameboy's. Similar hacks worked in many games and they generally involved turning off a Gameboy at a correct moment. General solution to all such problems is to make connected transactions atomic - either all occur or none occur. For instance, at once we add money to one account and subtract them from another. Atomic transactions are supported by most databases.

- When you look for clusters, you might require a list of points to not be empty.
- When you send an email to a user you might require that user to have an email, and this value to be a correct email address (assuming that user should check email correctness before using this function).

The most universal and direct Kotlin way to state those requirements is using the `require` function that checks this requirement and throws an exception if it is not satisfied:

```
1 fun factorial(n: Int): Long {  
2     require(n >= 0)  
3     return if (n <= 1) 1 else factorial(n - 1) * n  
4 }  
5  
6 fun findClusters(points: List<Point>): List<Cluster> {  
7     require(points.isNotEmpty())  
8     //...  
9 }  
10  
11 fun sendEmail(user: User, message: String) {  
12     requireNotNull(user.email)  
13     require(isValidEmail(user.email))  
14     //...  
15 }
```

Notice that these requirements are highly visible thanks to the fact they are declared at the very beginning of the functions. This makes them clear for the user reading those functions (though the requirements should be stated in documentation as well because not everyone reads function bodies).

Those expectations cannot be ignored, because the `require` function throws an

`IllegalArgumentException` when the predicate is not satisfied. When such a block is placed at the beginning of the function we know that if an argument is incorrect, the function will stop immediately and the user won't miss it. The exception will be clear in opposition to the potentially strange result that might propagate far until it fails. In other words, when we properly specify our expectations on arguments at the beginning of the function, we can then assume that those expectations will be satisfied.

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

```
1 fun factorial(n: Int): Long {  
2     require(n >= 0) { "Cannot calculate factorial of $n " +  
3         "because it is smaller than 0" }  
4     return if (n <= 1) 1 else factorial(n - 1) * n  
5 }
```

The `require` function is used when we specify requirements on arguments.

Another common case is when we have expectations on the current state, and in such a case, we can use the `check` function instead.

State

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

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

The standard way to check if those expectations on the state are satisfied is to use the `check` function:

```
1 fun speak(text: String) {  
2     check(isInitialized)  
3     //...  
4 }  
5  
6 fun getUserInfo(): UserInfo {  
7     checkNotNull(token)  
8     //...  
9 }  
10  
11 fun next(): T {  
12     check(isOpen)  
13     //...  
14 }
```

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. The 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. Although 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 the 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 it when we do not trust that our own implementation handles the state correctly. Although for such cases, when we are checking mainly for the sake of testing our own implementation, we have another function called `assert`.

Assertions

There are things we know to be true when a function is implemented correctly. For instance, when a function is asked to return

10 elements we might expect that it will return 10 elements. This is something we expect to be true, but it doesn't mean we are always right. We all make mistakes. Maybe we made a mistake in the implementation. Maybe someone changed a function we used and our function does not work properly anymore. Maybe our function does not work correctly anymore because it was refactored. For all those problems the most universal solution is that we should write unit tests that check if our expectations match reality:

```
1 class StackTest {  
2  
3     @Test  
4     fun `Stack pops correct number of elements`() {  
5         val stack = Stack(20) { it }  
6         val ret = stack.pop(10)  
7         assertEquals(10, ret.size)  
8     }  
9  
10    // ...  
11 }
```

Unit tests should be our most basic way to check implementation correctness but notice here that the fact that popped list size matches the desired one is rather universal to this function. It would be useful to add such a check in nearly every pop call. Having only a single check for this single use is rather naive because there might be some edge-cases. A better solution is to include the assertion in the function:

```
1 fun pop(num: Int = 1): List<T> {  
2     //...  
3     assert(ret.size == num)  
4     return ret  
5 }
```

Such conditions are currently enabled only in Kotlin/JVM, and they are not checked unless they are enabled using the `-ea` JVM option. We should rather treat them as part of unit tests - they check if our code works as expected. By default, they are not throwing any errors in production. They are enabled by default only when we run tests. This is generally desired because if we made an error, we might rather hope that the user won't notice. If this is a serious error that is probable and might have significant consequences, use `check` instead. The main advantages of having `assert` checks in functions instead of in unit tests are:

- Assertions make code self-checking and lead to more effective testing.
- Expectations are checked for every real use-case instead of for concrete cases.
- We can use them to check something at the exact point of execution.
- We make code fail early, closer to the actual problem. Thanks to that we can also easily find where and when the unexpected behavior started.

Just remember that for them to be used, we still need to write unit tests. In a standard application run, `assert` will not throw any exceptions.

Such assertions are a common practice in Python. Not so much in Java. In Kotlin feel free to use them to make your code more reliable.

Nullability and smart casting

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

```
1 public inline fun require(value: Boolean): Unit {  
2     contract {  
3         returns() implies value  
4     }  
5     require(value) { "Failed requirement." }  
6 }
```

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

```
1 fun changeDress(person: Person) {  
2     require(person.outfit is Dress)  
3     val dress: Dress = person.outfit  
4     //...  
5 }
```

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

```
1 class Person(val email: String?)  
2  
3 fun sendEmail(person: Person, message: String) {  
4     require(person.email != null)  
5     val email: String = person.email  
6     //...  
7 }
```

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” it:

```
1 class Person(val email: String?)  
2 fun validateEmail(email: String) { /*...*/ }  
3  
4 fun sendEmail(person: Person, text: String) {  
5     val email = requireNotNull(person.email)  
6     validateEmail(email)  
7     //...  
8 }  
9  
10 fun sendEmail(person: Person, text: String) {  
11     requireNotNull(person.email)  
12     validateEmail(person.email)  
13     //...  
14 }
```

For nullability, it is also popular to use the Elvis operator with `throw` or `return` on the right side. Such a structure is also highly readable and at the same time, 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:

```
1 fun sendEmail(person: Person, text: String) {  
2     val email: String = person.email ?: return  
3     //...  
4 }
```

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

```
1 fun sendEmail(person: Person, text: String) {  
2     val email: String = person.email ?: run {  
3         log("Email not sent, no email address")  
4         return  
5     }  
6     //...  
7 }
```

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

Summary

Specify your expectations to:

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

Four main mechanisms we use for that are:

- `require` block - a universal way to specify expectations on arguments.
- `check` block - a universal way to specify expectations on the state.
- `assert` block - a universal way to test in testing mode if something is true.
- Elvis operator with `return` or `throw`.

You might also use `error` `throw` to throw a different error.

Item 6: Prefer standard errors to custom ones

Functions `require`, `check` and `assert` cover the most common Kotlin errors, but there are also other kinds of unexpected situations we need to indicate. For instance, when you implement a library to parse the JSON format¹⁰, it is reasonable to throw a `JsonParsingException` when the provided JSON file does not have the correct format:

```
1 inline fun <reified T> String.readObject(): T {  
2     //...  
3     if (incorrectSign) {  
4         throw JsonParsingException()  
5     }  
6     //...  
7     return result  
8 }
```

Here we used a custom error because there is no suitable error in the standard library to indicate that 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 to understand. Here is the list of some of the most common exceptions you can use:

- `IllegalArgumentException` and `IllegalStateException` that we throw using `require` and `check` as described in Item 5: Specify your expectations on arguments and state.

¹⁰For the sake of practice. There are already great libraries for that, and they are already well tested, documented and optimized.

- **IndexOutOfBoundsException** - Indicate that the index parameter value is out of range. Used especially by collections and arrays. It is thrown for instance by `ArrayList.get(Int)`.
- **ConcurrentModificationException** - Indicate that concurrent modification is prohibited and yet it has been detected.
- **UnsupportedOperationException** - Indicate that the declared method is not supported by the object. Such a situation should be avoided and when a method is not supported, it should not be present in the class. ¹¹
- **NoSuchElementException** - Indicate 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.

¹¹One rule that is violated in such a case is the Interface Segregation Principle that states that no client should be forced to depend on methods it does not use.

Item 7: Prefer `null` or Failure result when the lack of result is possible

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

- We try to get data from some server, but there is a problem with our 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 the text, but this text is malformed

There are two main mechanisms to handle such situations:

- Return a `null` or a sealed class indicating failure (that is often named `Failure`)
- Throw an exception

There is an important difference between those 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). Main reasons for that are:

- The way exceptions propagate is less readable for most programmers and might be easily missed in the code.
- In Kotlin all exceptions are unchecked. Users are not forced or even encouraged to handle them. They are often not well documented. They are not really 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 compiler might otherwise perform.

On the other hand, `null` or `Failure` are both perfect to indicate an expected error. They are explicit, efficient, and can be handled in idiomatic ways. This is why the rule is that we **should prefer returning `null` or `Failure` when an error is expected, and throwing an exception when an error is not expected**. Here are some examples¹²:

```
1  inline fun <reified T> String.readObjectOrNull(): T? {  
2      //...  
3      if(incorrectSign) {  
4          return null  
5      }  
6      //...  
7      return result  
8  }  
9  
10 inline fun <reified T> String.readObject(): Result<T> {  
11     //...  
12     if(incorrectSign) {  
13         return Failure(JsonParsingException())  
14     }  
15     //...  
16     return Success(result)  
17 }  
18  
19 sealed class Result<out T>  
20 class Success<out T>(val result: T): Result<T>()  
21 class Failure(val throwable: Throwable): Result<Nothing>()  
22  
23 class JsonParsingException: Exception()
```

¹²Currently, in Kotlin 1.3, `kotlin.Result<T>` cannot be used as a return type so we implemented our own sum type representing success or failure.

Errors indicated this way are easier to handle and harder to miss. When we choose to use `null`, clients handling such a value can choose from the variety of null-safety supporting features like a safe-call or the Elvis operator:

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

When we choose to return a union type like `Result`, the user will be able to handle it using the `when`-expression:

```
1 val personResult = userText.readObject<Person>()
2 val age = when(personResult) {
3     is Success -> personResult.value.age
4     is Failure -> -1
5 }
```

Using such error handling is not only more efficient than the try-catch block but often also easier to use and more explicit. An exception can be missed and can stop our whole application. Whereas a `null` value or a sealed result class needs to be explicitly handled, but it won't interrupt the flow of the application.

Comparing nullable result and a sealed result class, we should prefer the latter when we need to pass additional information in case of failure, and `null` otherwise. Remember that `Failure` can hold any data you need.

It is common to have two variants of functions - one expecting that failure can occur and one treating it as an unexpected situation. A good example is that `List` has both:

- `get` which is used when we expect an element to be at the given position, and if it is not there, the function throws `IndexOutOfBoundsException`.

- `getOrNull`, which is used when we suppose that we might ask for an element out of range, and if that happens, we'll get `null`.

It also support other options, like `getOrDefault`, that is useful in some cases but in general might be easily replaced with `getOrNull` and Elvis operator `?:`.

This is a good practice because if developers know they are taking an element safely, they should not be forced to handle a nullable value, and at the same time, if they have any doubts, they should use `getOrNull` and handle the lack of value properly.

Item 8: Handle nulls properly

`null` means a lack of value. For property, it might mean that the value was not set or that it was removed. When a function returns `null` it might have a different meaning depending on the function:

- `String.toIntOrNull()` returns `null` when `String` cannot be correctly parsed to `Int`
- `Iterable<T>.firstOrNull(() -> Boolean)` returns `null` when there are no elements matching predicate from the argument.

In these and all other cases the meaning of `null` should be as clear as possible. This is because nullable values must be handled, and it is the API user (programmer using API element) who needs to decide how it is to be handled.

```
1 val printer: Printer? = getPrinter()
2 printer.print() // Compilation Error
3
4 printer?.print() // Safe call
5 if (printer != null) printer.print() // Smart casting
6 printer!!.print() // Not-null assertion
```

In general, there are 3 ways of how nullable types can be handled. We can:

- Handling nullability safely using safe call `?.`, smart casting, Elvis operator, etc.
- Throw an error
- Refactor this function or property so that it won't be nullable

Let's describe them one by one.

Handling nulls safely

As mentioned before, the two safest and most popular ways to handle nulls is by using safe call and smart casting:

```
1 printer?.print() // Safe call
2 if (printer != null) printer.print() // Smart casting
```

In both those cases, the function `print` will be called only when `printer` is not `null`. This is the safest option from the application user point of view. It is also really comfortable for programmers. No wonder why this is generally the most popular way how we handle nullable values.

Support for handling nullable variables in Kotlin is much wider than in other languages. One popular practice is to use the Elvis operator which provides a default value for a nullable type. It allows any expression including `return` and `throw`¹³ on its right side:

```
1 val printerName1 = printer?.name ?: "Unnamed"
2 val printerName2 = printer?.name ?: return
3 val printerName3 = printer?.name ?:
4     throw Error("Printer must be named")
```

Many objects have additional support. For instance, as it is common to ask for an empty collection instead of `null`, there is `Collection<T>?.isEmpty()` extension function returning not-nullable `List<T>`.

Smart casting is also supported by Kotlin's contracts feature that lets us smart cast in a function:

¹³It is possible because both `return` and `throw` declare `Nothing` as a return type which is a subtype of every type. More about it here: <https://bit.ly/2Gn6ftO>

```
1  println("What is your name?")
2  val name = readLine()
3  if (!name.isNullOrEmpty()) {
4      println("Hello ${name.toUpperCase()}")
5  }
6
7  val news: List<News>? = getNews()
8  if (!news.isNullOrEmpty()) {
9      news.forEach { notifyUser(it) }
10 }
```

All those options should be known to Kotlin developers, and they all provide useful ways to handle nullability properly.



Defensive and offensive programming

Handling all possibilities in a correct way - like here not using `printer` when it is `null` - is an implementation of *defensive programming*. *Defensive programming* is a blanket term for various practices increasing code stability once the code is in production, often by defending against the currently impossible. It is the best way when there is a correct way to handle all possible situations. It wouldn't be correct if we would expect that `printer` is not `null` and should be used. In such a case it would be impossible to handle such a situation safely, and we should instead use a technique called *offensive programming*. The idea behind *offensive programming* is that in case of an unexpected situation we complain about it loudly to inform the developer who led to such situation, and to force him or her to correct it. A direct implementation of this idea is `require`, `check` and `assert` presented in Item 5: Specify your expectations on arguments and state. It is important to understand that even though those two modes seem like being in conflict, they are not at all. They are more like yin and yang. Those are different modes both needed in our programs for the sake of safety, and we need to understand them both and use them appropriately.

Throw an error

One problem with safe handling is that if `printer` could sometimes be `null`, we will not be informed about it but instead `print` won't be called. This way we might have hidden important information. If we are expecting `printer` to never be `null`, we will be surprised when the `print` method isn't called. This can lead to errors that are really hard to find. When we have a strong expectation

about something that isn't met, it is better to throw an error to inform the programmer about the unexpected situation. It can be done using `throw`, as well as by using the not-null assertion `!!`, `requireNotNull`, `checkNotNull`, or other error throwing functions:

```
1 fun process(user: User) {  
2     requireNotNull(user.name)  
3     val context = checkNotNull(context)  
4     val networkService =  
5         getNetworkService(context) ?:  
6         throw NoInternetConnection()  
7     networkService.getData { data, userData ->  
8         show(data!!, userData!!)  
9     }  
10 }
```

The problems with the not-null assertion `!!`

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

A very simple example is a function looking for the largest among 4 arguments¹⁴. Let's say that we decided to implement it by packing all those arguments into a list and then using `max` to find the biggest

¹⁴In the Kotlin Standard Library such function is called `maxOf` and it can compare 2 or 3 values.

one. The problem is that it returns nullable because it returns `null` when the collection is empty. Though developer knowing that this list cannot be empty will likely use not-null assertion `!!`:

```
1 fun largestOf(a: Int, b: Int, c: Int, d: Int): Int =  
2     listOf(a, b, c, d).max()!!
```

As it was shown to me by Márton Braun who is a reviewer of this book, even in such a simple function not-null assertion `!!` can lead to NPE. Someone might need to refactor this function to accept any number of arguments and miss the fact that collection cannot be empty:

```
1 fun largestOf(vararg nums: Int): Int =  
2     nums.max()!!  
3  
4 largestOf() // NPE
```

Information about nullability was silenced and it can be easily missed when it might be important. Similarly with variables. Let's say that you have a variable that needs to be set later but it will surely be set before the first use. Setting it to `null` and using a not-null assertion `!!` is not a good option:

```
1 class UserControllerTest {  
2  
3     private var dao: UserDao? = null  
4     private var controller: UserController? = null  
5  
6     @BeforeEach  
7     fun init() {  
8         dao = mockk()  
9         controller = UserController(dao!!)  
10    }
```

```
11
12     @Test
13     fun test() {
14         controller!!.doSomething()
15     }
16
17 }
```

It is not only annoying that we need to unpack those properties every time, but we also block the possibility for those properties to actually have a meaningful `null` in the future. Later in this item, we will see that the correct way to handle such situations is to use `lateinit` or `Delegates.notNull`.

Nobody can predict how code will evolve in the future, and if you use not-null assertion `!!` or 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 null or a sealed result class result when the lack of result is possible). However, **explicit errors say much more than generic NPE and they should be nearly always preferred.**

Rare cases where not-null assertion `!!` does make sense are mainly a result of common interoperability with libraries in which nullability is not expressed correctly. When you interact with an API properly designed for Kotlin, this shouldn't be a norm.

In general **we should avoid using the not-null assertion `!!`**. This suggestion is rather widely approved by our community. Many teams have the policy to block 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 often is a code smell. **Seems like the way this operator looks like is no coincidence. `!!` seems to be screaming “Be careful” or “There is something wrong here”.**

Avoiding meaningless nullability

Nullability is a cost as it needs to be handled properly and we prefer avoiding nullability when it is not needed. `null` might pass an important message, but we should avoid situations where it would seem meaningless to other developers. They would then be tempted to use the unsafe not-null assertion `!!` or forced to repeat generic safe handling that only clutters the code. We should avoid nullability that does not make sense for a client. The most important ways for that are:

- Classes can provide variants of functions where the result is expected and in which lack of value is considered and nullable result or a sealed result class is returned. Simple example is `get` and `getOrNull` on `List<T>`. Those are explained in detail in Item 7: Prefer `null` or a sealed result class result when the lack of result is possible.
- Use `lateinit` property or `notNull` delegate when a value is surely set before use but later than during class creation.
- Do not return `null` instead of an empty collection. When we deal with a collection, like `List<Int>?` or `Set<String>?`, `null` has a different meaning than an empty collection. It means that no collection is present. To indicate a lack of elements, use an empty collection.
- Nullable enum and `None` enum value are two different messages. `null` is a special message that needs to be handled separately, but it is not present in the enum definition and so it can be added to any use-side you want.

Let's talk about `lateinit` property and `notNull` delegate as they deserve a deeper explanation.

lateinit property and notNull delegate

It is not uncommon in projects to have properties that cannot be initialized during class creation, but that will surely be initialized

before the first use. A typical example is when the property is set-up in a function called before all others, like in `@BeforeEach` in JUnit 5:

```
1  class UserControllerTest {
2
3      private var dao: UserDao? = null
4      private var controller: UserController? = null
5
6      @BeforeEach
7      fun init() {
8          dao = mockk()
9          controller = UserController(dao!!)
10     }
11
12     @Test
13     fun test() {
14         controller!!.doSomething()
15     }
16
17 }
```

Casting those properties from nullable to not null whenever we need to use them is highly undesirable. It is also meaningless as we expect that those values are set before tests. The correct solution to this problem is to use `lateinit` modifier that lets us initialize properties later:

```
1  class UserControllerTest {
2      private lateinit var dao: UserDao
3      private lateinit var controller: UserController
4
5      @BeforeEach
6      fun init() {
7          dao = mockk()
8          controller = UserController(dao)
9      }
10
11     @Test
12     fun test() {
13         controller.doSomething()
14     }
15 }
```

The cost of `lateinit` is that if we are wrong and we try to get value before it is initialized, then an exception will be thrown. Sounds scary but it is actually desired - we should use `lateinit` only when we are sure that it will be initialized before the first use. If we are wrong, we want to be informed about it. The main differences between `lateinit` and a nullable are:

- We do not need to “unpack” property every time to not-nullable
- We can easily make it nullable in the future if we need to use `null` to indicate something meaningful
- Once property is initialized, it cannot get back to an uninitialized state

`lateinit` is a 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 their lifecycle and we set properties in one of the first invoked methods to use it on the later ones. For instance when we set objects in `onCreate` in an

Android Activity, `viewDidAppear` in an iOS `UIViewController`, or `componentDidMount` in a React `React.Component`.

One case in which `lateinit` cannot be used is when we need to initialize a property with a type that, on JVM, associates to a primitive, like `Int`, `Long`, `Double` or `Boolean`. For such cases we have to use `Delegates.notNull` which is slightly slower, but supports those types:

```
1 class DoctorActivity: Activity() {
2     private var doctorId: Int by Delegates.notNull()
3     private var fromNotification: Boolean by
4         Delegates.notNull()
5
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         doctorId = intent.extras.getInt(DOCTOR_ID_ARG)
9         fromNotification = intent.extras
10            .getBoolean(FROM_NOTIFICATION_ARG)
11    }
12 }
```

These kinds of cases are often replaced with property delegates, like in the above example where we read the argument in `onCreate`, we could instead use a delegate that initializes those properties lazily:

```
1 class DoctorActivity: Activity() {
2     private var doctorId: Int by arg(DOCTOR_ID_ARG)
3     private var fromNotification: Boolean by
4         arg(FROM_NOTIFICATION_ARG)
5 }
```

The property delegation pattern is described in detail in Item 21: Use property delegation to extract common property patterns. One reason why they are so popular is that they help us safely avoid nullability.

Item 9: Close resources with `use`

There are resources that cannot be closed automatically, and we need to invoke the `close` method once we do not 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 the resource because these resources are rather expensive and they aren't easily closed by themselves (the Garbage Collector will eventually handle it if we do not keep any reference to this resource, but it will take some time). Therefore, to be sure that we will not miss closing them, we traditionally wrapped such resources in a `try-finally` block and called `close` there:

```
1 fun countCharactersInFile(path: String): Int {  
2     val reader = BufferedReader(FileReader(path))  
3     try {  
4         return reader.lineSequence().sumBy { it.length }  
5     } finally {  
6         reader.close()  
7     }  
8 }
```

Such a structure is complicated and incorrect. It is incorrect because `close` can throw an error, and such an error will not be caught. Also,

if we had errors from both the body of the `try` and from `finally` blocks, only one would be properly propagated. The behavior we should expect is for the information about the new error to be added to the previous one. The proper implementation of this is long and complicated, but it is also common and so it has been extracted into the `use` function from the standard library. It should be used to properly close resources and handle exceptions. This function can be used on any `Closeable` object:

```
1 fun countCharactersInFile(path: String): Int {  
2     val reader = BufferedReader(FileReader(path))  
3     reader.use {  
4         return reader.lineSequence().sumBy { it.length }  
5     }  
6 }
```

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

```
1 fun countCharactersInFile(path: String): Int {  
2     BufferedReader(FileReader(path)).use { reader ->  
3         return reader.lineSequence().sumBy { it.length }  
4     }  
5 }
```

As this support is often needed for files, and as 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:

```
1 fun countCharactersInFile(path: String): Int {  
2     File(path).useLines { lines ->  
3         return lines.sumBy { it.length }  
4     }  
5 }
```

This is a proper way to process even large files as 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 the file more than once, you need to open it more than once. The `useLines` function can be also used as an expression:

```
1 fun countCharactersInFile(path: String): Int =  
2     File(path).useLines { lines ->  
3         lines.sumBy { it.length }  
4     }
```

All the above implementations use sequences to operate on the file and it is the correct way to do it. Thanks to that we can always ready only one line instead of loading the content of the whole file. More about it in the item *Item 49: Prefer Sequence for big collections with more than one processing step.*

Summary

Operate on objects implementing `Closeable` or `AutoCloseable` using `use`. It is a safe and easy option. When you need to operate on a file, consider `useLines` that produce a sequence reading the next lines.

Item 10: 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 for that is to have different kinds of tests. One kind is checking that our application behaves correctly from the user's perspective. These kinds of tests are too often the only ones recognized by management as this is generally their primary goal to make the application behave correctly from outside, not internally. These kinds of tests do not even need developers at all. They can be handled by a sufficient number of testers or, what 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 tests that is much more useful for developers, and that is written by developers: unit tests. Here is an example unit test checking if our function `fib` calculating the Fibonacci number at n-th position gives us correct results for the first 5 numbers:

```
1 @Test
2 fun `fib works correctly for the first 5 positions`() {
3     assertEquals(1, fib(0))
4     assertEquals(1, fib(1))
5     assertEquals(2, fib(2))
6     assertEquals(3, fib(3))
7     assertEquals(5, fib(4))
8 }
```

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 were shown to be problematic in the past.
- Edge-cases and illegal arguments - for `Int` we might check for really big numbers like `Int.MAX_VALUE`. For a nullable object, it might be `null` or object filled with `null` values. There are no Fibonacci numbers for negative positions, so we might check how this function behaves then.

Unit tests can be really useful during development as they give fast feedback on how the implemented element works. Tests are only 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 write a unit test first and then implementation to satisfy it¹⁵.

The biggest advantages that result from unit tests are:

- **Well-tested elements tend to be more reliable.** There is also a psychological safety. When elements are well tested, we operate more confidently on 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

¹⁵Formally 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 those steps.

pleasurable¹⁶. It also helps reduce the cost of fixing bugs: the quicker you find them, the cheaper there are to fix them.

Clearly, there are also disadvantages to unit tests:

- It takes time to write unit tests. Though **in the long-term, good unit tests rather save our 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. It requires skills and understanding that are orthogonal to the rest of the development. 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 learn to unit test at least the important parts of the code**. Those are:

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

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

¹⁶I believe that no-one likes waiting for a project to fully build and start.

Summary

This chapter was started with a reflection that the first priority should be for our programs to behave correctly. It can be supported by using good practices presented in this chapter, but above that, 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 like 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∘.⊖-1 0 1∘.⍉c⍵}
```

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

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

These two are really 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 comes from the fact that Kotlin eliminates a lot of noise: boilerplate code and repetitive structures. It 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. This particular chapter serves as an introduction and provides a set of general suggestions. Although 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 start with a bit more abstract item about readability, which will introduce the general problem.

Item 11: 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 (this ratio was popularized by Robert C. Martin in the book *Clean Code*). If this seems unbelievable, just think about how much time you spend on 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 where they've been searching for an error for 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 the code to understand what is the logic or how implementation works. **Programming is mostly about reading, not writing.** Knowing that it should be clear that we should code with readability in mind.

Reducing cognitive load

Readability means something different to everyone. However, there are some rules that were formed based on experience or came from cognitive science. Just compare the following two implementations:

```
1 // Implementation A
2 if (person != null && person.isAdult) {
3     view.showPerson(person)
4 } else {
5     view.showError()
6 }
7
8 // Implementation B
9 person?.takeIf { it.isAdult }
10    ?.let(view::showPerson)
11    ?: 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 just as well remove the line breaks from the first implementation, and it wouldn't make it more readable.

How readable both constructs are, depends on how fast we can understand each of them. This, in turn, depends a lot on how much our brain is trained to understand each idiom (structure, function, pattern). For a beginner in Kotlin, surely implementation A is way more readable. It uses general idioms (`if/else`, `&&`, method calls). Implementation B has idioms that are typical to Kotlin (safe call `?..`, `takeIf`, `let`, Elvis operator `?:`, a bounded function reference `view::showPerson`). Surely, all those 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 the first language for most developers, 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, and that his person has never seen 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 will not be experienced with Kotlin. The brain will always need to spend a bit of time to recognize Kotlin-specific idioms. Even after years with Kotlin, it still takes much less time for me to understand the first implementation. Every less known idiom introduces a bit of complexity and 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 that we need to add additional operation on the `if` block. In the implementation A adding that is easy. In the implementation B we cannot use function reference anymore. Adding something to the

`else` block in the implementation B is even harder - we need to use some function to be able to hold more than a single expression on the right side of the Elvis operator:

```
1 if (person != null && person.isAdult) {  
2     view.showPerson(person)  
3     view.hideProgressWithSuccess()  
4 } else {  
5     view.showError()  
6     view.hideProgress()  
7 }  
8  
9 person?.takeIf { it.isAdult }  
10    ?.let {  
11        view.showPerson(it)  
12        view.hideProgressWithSuccess()  
13    } ?: run {  
14        view.showError()  
15        view.hideProgress()  
16    }
```

Debugging implementation A is also much simpler. No wonder why - debugging tools were made for such basic structures.

The general rule is that less common and “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 different error when person is `null` and different one when he or she is not an adult. On the implementation A using `if/else`, we can easily change `if/else` to `when` using IntelliJ refactoring, and then easily add additional branch. The same change on the code would be painful on the implementation B. It would probably need it to be totally rewritten.

Have you noticed that implementation A and B do not even 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` would return `null`, then the second implementation would call `showError` as well! This is definitely not obvious, and it teaches us that when we use less familiar structures, it is easier to spot unexpected behavior.

The general rule here is that we want to reduce cognitive load. Our brains recognize patterns and based on these patterns they build our 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 in the previous example I presented how `let` can be misused, it does not mean that it should be always avoided. It is a popular idiom reasonably used in a variety of contexts to actually make code better. One popular 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 mutable property can be modified by another thread. One great way to deal with that is to use safe call `let`:

```
1 class Person(val name: String)
2 var person: Person? = null
3
4 fun printName() {
5     person?.let {
6         print(it.name)
7     }
8 }
```

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

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

Here are examples showing those two (both additionally use function references):

```
1 students
2     .filter { it.result >= 50 }
3     .joinToString(separator = "\n") {
4         "${it.name} ${it.surname}, ${it.result}"
5     }
6     .let(::print)
7
8 var obj = FileInputStream("/file.gz")
9     .let(::BufferedInputStream)
10    .let(::ZipInputStream)
11    .let(::ObjectInputStream)
12    .readObject() as SomeObject
```

In all those cases we pay our price - this code is harder to debug and harder to be understood by less experienced Kotlin developers. But we pay for something and it seems like a fair deal. The problem is when we introduce a lot of complexity for no good reason.

There will always be discussions when something makes sense and when it does not. Balancing that is an art. It is good though to recognize how different structures introduce complexity or how they clarify things. Especially since when they are used together, the complexity of two structures 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 and what implicit, what idioms should we 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:

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

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

```
1 operator fun String.invoke(f: ()->String): String =  
2     this + f()  
3  
4 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. A `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 guards 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 12: Operator 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:

```
1 fun Int.factorial(): Int = (1..this).product()  
2  
3 fun Iterable<Int>.product(): Int =  
4     fold(1) { acc, i -> acc * i }
```

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

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

A mathematician will know that there is a special notation for factorials. It is an exclamation mark after a number:

```
1 10 * 6!
```

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

```
1 operator fun Int.not() = factorial()  
2  
3 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. How would the following look like?

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

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

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, known only to the 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 it all in mind to connect the pieces to understand the whole statement. We don't have such a problem in Kotlin, because each of these operators 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 are functions with a concrete name, and we expect all functions to do what their names indicate. This highly restricts what each operator can be used for. Using `not` to return `factorial` is a clear breach of this convention, and should never happen.

Unclear cases

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

```

1 operator fun Int.times(operation: () -> Unit): ()->Unit =
2     { repeat(this) { operation() } }
3
4 val tripledHello = 3 * { print("Hello") }
5
6 tripledHello() // Prints: HelloHelloHello

```

For others, it might be clear that it means that we want to call this function 3 times¹⁷:

```

1 operator fun Int.times(operation: ()->Unit) {
2     repeat(this) { operation() }
3 }
4
5 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:

```

1 infix fun Int.timesRepeated(operation: ()->Unit) = {
2     repeat(this) { operation() }
3 }
4
5 val tripledHello = 3 timesRepeated { print("Hello") }
6 tripledHello() // Prints: HelloHelloHello

```

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

¹⁷The difference is that the first one is producing a function while the other one calls a function. In the first case the result of multiplication is ()->Unit and in the second case it is Unit

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

When is it fine to break this rule?

There is one very important case when 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:

```
1 body {
2     div {
3         +"Some text"
4     }
5 }
```

You can see that to add text into 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 a surprise to readers that different rules apply.

Summary

Use operator overloading conscientiously. The function 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 13: 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:

```
1 fun keyIsCorrect(key: String): Boolean = //...
2
3 if(!keyIsCorrect(key)) return
```

We are able to do this:

```
1 fun verifyKey(key: String): Unit? = //...
2
3 verifyKey(key) ?: return
```

This appears to be the expected answer. What was missing in my friend’s interview was a way more important question: “But 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:

```
1 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:

```
1 if (person != null && person.isAdult) {  
2     view.showPerson(person)  
3 } else {  
4     view.showError()  
5 }
```

Compare the following two notations:

```
1 if(!keyIsCorrect(key)) return  
2  
3 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 14: Specify the variable type when it is not clear

Kotlin has a great type inference system that allows us to omit types when those are obvious for developers:

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

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

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

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

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

Improving readability is not the only case for type specification. It is also for safety as shown in the *Chapter: Safety* on *Item 3: Eliminate platform types as soon as possible* and *Item 4: Do not*

expose inferred types. Type might be important information both for a developer and for the compiler. Whenever it is, do not bother to specify the type. It costs little and can help a lot.

Item 15: 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 it means a reference to the class to which the method is associated:

```
1  class User: Person() {  
2      private var beersDrunk: Int = 0  
3  
4      fun drinkBeers(num: Int) {  
5          // ...  
6          this.beersDrunk += num  
7          // ...  
8      }  
9  }
```

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 written without explicit receivers:

```
1  fun <T : Comparable<T>> List<T>.quickSort(): List<T> {  
2      if (size < 2) return this  
3      val pivot = first()  
4      val (smaller, bigger) = drop(1)  
5          .partition { it < pivot }  
6      return smaller.quickSort() + pivot + bigger.quickSort()  
7  }
```

With this one written using them:

```

1 fun <T : Comparable<T>> List<T>.quickSort(): List<T> {
2     if (this.size < 2) return this
3     val pivot = this.first()
4     val (smaller, bigger) = this.drop(1)
5         .partition { it < pivot }
6     return smaller.quickSort() + pivot + bigger.quickSort()
7 }
```

The usage is the same for both functions:

```

1 listOf(3, 2, 5, 1, 6).quickSort() // [1, 2, 3, 5, 6]
2 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 and we should avoid them. It is safer to use an object using explicit receiver. To understand this problem, see the following code¹⁸:

```

1 class Node(val name: String) {
2
3     fun makeChild(childName: String) =
4         create("$name.$childName")
5             .apply { print("Created ${name}") }
6
7     fun create(name: String): Node? = Node(name)
8 }
9
```

¹⁸Inspired by a puzzler initially added by Roman Dawydkin to the Dmitry Kandalov collection and later presented on KotlinConf by Anton Keks.

```
10 fun main() {  
11     val node = Node("parent")  
12     node.makeChild("child")  
13 }
```

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

It is probably expected that the result should be “Created parent.child”, but the actual result is “Created parent”. Why? To investigate, we can use explicit receiver before name:

```
1 class Node(val name: String) {  
2  
3     fun makeChild(childName: String) =  
4         create("$name.$childName")  
5             .apply { print("Created ${this.name}") }  
6             // Compilation error  
7  
8     fun create(name: String): Node? = Node(name)  
9 }
```

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

```
1 class Node(val name: String) {  
2  
3     fun makeChild(childName: String) =  
4         create("$name.$childName")  
5             .apply { print("Created ${this?.name}") }  
6  
7     fun create(name: String): Node? = Node(name)  
8 }  
9  
10 fun main() {  
11     val node = Node("parent")  
12     node.makeChild("child")  
13     // Prints: Created parent.child  
14 }
```

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

```
1 class Node(val name: String) {  
2  
3     fun makeChild(childName: String) =  
4         create("$name.$childName")  
5             .also { print("Created ${it?.name}") }  
6  
7     fun create(name: String): Node? = Node(name)  
8 }
```

When receiver is not clear, we either prefer to avoid it or we clarify it using explicit receiver. When we use receiver without label, we mean the closest one. When we want to use outer receiver we need to use a label. In such case it is especially useful to use it explicitly. Here is an example showing they both used:

```
1  class Node(val name: String) {  
2  
3      fun makeChild(childName: String) =  
4          create("$name.$childName").apply {  
5              print("Created ${this?.name} in "+  
6                  " ${this@Node.name}")  
7          }  
8  
9      fun create(name: String): Node? = Node(name)  
10 }  
11  
12 fun main() {  
13     val node = Node("parent")  
14     node.makeChild("child")  
15     // Created parent.child in parent  
16 }
```

This way direct receiver clarifies what receiver do we mean. This might be an 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 on 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 DSLs are designed in such a way. However, in DSLs, it is especially dangerous to accidentally use functions from an outer scope. Think of a simple HTML DSL we use to make an HTML table:

```
1  table {  
2      tr {  
3          td { +"Column 1" }  
4          td { +"Column 2" }  
5      }  
6      tr {  
7          td { +"Value 1" }  
8          td { +"Value 2" }  
9      }  
10 }
```

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 DSL:

```
1  table {  
2      tr {  
3          td { +"Column 1" }  
4          td { +"Column 2" }  
5          tr {  
6              td { +"Value 1" }  
7              td { +"Value 2" }  
8          }  
9      }  
10 }
```

To restrict such usage, we have a special meta-annotation (an annotation for annotations) that restricts 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, inside this builder implicit receiver use won't be possible. Here is an example of how `DslMarker` might be used:

```
1 @DslMarker
2 annotation class HtmlDsl
3
4 fun table(f: TableDsl.() -> Unit) { /*...*/ }
5
6 @HtmlDsl
7 class TableDsl { /*...*/ }
```

With that, it is prohibited to use outer receiver implicitly:

```
1 table {
2     tr {
3         td { +"Column 1" }
4         td { +"Column 2" }
5         tr { // COMPILATION ERROR
6             td { +"Value 1" }
7             td { +"Value 2" }
8         }
9     }
10 }
```

Using functions from an outer receiver requires explicit receiver usage:

```
1 table {
2     tr {
3         td { +"Column 1" }
4         td { +"Column 2" }
5         this@table.tr {
6             td { +"Value 1" }
7             td { +"Value 2" }
8         }
9     }
10 }
```

The DSL marker is a very important mechanism that we use to force usage of either the closest receiver or explicit receiver usage. However, it is generally better to not use explicit receivers in DSLs anyway. Respect DSL design and use it accordingly.

Summary

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

Item 16: Properties should represent state, not behavior

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

```
1 // Kotlin property
2 var name: String? = null
3
4 // Java field
5 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. Starting with the fact that they can always have custom setters and getters:

```
1 var name: String? = null
2     get() = field?.toUpperCase()
3     set(value) {
4         if(!value.isNullOrEmpty()) {
5             field = value
6         }
7     }
```

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

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

For a read-write property `var`, we can make a property by defining a getter and 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 used `Date` from the Java stdlib. Then at some point for a reason, the object cannot store the property of this type anymore. Perhaps because of a serialization issue, or maybe because you lifted this object 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 property `millis`, and modify the `date` property to not hold data but instead to wrap/unwrap that other property.

```
1 var date: Date  
2     get() = Date(millis)  
3     set(value) {  
4         millis = value.time  
5     }
```

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:

```
1 interface Person {  
2     val name: String  
3 }
```

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

```
1 open class Supercomputer {  
2     open val theAnswer: Long = 42  
3 }  
4  
5 class AppleComputer : Supercomputer() {  
6     override val theAnswer: Long = 1_800_275_2273  
7 }
```

For the same reason, we can delegate properties:

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

Property delegation is described in detail in Item 21: Use property delegation to extract common property patterns. Because properties are essentially functions, we can make extension properties as well:

```
1 val Context.preferences: SharedPreferences  
2     get() = PreferenceManager  
3             .getDefaultSharedPreferences(this)  
4  
5 val Context.inflater: LayoutInflater  
6     get() = getSystemService(  
7             Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater  
8  
9 val Context.notificationManager: NotificationManager  
10    get() = getSystemService(Context.NOTIFICATION_SERVICE)  
11        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 behaviour like in the example below:

```
1 // DON'T DO THIS!
2 val Tree<Int>.sum: Int
3     get() = when (this) {
4         is Leaf -> value
5         is Node -> left.sum + right.sum
6     }
```

Here `sum` property iterates over all elements and so it represents algorithmic behavior. Therefore this property is misleading: finding the answer can be computationally heavy for big collections, and this is not expected at all for a getter. This should not be a property, this should be a function:

```
1 fun Tree<Int>.sum(): Int = when (this) {
2     is Leaf -> value
3     is Node -> left.sum() + right.sum()
4 }
```

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

- **Operation is computationally expensive or has computational complexity higher than O(1)** - A user does not expect that using a property might be expensive. If it is, using a function is better because it communicates that it might be and that user might be parsimonious using it, or the developer might consider caching it.
- **It involves business logic (how the application acts)** - when we read code, we do not expect that a property might do anything more than simple actions like logging, notifying listeners, or updating a bound element.

- **It is not deterministic** - Calling the member twice in succession produces different results.
- **It is a conversion, such as Int.toDouble()** - It is a matter of convention that conversions are a method or an extension function. Using a property would seem like referencing some inner part instead of wrapping the whole object.
- **Getters should not change property state** - We expect that we can use getters freely without worrying about property state modifications.

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

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

On the other hand, to get and set 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, and if you need to modify them later, use custom getters and setters:

```
1 // DON'T DO THIS!
2 class UserIncorrect {
3     private var name: String = ""
4
5     fun getName() = name
6
7     fun setName(name: String) {
8         this.name = name
9     }
10 }
11
12 class UserCorrect {
13     var name: String = ""
14 }
```

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

Item 17: Consider naming arguments

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

```
1 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 all¹⁹. We can make it easier to read by clarifying those arguments whose values do not clearly indicate what they mean. The best way to do that is by using named arguments:

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

We could achieve a similar result by naming variable:

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

Although naming the argument is more reliable. A variable name specifies developer intention, but not necessarily 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 while named values do not. This is why it is still reasonable to use named arguments when we have values named anyway:

¹⁹IntelliJ now helps by displaying hints when you put literals in a function call, but this can be turned off or one might use a different IDE.

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

When should we use named arguments?

Clearly, named arguments are longer, but they have two important advantages:

- Name that indicates what value is expected.
- They are safer because they are independent of order.

The argument name is important information not only for a developer using this function but also for the one reading how it was used. Take a look at this call:

```
1 sleep(100)
```

How much will it sleep? 100 ms? Maybe 100 seconds? We can clarify it using a named argument:

```
1 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 time unit:

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

Or we could use an extension property to create a DSL-like syntax:

```
1 sleep(100.ms)
```

Types are a good way to pass such information. If you are concerned about efficiency, use inline classes as described in *Item 46: Use inline modifier for functions with parameters of functional types*. They help us with parameter safety, but they do not solve all problems. Some arguments might still be unclear. Some arguments might still be placed on wrong positions. This is why I still suggest considering named arguments, especially to parameters:

- with default arguments,
- with the same type as other parameters,
- of functional type, if they're not the last parameter.

Parameters with default arguments

When a property has a default argument, we should nearly always use it by name. Such optional parameters are changed more often than those that are required. We don't want to miss such a change. Function name generally indicates what are its non-optional arguments, but not what are its optional ones. This is why it is safer and generally cleaner to name optional arguments.²⁰

Many parameters with the same type

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

²⁰Some best practices for different languages even suggest to always name optional arguments. One popular example is Effective Python by Brett Slatkin with Item 19: Provide Optional Behavior with Keyword Arguments.

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

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

```
1 sendEmail(  
2     to = "contact@kt.academy",  
3     message = "Hello, ..."  
4 )
```

Parameters of function type

Finally, we should treat parameters with function types specially. There is one 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 that is the block of code that should be repeated. When you see `thread`, it is intuitive that the block after that is the body of this new thread. Such names only describe the function used at the last position.

```
1 thread {  
2     // ...  
3 }
```

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:

```
1 val view = linearLayout {  
2     text("Click below")  
3     button({ /* 1 */ }, { /* 2 */ })  
4 }
```

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

```
1 val view = linearLayout {  
2     text("Click below")  
3     button(onClick = { /* 1 */ }) {  
4         /* 2 */  
5     }  
6 }
```

Multiple optional arguments of a function type can be especially confusing:

```
1 fun call(before: ()->Unit = {}, after: ()->Unit = {}){  
2     before()  
3     print("Middle")  
4     after()  
5 }  
6  
7 call({ print("CALL") }) // CALLMiddle  
8 call { print("CALL") } // MiddleCALL
```

To prevent such situations, when there is no single argument of a function type with special meaning, name them all:

```
1 call(before = { print("CALL") }) // CALLMiddle  
2 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 case of error,
- after the observable is finished.

In Java I've often seen people setting them up using lambda expressions, and specifying in comments which method each lambda expression is:

```
1 // Java
2 observable.getUsers()
3     .subscribe((List<User> users) -> { // onNext
4         // ...
5     }, (Throwable throwable) -> { // onError
6         // ...
7     }, () -> { // onCompleted
8         // ...
9 });

```

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

```
1 observable.getUsers()
2     .subscribeBy(
3         onNext = { users: List<User> ->
4             // ...
5         },
6         onError = { throwable: Throwable ->
7             // ...
8         },
9         onCompleted = {
10             // ...
11         })

```

Notice that I changed function name from `subscribe` to `subscribeBy`. It is because RxJava is written in Java and **we cannot use named arguments when we call Java functions**. It is because Java does

not preserve information about function names. To be able to use named arguments we often need to make our Kotlin wrappers for those functions (extension functions that are alternatives to those functions).

Summary

Named arguments are not only useful when we need to skip some default values. They are important information for developers reading our code, and they can improve the safety of our code. We should consider them especially when we have more 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 described in the documentation in a section aptly called “Coding Conventions”²¹. Those 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 those conventions as they are described in the documentation. They should also be respected when they change - which 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 “Set from...” link in the upper right corner, and select “Predefined style / Kotlin style guide” from the menu.
- ktlint²² - popular linter that analyzes your code and notifies you about all coding conventions 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 are post-Java developers. One rule that I see often

²¹Link: <https://kotlinlang.org/docs/reference/coding-conventions.html>

²²Link: <https://github.com/pinterest/ktlint>

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 in a way so that every parameter is on another line²³, and there is no parameter in the first line:

```
1 class Person(  
2     val id: Int = 0,  
3     val name: String = "",  
4     val surname: String = ""  
5 ) : Human(id, name) {  
6     // body  
7 }
```

Similarly, this is how we format a long function:

```
1 public fun <T> Iterable<T>.joinToString(  
2     separator: CharSequence = ", ",  
3     prefix: CharSequence = "",  
4     postfix: CharSequence = "",  
5     limit: Int = -1,  
6     truncated: CharSequence = "...",  
7     transform: ((T) -> CharSequence)? = null  
8 ): String {  
9     // ...  
10 }
```

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

²³It is possible to have parameters strongly related to each other like x and y on the same line, though personally, I am not in favour of that.

```
1 // Don't do that
2 class Person(val id: Int = 0,
3             val name: String = "",
4             val surname: String = "") : Human(id, name) {
5     // body
6 }
```

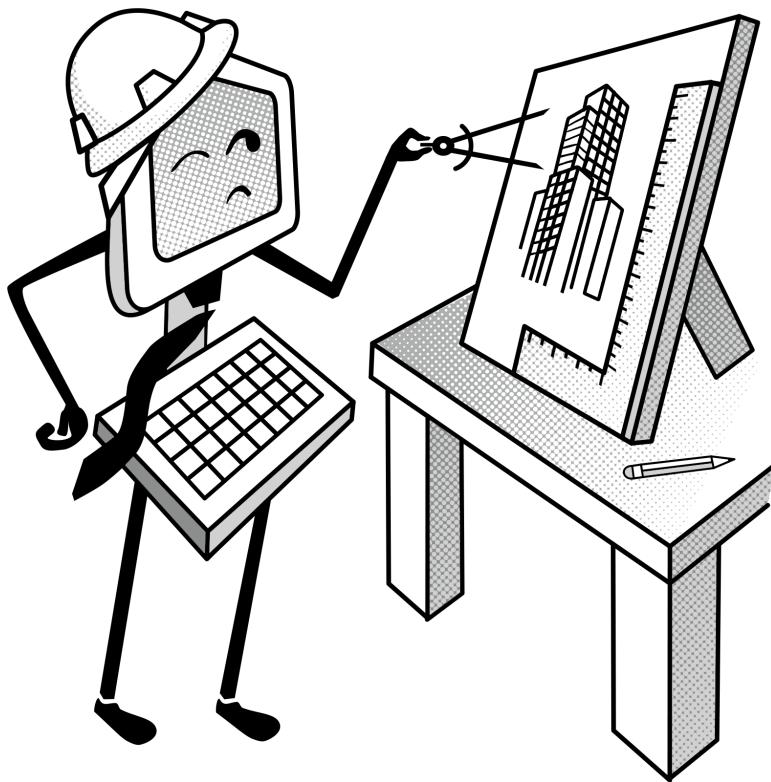
It can be problematic in 2 ways:

- Arguments on 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 tend to be still too wide. Width of the class defined this way is the class name with `class` keyword and the longest primary constructor parameter, or last parameter plus superclasses and interfaces.

Some teams might decide to use slightly different conventions. This is fine, but then those conventions should be respected all around the given 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, and 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, 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 (`print` function in Kotlin/JVM)? It's one of the most basic functions used again and again and yet, would you be able to implement it yourself if it would vanish one day? Truth is that this is not an easy task. Especially if the rest of `java.io` would vanish as well. You would need to implement the communication with the operating system in C using JNI, separately for each operating system you support²⁴. 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. A backend developer doesn't need to know too much about HTTP and HTTPS protocols even though they work with them every day. You don't need to know any sorting algorithms to call `Iterable.sorted`. 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*.

Sounds enthusiastic, 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 if we need to change them both, but it's harder and more error-prone when we need to change only one.

²⁴Read about it in this article: <https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works/>

This chapter is dedicated to reusability. It touches on many subjects that developers do intuitively. We do so because we learned them through practice. Mainly through observations of how something we did in the past has an impact on us now. We extracted something, and now it causes problems. We haven't extracted something, and now we have a problem when we need to make some changes. Sometimes we deal with code written by different developers years ago, 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.

It has one problem though: it requires years of practice. To speed it up and to 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.

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. Till today whenever I reflect on that, I am amazed how well a single and clear sentence expresses the key idea behind the “Do not repeat knowledge” principle. It is also often known as DRY principle after the Pragmatic Programmer book that described the Don’t Repeat Yourself rule. Some developers might be familiar with the WET antipattern²⁵, that sarcastically teaches us the same. DRY is also connected to the Single Source of Truth (SSOT) practice. As you can see, this rule is quite popular and has many names. However, 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 can be stated by code or data. It can also be stated by 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.

With knowledge defined this way, everything in our projects is some kind of knowledge. Of course, there are many different kinds of knowledge: how an algorithm should work, what UI should look

²⁵Standing for We Enjoy Typing, Waste Everyone’s Time or Write Everything Twice.

like, what result we wish to achieve, etc. There are also many ways to express it: 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 expected behavior

The main difference between them is that business logic changes a lot over time, while common algorithms generally do not 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 will concentrate on algorithms in the next item. For now, let's concentrate on the first point, that is the logic - knowledge about our program.

Everything can change

There is a saying that in programming the only constant is change. Just think of projects from 10 or 20 years ago. It is not such a long time. Can you point a single popular application or website that hasn't changed? 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 questions were the same as the previous year. Einstein responded that it was true, but 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 knowledge in our projects will change as well. For instance, here are very typical reasons for the change:

- The company learns more about user needs or habits
- Design standards change
- We need to adjust to changes in the platform, libraries, or some tools

Most projects nowadays change requirements and parts of the internal structure every few months. This is often something desired. Many popular management systems are agile and fit to support constant changes in requirements. Slack was initially a game named Glitch²⁶. 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 changes 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 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 are already modified because they were integrated with other functionalities? It might be tough to change them all in the same way. Those are real problems.

²⁶See the presentation How game mechanics can drive product loyalty by Ali Rayl, <https://vimeo.com/groups/359614/videos/154931121>

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

Another example: Let's say that we use a database in our project, and 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 some table structure defined only once, we wouldn't have such a problem.

On both examples, you can see how dangerous and problematic knowledge repetition is. It makes projects less scalable and more fragile. Good news is that we, programmers, work 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 custom view/component to represent it. Instead of writing SQL in text format, we can use an ORM (like Hibernate) or DAO (like Exposed).

All those solutions represent different kinds of abstractions and they protect us from a different kinds of redundancy. Analysis of different kinds of abstractions is presented in *Item 27: 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. This is when they only look similar but represent different knowledge.

Let's start from an example. Let's say that we have two independent Android applications in the same project. Their build tool configurations are similar so it might be tempting to extract it.

But what if we do that? Applications are independent so if we will need to change something in the configuration, we will most likely need to change it only in one of them. Changes after this reckless extraction are harder, not easier. Configuration reading is harder as well - configurations have their boilerplate code, but developers are already familiar with it. Making abstractions means designing our own API, which is another thing to learn for a developer using this API. This is a perfect example of how problematic it is when we 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 going to change together or separately?** Pragmatically this is the most important question because this is the biggest result of having a common part extracted: it is easier to change them both, but it is harder to change only a single one.

One useful heuristic is that if business rules come from different sources, we should assume that they will more likely 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*.

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 rule²⁷ can be simplified by the statement that there should be no such situations when two actors need to change the same class. By actor, we mean a source of change. They are often personified by developers from different departments who know little about each other’s work and domain. Although even if there is only a single developer in a project, but having multiple managers, they should

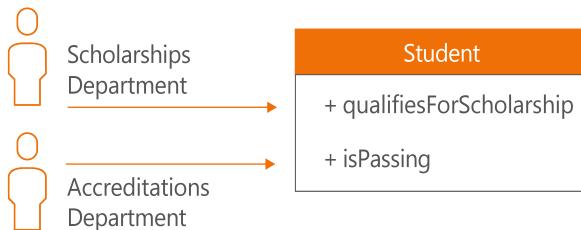
²⁷As described by the inventor, Robert C. Martin, in his book Clean Architecture

be treated as separate actors. Those are two sources of changes knowing little about each other domains. The situation when 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 class `Student`. This class is used both by the Scholarships Department and the Accreditations Department. Developers from those two departments introduced two different properties:

- `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 if a student has enough points to qualify for a Scholarship.

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



```
1 class Student {  
2     // ...  
3  
4     fun isPassing(): Boolean =  
5         calculatePointsFromPassedCourses() > 15  
6  
7     fun qualifiesForScholarship(): Boolean =  
8         calculatePointsFromPassedCourses() > 30  
9  
10    private fun calculatePointsFromPassedCourses(): Int {  
11        // ...  
12    }  
13}
```

Then, original rules change and the dean decides that less important courses should not qualify for scholarship points calculation. A developer who was sent to introduce this change checked function `qualifiesForScholarship`, finds out that it calls the private method `calculatePointsFromPassedCourses` and changes it to skip courses that do not qualify. Unintentionally, that developer changed the behavior of `isPassing` as well. Students who were supposed to pass, got informed that they failed the semester. You can imagine their reaction.

It is true that we could easily prevent such situation if we would have unit tests (Item 10: Write unit tests), but let's skip this aspect for now.

The developer might check where else the function is used. Although the problem is that this developer didn't expect that this private function was used by another property with a totally different responsibility. Private functions are rarely used just by more than one function.

This problem, in general, is that it is easy to couple responsibilities located very close (in the same class/file). A simple solution would be to extract these responsibilities into separate classes. We might

have separate classes `StudentIsPassingValidator` and `StudentQualifiesForScholarshipValidator`. Though in Kotlin we don't need to use such heavy artillery (see more at *Chapter 4: Design abstractions*). We can just define `qualifiesForScholarship` and `calculatePointsFromPassedCourses` as extension functions on `Student` located in separate modules: one over which Scholarships Department is responsible, and another over which Accreditations Department is responsible.

```
1 // scholarship module
2 fun Student.qualifiesForScholarship(): Boolean {
3     /*...*/
4 }
5
6 // accreditations module
7 fun Student.calculatePointsFromPassedCourses(): Boolean {
8     /*...*/
9 }
```

What about extracting a function for calculating results? We can do it, but it cannot be a private function used as a helper for both 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, each for every department.

Both options are safe.

The *Single Responsibility Principle* teaches us two things:

1. Knowledge coming from two different sources (here two different departments) is very likely to change independently, and we should rather treat it as a different knowledge.

2. We should separate different knowledge because otherwise, it is tempting to reuse parts 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 a bunch of elements has similar parts and it is likely that we will need to change it for all instances, extract it and save time on searching through the project and update many instances. On the other hand, protect yourself from unintentional modifications by separating parts that are coming from different sources. Often it's even more 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 looking 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 if something should be extracted or not. This is why it is an art to designing information systems well. It 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. Those might be mathematical operations, collection processing, or any other common behavior. Sometimes those algorithms can be long and complicated, like optimized sorting algorithms. There are also many simple examples though, like number coercion in a range:

```
1 val percent = when {
2     numberFromUser > 100 -> 100
3     numberFromUser < 0 -> 0
4     else -> numberFromUser
5 }
```

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

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

The advantages of extracting even short but repetitive algorithms are:

- **Programming is faster** because a single call is shorter than an algorithm (list of steps).
- **They are named, so we need to know the concept by name instead of recognizing it by reading its implementation.** This is easier for developers who are familiar with the concept. This might be harder for new developers who are not familiar with these concepts, but it pays off to learn the names

of repetitive algorithms. Once they learn it, they will profit from that in the future.

- **We eliminate noise, so it is 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`. 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 implemented sorting has natural or descending order. Comments before an algorithm implementation are not really helpful either. Practice shows that developers do change code without updating comments, and over time we lose trust in them.
- **They can be optimized once, and we profit from this optimization everywhere we use those functions.**

Learn the standard library

Common algorithms are nearly always already defined by someone else. Most libraries are just collections of common algorithms. The most special among them is the `stdlib` (standard library). It 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 reinvent the wheel time and time again. To see an example, take a look at this snippet taken from an open-source project:

```
1 override fun saveCallResult(item: SourceResponse) {  
2     var sourceList = ArrayList<SourceEntity>()  
3     item.sources.forEach {  
4         var sourceEntity = SourceEntity()  
5         sourceEntity.id = it.id  
6         sourceEntity.category = it.category  
7         sourceEntity.country = it.country  
8         sourceEntity.description = it.description  
9         sourceList.add(sourceEntity)  
10    }  
11    db.insertSources(sourceList)  
12 }
```

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, and instead, we should use a factory method or a primary constructor (*Chapter 5: Objects creation*). If for a 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:

```
1 override fun saveCallResult(item: SourceResponse) {  
2     val sourceEntries = item.sources.map(::sourceToEntry)  
3     db.insertSources(sourceEntries)  
4 }  
5  
6 private fun sourceToEntry(source: Source) = SourceEntity()  
7     .apply {  
8         id = source.id  
9         category = source.category  
10        country = source.country  
11        description = source.description  
12    }
```

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 a product of numbers in a collection? It is a well-known abstraction, and so it is good to define it as a universal utility function:

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

You don't need to wait for more than one use. It is a well-known mathematical concept and its name should be clear for developers. Maybe another developer will need to use it later 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, and so should be considered as a cost. We should be aware not to 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, property delegates and ending up on classes. Although extension functions are a really good choice:

- Functions do not hold state, and so they are perfect to represent behavior. Especially if it has no side-effects.
- Compared to 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 methods on objects, 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")`. It is because when you place a dot after `"Text"` you'll see as suggestions all 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 are calling a method, it is easy to confuse a top-level function with a method from the class or superclass, and their expected behavior is very different. Top-level extension functions do not have this problem, because they need to 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 property delegation to extract common property patterns

Edu

One new feature introduced by Kotlin that supports code reuse is property delegation. It gives us a universal way to extract common property behavior. One important example is the lazy property - a property that needs to be initialized on demand during its first use. Such a pattern is really popular, and in languages that do not support its extraction (like Java or JavaScript), it is implemented every time it is needed. In Kotlin, it can easily be extracted using property delegation. In the Kotlin stdlib, we can find the function `lazy` that returns a property delegate that implements the lazy property pattern:

```
1 val value by lazy { createValue() }
```

This is not the only repetitive property pattern. Another important example is the observable property - a property that does something whenever it is changed. For instance, let's say that you have a list adapter drawing a list. Whenever data change inside it, we need to redraw changed items. Or you might need to log all changes of a property. Both cases can be implemented using `observable` from stdlib:

```
1 var items: List<Item> by
2     Delegates.observable(listOf()) { _, _, _ ->
3         notifyDataSetChanged()
4     }
5
6 var key: String? by
7     Delegates.observable(null) { _, old, new ->
8         Log.e("key changed from $old to $new")
9     }
```

The lazy and observable delegates are not special from the language's point of view²⁸. They can be extracted thanks to a more general property delegation mechanism that can be used to extract many other patterns as well. Some good examples are View and Resource Binding, Dependency Injection (formally Service Location), or Data Binding. Many of these patterns require annotation processing in Java, however Kotlin allows you to replace them with easy, type-safe property delegation.

```
1 // View and resource binding example in Android
2 private val button: Button by bindView(R.id.button)
3 private val textSize by bindDimension(R.dimen.font_size)
4 private val doctor: Doctor by argExtra(DOCTOR_ARG)
5
6 // Dependency Injection using Koin
7 private val presenter: MainPresenter by inject()
8 private val repository: NetworkRepository by inject()
9 private val vm: MainViewModel by viewModel()
10
11 // Data binding
12 private val port by bindConfiguration("port")
13 private val token: String by preferences.bind(TOKEN_KEY)
```

To understand how this is possible and how we can extract other common behavior using property delegation, let's start from a very simple property delegate. Let's say that we need to track how some properties are used, and for that, we define custom getters and setters that log their changes:

²⁸Unlike in Swift or Scala, where lazy has built-in support.

```
1 var token: String? = null
2     get() {
3         print("token returned value $field")
4         return field
5     }
6     set(value) {
7         print("token changed from $field to $value")
8         field = value
9     }
10
11 var attempts: Int = 0
12     get() {
13         print("attempts returned value $field")
14         return field
15     }
16     set(value) {
17         print("attempts changed from $field to $value")
18         field = value
19     }
```

Even though their types are different, the behavior of those two properties is nearly identical. It seems like a repeatable pattern that might be needed more often in our project. This behavior can be extracted using property delegation. Delegation is based on the idea that a property is defined by its accessors - the getter in a `val`, and the getter and setter in a `var` - and those methods can be delegated into methods of another object. The getter will be delegated to the `getValue` function, and the setter to the `setValue` function. We then place such an object on the right side of the `by` keyword. To keep exactly the same property behaviors as in the example above, we can create the following delegate:

```
1 var token: String? by LoggingProperty(null)
2 var attempts: Int by LoggingProperty(0)
3
4 private class LoggingProperty<T>(var value: T) {
5     operator fun getValue(
6         thisRef: Any?,
7         prop: KProperty<*>
8     ): T {
9         print("${prop.name} returned value $value")
10        return value
11    }
12
13    operator fun setValue(
14        thisRef: Any?,
15        prop: KProperty<*>,
16        newValue: T
17    ) {
18        val name = prop.name
19        print("$name changed from $value to $newValue")
20        value = newValue
21    }
22 }
```

To fully understand how property delegation works, take a look at what `by` is compiled to. The above `token` property will be compiled to something similar to the following code²⁹:

²⁹When property used at top-level, instead of `this` there will be `null`.

```
1 @JvmField
2 private val `token$delegate` =
3     LoggingProperty<String?>(null)
4 var token: String?
5     get() = `token$delegate`.getValue(this, ::token)
6     set(value) {
7         `token$delegate`.setValue(this, ::token, value)
8     }
```

As you can see, `getValue` and `setValue` methods operate not only on the value, but they also receive a bounded reference to the property, as well as a context (`this`). The reference to the property is most often used to get its name, and sometimes to get information about annotations. Context gives us information about where the function is used.

When we have multiple `getValue` and `setValue` methods but with different context types, different ones will be chosen in different situations. This fact can be used in clever ways. For instance, we might need a delegate that can be used in different kinds of views, but with each of them, it should behave differently based on what is offered by the context:

```
1 class SwipeRefreshBinderDelegate(val id: Int) {
2     private var cache: SwipeRefreshLayout? = null
3
4     operator fun getValue(
5         activity: Activity,
6         prop: KProperty<*>
7     ): SwipeRefreshLayout {
8         return cache ?: activity
9             .findViewById<SwipeRefreshLayout>(id)
10            .also { cache = it }
11    }
12
13    operator fun getValue(
```

```
14     fragment: Fragment,
15     prop: KProperty<*>
16 ): SwipeRefreshLayout {
17     return cache ?: fragment.view
18         .findViewById<SwipeRefreshLayout>(id)
19         .also { cache = it }
20     }
21 }
```

To make it possible to use an object as a property delegate, all it needs is `getValue` operator for `val`, and `getValue` and `setValue` operators for `var`. This operation can be a member function, but it can be also an extension. For instance, `Map<String, >` can be used as a property delegate:

```
1 val map: Map<String, Any> = mapOf(
2     "name" to "Marcin",
3     "kotlinProgrammer" to true
4 )
5 val name by map
6 print(name) // Marcin
```

It is possible thanks to the fact that there is the following extension function in the Kotlin stdlib:

```
1 inline operator fun <V, V1 : V> Map<in String, V>
2     .getValue(thisRef: Any?, property: KProperty<*>): V1 =
3     getOrImplicitDefault(property.name) as V1
```

Kotlin standard library has some property delegates that we should know. Those are:

- `lazy`
- `Delegates.observable`

- `Delegates.vetoable`
- `Delegates.notNull`

It is worth to know them, and if you notice a common pattern around properties in your project, remember that you can make your own property delegate.

Summary

Property delegates have full control over properties and have nearly all the information about their context. This feature can be used to extract practically any property behavior. `lazy` and `observable` are just two examples from the standard library. Property delegation is a universal way to extract property patterns, and as practice shows, there are a variety of such patterns. It is a powerful and generic feature that every Kotlin developer should have in their toolbox. When we know it, we have more options for common pattern extraction or to define a better API.

Item 22: Use generics when implementing common algorithms

Edu

Similarly, 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 (so having type parameters) are called generic functions³⁰. One known example is the `filter` function from stdlib that has type parameter T:

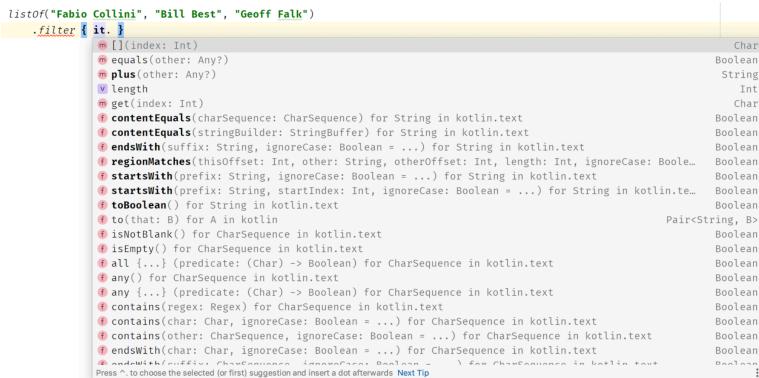
```
1 inline fun <T> Iterable<T>.filter(  
2     predicate: (T) -> Boolean  
3 ): List<T> {  
4     val destination = ArrayList<T>()  
5     for (element in this) {  
6         if (predicate(element)) {  
7             destination.add(element)  
8         }  
9     }  
10    return destination  
11 }
```

Type parameters are useful to the compiler since they allow it to check and correctly infer types a bit further, what makes our programs safer and programming more pleasurable for developers³¹. For instance, when we use `filter`, inside the lambda expression, the compiler knows that an argument is of the same type as the type

³⁰For functions, we define type parameters in angle brackets between `fun` keyword and function name, for classes or interfaces we define them in angle brackets after their name

³¹Notice that all that profits 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).

of elements in the collection, so it protects us from using something illegal and the IDE can give us useful suggestions.



Generics were primarily introduced to classes and interfaces to allow the creation of collections with only concrete types, like `List<String>` or `Set<User>`. Those types are lost during compilation but when we are developing, the compiler forces us to pass only elements of the correct type. For instance `Int` when we add to `MutableList<Int>`. Also, thanks to them, the compiler knows that the returned type is `User` when we get an element from `Set<User>`. This way 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 24: Consider variance 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 supertype after a colon. This type can include previous type parameters:

```
1 fun <T : Comparable<T>> Iterable<T>.sorted(): List<T> {
2     /*...*/
3 }
4
5 fun <T, C : MutableCollection<in T>>
6 Iterable<T>.toCollection(destination: C): C {
7     /*...*/
8 }
9
10 class ListAdapter<T: ItemAdaper>(/*...*/) { /*...*/ }
```

One important result of having a constraint is that instances of this type can use all the methods this type offers. This way 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 constraint 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-nullable type:

```
1 inline fun <T, R : Any> Iterable<T>.mapNotNull(
2     transform: (T) -> R?
3 ): List<R> {
4     return mapNotNullTo(ArrayList<R>(), transform)
5 }
```

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

```
1 fun <T: Animal> pet(animal: T) where T: GoodTempered {  
2     /*...*/  
3 }  
4  
5 // OR  
6  
7 fun <T> pet(animal: T) where T: Animal, T: GoodTempered {  
8     /*...*/  
9 }
```

Summary

Type parameters are an important part of Kotlin typing 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 methods offered by this type.

Item 23: Avoid shadowing type parameters

It is possible to define property and parameters with the same name due to shadowing. Local parameter shadows outer scope property. There is no warning because such a situation is not uncommon and is rather visible for developers:

```
1 class Forest(val name: String) {  
2  
3     fun addTree(name: String) {  
4         // ...  
5     }  
6 }
```

On the other hand, the same can happen when we shadow class type parameter with a function type parameter. Such a situation is less visible and can lead to serious problems. This mistake is often done by developers not understanding well how generics work.

```
1 interface Tree  
2 class Birch: Tree  
3 class Spruce: Tree  
4  
5 class Forest<T: Tree> {  
6  
7     fun <T: Tree> addTree(tree: T) {  
8         // ...  
9     }  
10 }
```

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

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

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

```
1 class Forest<T: Tree> {
2
3     fun addTree(tree: T) {
4         // ...
5     }
6 }
7
8 // Usage
9 val forest = Forest<Birch>()
10 forest.addTree(Birch())
11 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 the other type parameter:

```
1 class Forest<T: Tree> {
2
3     fun <ST: T> addTree(tree: ST) {
4         // ...
5     }
6 }
```

Summary

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

Item 24: Consider variance for generic types

Edu

Let's say that we have the following generic class:

```
1 class Cup<T>
```

Type parameter `T` in the above declaration does not have any variance modifier (`out` or `in`) and by default, it is invariant. It means that there is no relation between any two types generated by this generic class. For instance, there is no relation between `Cup<Int>` and `Cup<Number>`, `Cup<Any>` or `Cup<Nothing>`.

```
1 fun main() {  
2     val anys: Cup<Any> = Cup<Int>() // Error: Type mismatch  
3     val nothings: Cup<Nothing> = Cup<Int>() // Error  
4 }
```

If we need such a relation, then we should use variance modifiers: `out` or `in`. `out` makes type parameter covariant. It means that when `A` is a subtype of `B`, and `Cup` is covariant (`out` modifier), then type `Cup<A>` is a subtype of `Cup`:

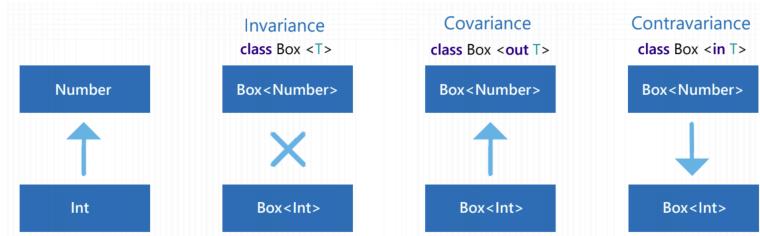
```
1 class Cup<out T>  
2 open class Dog  
3 class Puppy: Dog()  
4  
5 fun main(args: Array<String>) {  
6     val b: Cup<Dog> = Cup<Puppy>() // OK  
7     val a: Cup<Puppy> = Cup<Dog>() // Error  
8  
9     val anys: Cup<Any> = Cup<Int>() // OK  
10    val nothings: Cup<Nothing> = Cup<Int>() // Error  
11 }
```

The opposite effect can be achieved using `in` modifier, which makes type parameter contravariant. It means that when A is a subtype of B, and Cup is contravariant, then Cup<A> is a supertype of Cup:

```

1 class Cup<in T>
2 open class Dog
3 class Puppy(): Dog()
4
5 fun main(args: Array<String>) {
6     val b: Cup<Dog> = Cup<Puppy>() // Error
7     val a: Cup<Puppy> = Cup<Dog>() // OK
8
9     val anys: Cup<Any> = Cup<Int>() // Error
10    val nothings: Cup<Nothing> = Cup<Int>() // OK
11 }
```

Those variance modifiers are illustrated in the below diagram:



Function types

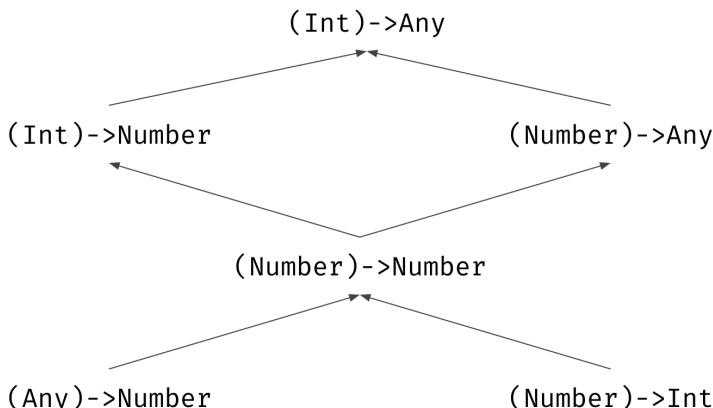
In function types (explained deeply in Item 35: Consider defining a DSL for complex object creation) there are relations between function types with different expected types of parameters or return types. To see it practically, think of a function that expects as argument a function accepting an `Int` and returning `Any`:

```
1 fun printProcessedNumber(transition: (Int)->Any) {  
2     print(transition(42))  
3 }
```

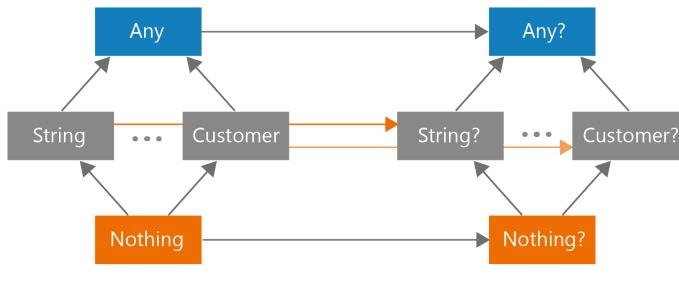
Based on its definition, such a function can accept a function of type `(Int)->Any`, but it would also work with: `(Int)->Number`, `(Number)->Any`, `(Number)->Number`, `(Any)->Number`, `(Number)->Int`, etc.

```
1 val inttoDouble: (Int) -> Number = { it.toDouble() }  
2 val numberAsText: (Number) -> Any = { it.toShort() }  
3 val identity: (Number) -> Number = { it }  
4 val numberToInt: (Number) -> Int = { it.toInt() }  
5 val numberHash: (Any) -> Number = { it.hashCode() }  
6 printProcessedNumber(inttoDouble)  
7 printProcessedNumber(numberAsText)  
8 printProcessedNumber(identity)  
9 printProcessedNumber(numberToInt)  
10 printProcessedNumber(numberHash)
```

It is because between those all types there is the following relation:



Notice that when we go down in this hierarchy, the parameter type moves towards types that are higher in the typing system hierarchy, and the return type moves toward types that are lower.



Kotlin type hierarchy

It is no coincidence. All parameter types in Kotlin function types are contravariant, as the name of this variance modifier `in` suggests. All return types in Kotlin function types are covariant, as the name of this variance modifier `out` suggests

$$(T_{1\text{in}}, T_{2\text{in}}) \rightarrow T_{\text{out}}$$

↙ ↑
 in out
 (contravariance) (covariance)

This fact supports us when we use function types, but it is not the only popular Kotlin type with variance modifiers. A more popular one is `List` which is covariant in Kotlin (`out` modifier). Unlike `MutableList` which is invariant (no variance modifier). To understand why we need to understand the safety of variance

modifiers.

The safety of variance modifiers

In Java, arrays are covariant. Many sources state that the reason behind this decision was to make it possible to create functions, like `sort`, that makes generic operations on arrays of every type. But there is a big problem with this decision. To understand it, let's analyze following valid operations, which produce no compilation time error, but instead throws runtime error:

```
1 // Java
2 Integer[] numbers = {1, 4, 2, 1};
3 Object[] objects = numbers;
4 objects[2] = "B"; // Runtime error: ArrayStoreException
```

As you can see, casting `numbers` to `Object[]` didn't change the actual type used inside the structure (it is still `Integer`), so when we try to assign a value of type `String` to this array, then an error occurs. This is clearly a Java flaw, and Kotlin protects us from that by making `Array` (as well as `IntArray`, `CharArray`, etc.) invariant (so upcasting from `Array<Int>` to `Array<Any>` is not possible).

To understand what went wrong here, we should first realize that when a parameter type is expected, we can pass any subtype of this type as well. Therefore when we pass an argument we can do implicit upcasting.

```
1 open class Dog
2 class Puppy: Dog()
3 class Hound: Dog()
4
5 fun takeDog(dog: Dog) {}
6
7 takeDog(Dog())
8 takeDog(Puppy())
9 takeDog(Hound())
```

This does not get along with covariance. If a covariant type parameter (`out` modifier) was present at in-position (for instance a type of a parameter), by connecting covariance and up-casting, we would be able to pass any type we want. Clearly, this wouldn't be safe since `value` is typed to a very concrete type and so when it is typed to `Dog` it cannot hold `String`.

```
1 class Box<out T> {
2     private var value: T? = null
3
4     // Illegal in Kotlin
5     fun set(value: T) {
6         this.value = value
7     }
8
9     fun get(): T = value ?: error("Value not set")
10 }
11
12 val puppyBox = Box<Puppy>()
13 val dogBox: Box<Dog> = puppyBox
14 dogBox.set(Hound()) // But I have a place for a Puppy
15
16 val dogHouse = Box<Dog>()
17 val box: Box<Any> = dogHouse
18 box.set("Some string") // But I have a place for a Dog
19 box.set(42) // But I have a place for a Dog
```

Such a situation wouldn't be safe, because after casting, the actual object stays the same and it is only treated differently by the typing system. We are trying to set `Int`, but we have only a place for a `Dog`. We would have an error if it was possible. This is why Kotlin prevents such a situation by prohibiting using covariant (`out` modifier) type parameters at a public in-position.

```
1 class Box<out T> {
2     var value: T? = null // Error
3
4     fun set(value: T) { // Error
5         this.value = value
6     }
7
8     fun get(): T = value ?: error("Value not set")
9 }
```

It is fine when we limit visibility to `private` because inside the object we cannot use covariance to up-cast object:

```
1 class Box<out T> {
2     private var value: T? = null
3
4     private set(value: T) {
5         this.value = value
6     }
7
8     fun get(): T = value ?: error("Value not set")
9 }
```

Covariance (`out` modifier) is perfectly safe with public out-positions and so they are not limited. This is why we use covariance (`out` modifier) for types that are produced or only exposed. It is often used for producers or immutable data holders.

One good example is a `List<T>` in which `T` is covariant in Kotlin. Thanks to that when a function expects `List<Any?>`, we can give any kind of list without any transformation needed. In `MutableList<T>`, `T` is invariant because it is used at in-position and it wouldn't be safe:

```
1 fun append(list: MutableList<Any>) {  
2     list.add(42)  
3 }  
4  
5 val strs = mutableListOf<String>("A", "B", "C")  
6 append(strs) // Illegal in Kotlin  
7 val str: String = strs[3]  
8 print(str)
```

Another good example is `Response` which can benefit a lot from using it. You can see how they can be used in the following snippet. Thanks to the variance modifiers, the following desired facts come true:

- When we expect `Response<T>`, response with any subtype of `T` will be accepted. For instance, when `Response<Any>` is expected, we accept `Response<Int>` as well as `Response<String>`.
- When we expect `Response<T1, T2>`, response with any subtype of `T1` and `T2` will be accepted.
- When we expect `Failure<T>`, failure with any subtype of `T` will be accepted. For instance, when `Failure<Number>` is expected, `Failure<Int>` or `Failure<Double>` are accepted. When `Failure<Any>` is expected, we accept `Failure<Int>` as well as `Failure<String>`.
- `Success` does not need to specify a type of potential error, and `Failure` does not need to specify a type of potential success value. This is achieved thanks to covariance and `Nothing` type.

```
1 sealed class Response<out R, out E>
2 class Success<out R>(val value: R): Response<R, Nothing>()
3 class Failure<out E>(val error: E): Response<Nothing, E>()
```

A similar problem as with covariance and public in-positions occurs when we are trying to use a contravariant type parameter (`in` modifier) as a public out-position (return type of a function or a property type). Out-positions also allow implicit up-casting.

```
1 open class Car
2 interface Boat
3 class Amphibious: Car(), Boat
4
5 fun getAmphibious(): Amphibious = Amphibious()
6
7 val car: Car = getAmphibious()
8 val boat: Boat = getAmphibious()
```

This fact does not get along with contravariance (`in` modifier). They both can be used again to move from any box to expect anything else:

```
1 class Box<in T>(
2     // Illegal in Kotlin
3     val value: T
4 )
5
6 val garage: Box<Car> = Box(Car())
7 val amphibiousSpot: Box<Amphibious> = garage
8 val boat: Boat = garage.value // But I only have a Car
9
10 val noSpot: Box<Nothing> = Box<Car>(Car())
11 val boat: Nothing = noSpot.value
12 // I cannot produce Nothing!
```

To prevent such a situation, Kotlin prohibits using contravariant (`in` modifier) type parameters at public out-positions:

```
1 class Box<in T> {
2     var value: T? = null // Error
3
4     fun set(value: T) {
5         this.value = value
6     }
7
8     fun get(): T = value // Error
9         ?: error("Value not set")
10 }
```

Again, it is fine when those elements are private:

```
1 class Box<in T> {
2     private var value: T? = null
3
4     fun set(value: T) {
5         this.value = value
6     }
7
8     private fun get(): T = value
9         ?: error("Value not set")
10 }
```

This way we use contravariance (`in` modifier) for type parameters that are only consumed or accepted. One known example is `kotlin.coroutines.Continuation`:

```
1 public interface Continuation<in T> {  
2     public val context: CoroutineContext  
3     public fun resumeWith(result: Result<T>)  
4 }
```

Variance modifier positions

Variance modifiers can be used in two positions. The first one, the declaration-side, is more common. It is a modifier on the class or interface declaration. It will affect all the places where the class or interface is used.

```
1 // Declaration-side variance modifier  
2 class Box<out T>(val value: T)  
3 val boxStr: Box<String> = Box("Str")  
4 val boxAny: Box<Any> = boxStr
```

The other one is the use-site, which is a variance modifier for a particular variable.

```
1 class Box<T>(val value: T)  
2 val boxStr: Box<String> = Box("Str")  
3 // Use-side variance modifier  
4 val boxAny: Box<out Any> = boxStr
```

We use use-site variance when for some reason we cannot provide variance modifiers for all instances, and yet you need it for one variable. For instance, `MutableList` cannot have `in` modifier because then it wouldn't allow returning elements (as described in the next section), but for a single parameter type we can make its type contravariant (`in` modifier) to allow any collections that can accept some type:

```
1 interface Dog
2 interface Cutie
3 data class Puppy(val name: String): Dog, Cutie
4 data class Hound(val name: String): Dog
5 data class Cat(val name: String): Cutie
6
7 fun fillWithPuppies(list: MutableList<in Puppy>) {
8     list.add(Puppy("Jim"))
9     list.add(Puppy("Beam"))
10 }
11
12 val dogs = mutableListOf<Dog>(Hound("Pluto"))
13 fillWithPuppies(dogs)
14 println(dogs)
15 // [Hound(name=Pluto), Puppy(name=Jim), Puppy(name=Beam)]
16
17 val animals = mutableListOf<Cutie>(Cat("Felix"))
18 fillWithPuppies(animals)
19 println(animals)
20 // [Cat(name=Felix), Puppy(name=Jim), Puppy(name=Beam)]
```

Notice that when we use variance modifiers, some positions are limited. When we have `MutableList<out T>`, we can use `get` to get elements and we receive an instance typed as `T`, but we cannot use `set` because it expects us to pass an argument of type `Nothing`. It is because list with any subtype of `T` might be passed there including the subtype of every type that is `Nothing`. When we use `MutableList<in T>`, we can use both `get` and `set`, but when we use `get`, the returned type is `Any?` because there might be a list with any supertype of `T` including the supertype of every type that is `Any?`. Therefore, we can freely use `out` when we only read from a generic object, and `in` when we only modify that generic object.

Summary

Kotlin has powerful generics that support constraints and also allow to have a relation between generic types with different type arguments declared both on declaration-side as well as on use-side. This fact gives us great support when we operate on generic objects. We have the following type modifiers:

- The default variance behavior of a type parameter is invariance. If in `Cup<T>`, type parameter `T` is invariant and `A` is a subtype of `B` then there is no relation between `Cup<A>` and `Cup`.
- `out` modifier makes type parameter covariant. If in `Cup<T>`, type parameter `T` is covariant and `A` is a subtype of `B`, then `Cup<A>` is a subtype of `Cup`. Covariant types can be used at `out`-positions.
- `in` makes type parameter contravariant. If in `Cup<T>`, type parameter `T` is contravariant and `A` is a subtype of `B`, then `Cup` is a subtype of `Cup<A>`. Contravariant types can be used at `in`-positions.

In Kotlin:

- Type parameter of `List` and `Set` are covariant (`out` modifier) so for instance, we can pass any list where `List<Any>` is expected. Also, the type parameter representing value type in `Map` is covariant (`out` modifier). Type parameters of `Array`, `MutableList`, `MutableSet`, `MutableMap` are invariant (no variance modifier).
- In function types parameter types are contravariant (`in` modifier) and return type is covariant (`out` modifier).
- We use covariance (`out` modifier) for types that are only returned (produced or exposed).
- We use contravariance (`in` modifier) for types that are only accepted.

Item 25: Reuse between different platforms by extracting common modules

Basics

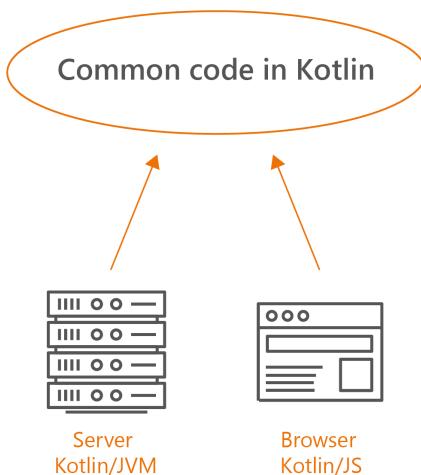
Companies rarely write applications only for just a single platform³². They would rather develop a product for two or more platforms, and their products, nowadays, often rely on several applications running on different platforms. Think of client and server applications communicating through network 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 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, those products need a backend application (also called server-side). On websites, JavaScript is the 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 like Ktor. This is why many backend projects migrate from Java to Kotlin. A great part of Kotlin is that it can

³²In Kotlin, we view the JVM, Android, JavaScript, iOS, Linux, Windows, Mac and even embedded systems like STM32 as separate platforms.

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. What is even better, we can have parts that compile both to JVM bytecode and to JavaScript. Those are shared parts. We can place there, for instance, universal tools, API endpoint definitions, common abstractions, etc. that we might want to reuse.



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.

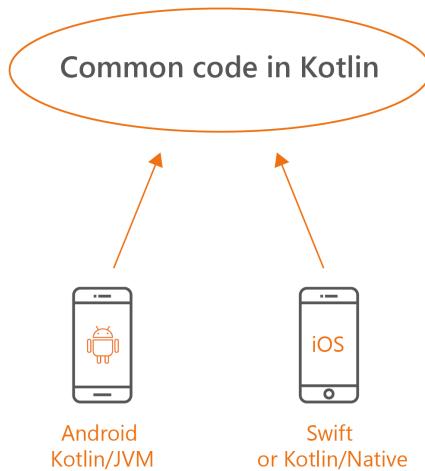
Using Kotlin multiplatform capabilities, we can implement this logic only once and reuse it between those two platforms. We can make a common module and implement business logic there. Business logic should be independent of frameworks and platforms anyway (Clean Architecture). Such common logic can be written in pure Kotlin or using other common modules, and it can then be used on different platforms.

In Android, it can be used directly since Android is built the same way using Gradle. The experience is similar to having those common parts in our Android project.

For iOS, we compile these common parts to an Objective-C framework using Kotlin/Native which is compiled using LLVM³³ into native code³⁴. We can then use it from Swift in Xcode or AppCode. Alternatively, we can implement our whole application using Kotlin/Native.

³³Like Swift or Rust.

³⁴Native 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.



Libraries

Defining common modules is also a powerful tool for libraries. In particular, those that do not depend strongly on platform can easily be moved to a common module and allow users to use them from all languages running on the JVM, JavaScript or natively (so from Java, Scala, JavaScript, CoffeeScript, TypeScript, C, Objective-C, Swift, Python, C#, etc.).

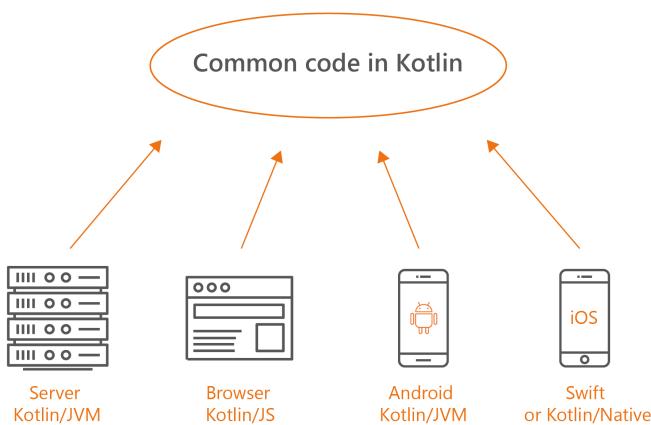
All together

We can use all those platforms together. Using Kotlin, we can develop for nearly all kinds of popular devices and platforms, and reuse code between them however we want. 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 Mac OS 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 tool to eliminate redundancy and to reuse common logic and common algorithms.

Chapter 4: Abstraction design

Abstraction is one of the most important concepts of the programming world. In OOP (Object-Oriented Programming) abstraction is one of 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 composition³⁵. As you can see, we treat abstraction seriously. Although 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 the interface. It is an abstraction of a class because it expresses only a subset of traits. Concretely, it is a set of methods and properties.

³⁵Category Theory for Programmers by Bartosz Milewski



Reality

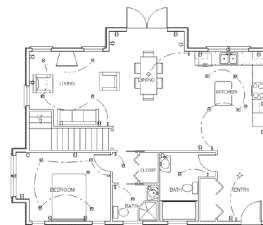


Abstraction

There is no single abstraction for every instance. There are many. In terms of objects, it can be expressed by many interfaces implemented or by multiple superclasses. It is an inseparable feature of abstraction that it decides what should be hidden and what should be exposed.



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 some String, it is easy to forget that it is a complex object where each character is represented on 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 method `maxOf` that would return the biggest of two numbers:

```
1 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:

```
1 val biggest = if (x > y) x else y
2
3 val height =
4     if (minHeight > calculatedHeight) minHeight
5     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 a 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:

```
1 // Java
2 List<String> names = new ArrayList<>();
3 for (User user : users) {
4     names.add(user.getName());
5 }
```

In Kotlin, since the beginning we have been able to express it using a simple function:

```
1 val names = users.map { it.name }
```

Lazy property initialization pattern still cannot be expressed in Java. In Kotlin, we use a property delegate instead:

```
1 val connection by lazy { makeConnection() }
```

Who knows how many other concepts are there, 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 patterns³⁶. 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 in terms of the abstractions directly.

Car metaphor

Many things happen when you drive a car. It requires the co-ordinated 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 interface—the steering wheel, gear shifter, and pedals—to operate the vehicle. Everything under the hood can change. A mechanic can change from petrol to natural gas, and then to diesel, without us even knowing about it. As cars introduce more and more electronic elements and special systems, the interface remains the same for the most part. With such changes under the hood, the car's performance would likely also change; however, we are able to operate the car regardless.

A car has a well-defined interface. Despite all of the complex components, it is simple to use. The steering wheel represents an

³⁶Structure and Interpretation of Computer Programs by Hal Abelson and Gerald Jay Sussman with Julie Sussman

abstraction for left-right direction change, the gear shifter is an abstraction for forward-backward direction change, the gas pedal 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 car construction. They only need to understand how to drive it. Similarly, creators or car enthusiasts can change everything in the car, and it is fine as long as driving stays the same. Remember this metaphor as we will refer to it throughout the 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 26: Each function should be written in terms of a single level of abstraction*, we will see how to use abstractions to organize the code. In *Item 27: Use abstraction to protect code against changes*, we will see how to use abstractions to give ourselves the freedom to change. Then we will spend the rest of this chapter on creating and using abstractions.

This is a pretty high-level chapter, and the rules presented here are a bit more abstract. Just after this chapter, we will cover some more concrete aspects of OOP design in *Chapter 5: Object creation* and *Chapter 6: Class design*. They will dive into deeper aspects of class implementation and use, but they will both build on this chapter.

Item 26: Each function should be written in terms of a single level of abstraction

Not Kotlin-specific

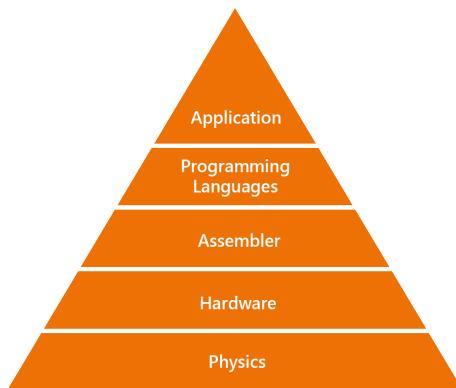
Basics

A computer is an extremely complex device, but we can work with it thanks to the fact that its complexity was split into different elements in distinct layers.

From a programmer's perspective, the lowest abstraction layer of a computer is hardware. Going up, since we generally write code for processors, the next interesting layer is a processor control commands. 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 the Assembly language is difficult, and it is absolutely unthinkable to build today's applications this way. To simplify programming, engineers introduced a compiler: a program that translates one language into another (generally a lower-level one). First compilers were written in the 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. These languages are used to write programs and applications. Later, the concepts of the abstract machine and interpreted languages were invented and it is hard to place languages like Java or JavaScript on this pyramid, but the general notion of abstraction layers stayed as an idea.

The big advantage of having well-separated layers is that when one operates on a specific layer, they can rely on lower levels working as expected, 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, they don't need to worry about

changing applications as long as creators adjust the upper layer - 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 were built upon layers in computer science. This is why computer scientists started distinguishing how high-level something is. The higher the level, the further from physics. In programming, we say that the higher level, the further from the processor. The higher the level, the fewer details we need to worry about. But you are trading this simplicity with a lack of control. In C, memory management is an important part of your job. In Java, the Garbage Collector handles it automatically for you, but optimizing memory usage is much harder.

Single Level of Abstraction principle

Just like computer science problems were 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 that 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 to make coffee. 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 could definitely implement all the necessary logic inside that unique function:

```
1 class CoffeeMachine {  
2  
3     fun makeCoffee() {  
4         // Declarations of hundreds of variables  
5         // Complex logic to coordinate everything  
6         // with many low-level optimizations  
7     }  
8 }
```

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 reading it we would constantly focus our attention on the details. It would also be hard to find anything. Just imagine that you are asked to make a small modification, like to modify the temperature of the water, and to do that, you would probably need to understand this whole function, and it would be absurdly hard. Our memory is limited and we do not want a programmer to waste time on the unnecessary details. This is why it is better to extract high-level steps as separate functions:

```
1  class CoffeeMachine {  
2  
3      fun makeCoffee() {  
4          boilWater()  
5          brewCoffee()  
6          pourCoffee()  
7          pourMilk()  
8      }  
9  
10     private fun boilWater() {  
11         // ...  
12     }  
13  
14     private fun brewCoffee() {  
15         // ...  
16     }  
17  
18     private fun pourCoffee() {  
19         // ...  
20     }  
21  
22     private fun pourMilk() {  
23         // ...  
24     }  
25 }
```

Now you can clearly see what the general flow of this function is. Those private functions are just like chapters in a book. Thanks to that, if you need to change something, you can jump directly where it is implemented. We just extracted higher-level procedures, which greatly simplified the comprehension of our first procedure. We made it readable, and if someone wanted to understand it at a lower level, they can just jump there and read it. By extracting very simple abstractions, we 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 responsibilities³⁷. If one of them is more complex, we should extract intermediary abstractions³⁸. As a result, we should achieve many small and readable functions, all localized at a single level of abstraction. At every level of abstraction we operate on abstract terms (methods and classes) and if you want to clarify them, you can always jump into their definition (in IntelliJ or Android Studio, holding the Ctrl key [Command on Mac] while you click on the function name will take you to the implementation). This way we lose nothing from extracting those functions, and make our code more readable.

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 and the difference is that it does not have milk. When parts of the process are extracted, we can now reuse them easily:

```
1 fun makeEspressoCoffee() {  
2     boilWater()  
3     brewCoffee()  
4     pourCoffee()  
5 }
```

It is also so that we can now unit test separately smaller functions like `boilWater` or `brewCoffee`, instead of more complex functions like `makeCoffee` or `makeEspressoCoffee`.

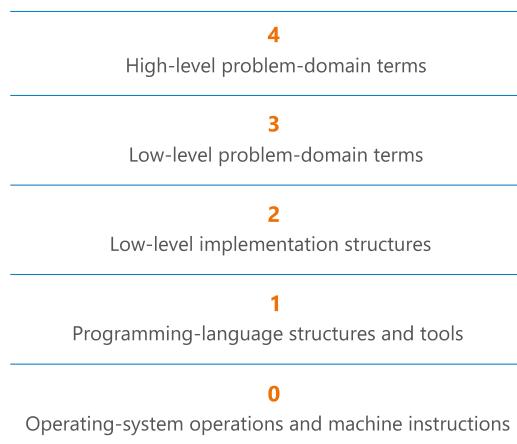
Abstraction levels in program architecture

The notion of layers of abstractions is also applicable to higher levels than functions. We separate abstraction to hide details of a

³⁷Clean Code by Robert Cecil Martin

³⁸Those might be functions like in this example, as well as classes or other kinds of abstractions. Differences will be shown in the next item, Item 27: Use abstraction to protect code against changes.

subsystem, allowing the separation of concerns to facilitate interoperability and platform independence. It means defining higher levels in problem-domain terms³⁹.



This notion is also important when we design modular systems. Separated modules are a strong separation that can hide layer-specific elements. When we write applications, the general understanding is that those modules that represent input or output (views in 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 layers⁴⁰.

We say that projects with well-separated layers stratified. In a well-stratified project, one can view the system at any single level and get a consistent view⁴¹. Stratification is generally desired in programs.

³⁹Code Complete by Steve McConnell, 2nd Edition, Section 34.6

⁴⁰Clean Architecture: A Craftsman's Guide to Software Structure and Design by Robert C. Martin, 1st Edition

⁴¹Code Complete by Steve McConnell, 2nd Edition, Section 5.2

Summary

Making separate abstraction layers is a popular concept used in programming. It helps us organize knowledge and hide details of a subsystem, allowing the separation of concerns to facilitate interoperability and platform independence. We separate abstractions in many ways, like functions, classes, modules. We should try not to make any of those layers too big. Smaller abstractions operating on a single layer are easier to understand. The general notion of abstraction level is that the closer 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 27: 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 do not only protect users from those details, but we also give ourselves the freedom to change this code later. Often without users even knowing about 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.

Thinking about the car metaphor mentioned before, car manufacturers and mechanics can change everything under the hood of the car, and as long as the operation remains the same, a user won't notice. This provides manufacturers the freedom to make more environment-friendly cars or to add more sensors to make cars safer.

In this item, we will see how different kinds of abstractions give us freedom by protecting us from a variety of changes. We will examine three practical cases, and in the end, discuss finding a balance in a number of abstractions. Let's start with the simplest kind of abstraction: constant value.

Constant

Literal constant values are rarely self-explanatory and they are especially problematic when they repeat in our code. Moving the values into constant properties not only assigns the value a meaningful name, but also helps us better manage when this constant

needs to be changed. Let's see a simple example with password validation:

```
1 fun isPasswordValid(text: String): Boolean {  
2     if(text.length < 7) return false  
3     //...  
4 }
```

The number 7 can be understood based on context, but it would be easier if it would be extracted into a constant:

```
1 const val MIN_PASSWORD_LENGTH = 7  
2  
3 fun isPasswordValid(text: String): Boolean {  
4     if(text.length < MIN_PASSWORD_LENGTH) return false  
5     //...  
6 }
```

With that, it is easier to modify the minimum password size. We don't need to understand validation logic, but 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:

```
1 val MAX_THREADS = 10
```

Once it is extracted, you can easily change it when 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 constant:

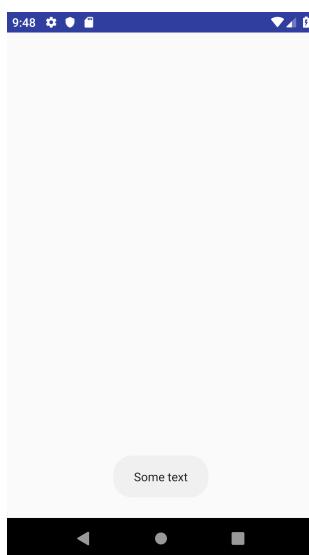
- Names it
- Helps us change its value in the future

We will see similar results for different kinds of abstractions as well.

Function

Imagine that you are developing an application and you noticed that you often need to display a toast message to a user. This is how you do it programmatically:

```
1  Toast.makeText(this, message, Toast.LENGTH_LONG).show()
```



Toast message in Android

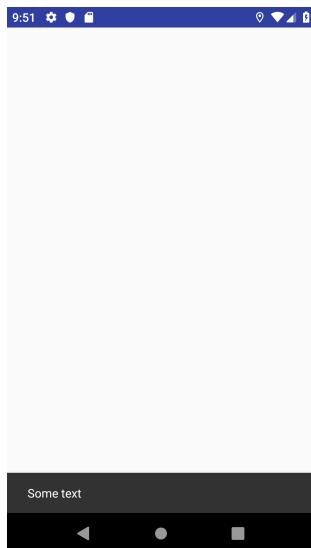
We can extract this common algorithm into a simple extension function for displaying toast:

```
1 fun Context.toast(  
2     message: String,  
3     duration: Int = Toast.LENGTH_LONG  
4 ) {  
5     Toast.makeText(this, message, duration).show()  
6 }  
7  
8 // Usage  
9 context.toast(message)  
10  
11 // Usage in Activity or subclasses of Context  
12 toast(message)
```

This change helped 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, was to change (what is rather unlikely). Though there are changes we are not prepared for.

What if we had to change the way we display messages to the user from toasts to snackbars (a different kind of message display)? A simple answer is that having this functionality extracted, we can just change the implementation inside this function and rename it.

```
1 fun Context.snackbar(  
2     message: String,  
3     length: Int = Toast.LENGTH_LONG  
4 ) {  
5     //...  
6 }
```



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 internally⁴². 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 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 be problematic as well:

⁴²When a function is a part of external API, we cannot easily adjust calls and so we are stuck with the old name for at least some time (Item 28: Specify API stability).

```
1 fun Context.snackbar(  
2     message: String,  
3     duration: Int = Snackbar.LENGTH_LONG  
4 ) {  
5     //...  
6 }
```

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 instead the fact that we want to display the message to a user. What we need is a more abstract method to display a message. Having that in mind, a programmer could hide toast display behind a higher-level function `showMessage`, which would be independent of the concept of toast:

```
1 fun Context.showMessage(  
2     message: String,  
3     duration: MessageLength = MessageLength.LONG  
4 ) {  
5     val toastDuration = when(duration) {  
6         SHORT -> Toast.LENGTH_SHORT  
7         LONG -> Toast.LENGTH_LONG  
8     }  
9     Toast.makeText(this, message, toastDuration).show()  
10 }  
11  
12 enum class MessageLength { SHORT, LONG }
```

The biggest change here is the name. Some developers might neglect the importance of this change saying that a name is just a label and it 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 this 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.

Class

Here is how we can abstract message display into a class:

```
1  class MessageDisplay(val context: Context) {  
2  
3      fun show(  
4          message: String,  
5          duration: MessageLength = MessageLength.LONG  
6      ) {  
7          val toastDuration = when(duration) {  
8              SHORT -> Toast.LENGTH_SHORT  
9              LONG -> Toast.LENGTH_LONG  
10         }  
11         Toast.makeText(context, message, toastDuration)  
12             .show()  
13     }  
14 }  
15  
16 enum class MessageLength { SHORT, LONG }  
17  
18 // Usage  
19 val messageDisplay = MessageDisplay(context)  
20 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. Using a dependency injection framework we can delegate the class creation:

```
1 @Inject lateinit var messageDisplay: MessageDisplay
```

Additionally, we can mock the class to test the functionality of other classes that depend on the specified class. This is possible because we can mock classes for testing purposes:

```
1 val messageDisplay: MessageDisplay = mockk()
```

Furthermore, one could add more methods to set up message display:

```
1 messageDisplay.setChristmasMode(true)
```

As you can see, the class gives us more freedom. But they still have their 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 one could serve a subclass instead. This abstraction is still strongly bound to this class though. To get more freedom we can make it even more abstract and hide this class behind an interface.

Interface

Reading the Kotlin standard library, you might notice that nearly everything is represented as an interface. Just take a look at a few examples:

- `listOf` function returns `List`, which is an interface. This is similar to other factory methods (we will explain them in Item 33, Consider factory methods instead of constructors).
- Collection processing functions are extension functions on `Iterable` or `Collection`, and return `List`, `Map`, etc. Those are all interfaces.

- Property delegates are hidden behind `ReadOnlyProperty` or `ReadWriteProperty` which are also interfaces. Actual classes are often private. Function `lazy` declares interface `Lazy` as its return type as well.

It is common practice for library creators to restrict inner class visibility 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 - Kotlin generally uses platform-specific native collections. This 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 can look like when we hide our class behind an interface:

```
1  interface MessageDisplay {  
2      fun show(  
3          message: String,  
4          duration: MessageLength = LONG  
5      )  
6  }  
7  
8  class ToastDisplay(val context: Context): MessageDisplay {  
9  
10     override fun show(  
11         message: String,
```

```
12     duration: MessageLength
13   ) {
14     val toastDuration = when(duration) {
15       SHORT -> Toast.LENGTH_SHORT
16       LONG -> Toast.LENGTH_LONG
17     }
18     Toast.makeText(context, message, toastDuration)
19     .show()
20   }
21 }
22
23 enum class MessageLength { SHORT, LONG }
```

In return, we 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 different implementations for each platform. For instance, in 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 any mocking library:

```
1 val messageDisplay: MessageDisplay = TestMessageDisplay()
```

Finally, the declaration is more decoupled from usage, and 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 next ID, and increment it whenever we need a new ID:

```
1 var nextId: Int = 0
2
3 // Usage
4
5 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.

Supposing that for now we accept this solution, we should protect ourselves from change by extracting ID creation into a function:

```
1 private var nextId: Int = 0
2 fun getNextId(): Int = nextId++
3
4 // Usage
5 val newId = getNextId()
```

Notice though that this solution only protects us from ID creation change. There are many changes that we are still prone to. The biggest one is the change of ID type. What if one day we need to keep ID as a String? Also notice that someone seeing 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 that and to let ourselves change ID type easily in the future, we might extract ID as a class:

```
1 data class Id(private val id: Int)  
2  
3 private var nextId: Int = 0  
4 fun getNextId(): Id = Id(nextId++)
```

Once again, it is clear that more abstractions give us more freedom, but also make definitions and usage harder to define and to understand.

Abstractions give freedom

We've presented a few common ways to introduce abstraction:

- Extracting constant
- Wrapping behavior into a function
- Wrapping function into a class
- Hiding a class behind an interface
- Wrapping universal objects into specialistic

We've seen how each of those gave us different kinds of freedom. Notice that there are many more tools available. Just to name a few:

- Using generic type parameters
- Extracting inner classes
- Restricting creation, for instance by forcing object creation via factory method⁴³

On the other hand, abstractions have their dark side. They give us freedom and split code, but often they can make code harder to understand and to modify. Let's talk about problems with abstractions.

⁴³More about it in Chapter 4: Objects creation

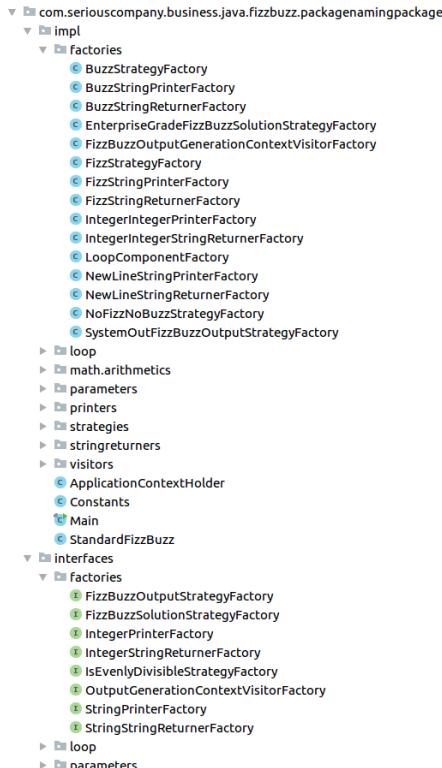
Problems with abstraction

Adding new abstractions requires readers of the code to learn or already be familiar with the specific concept. When we define another abstraction, it is another thing that needs to be understood in our project. Of course, it is less of a problem when we restrict abstractions visibility (Item 30: 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 abstraction is having this cost and we should not abstract everything by default.

We can infinitely extract abstractions, but soon this will make more harm than good. This fact was parodied in the FizzBuzz Enterprise Edition project⁴⁴ where authors showed that even for such a simple problem as Fizz Buzz⁴⁵, one can extract a ridiculous amount of abstractions making this problem extremely hard to use and understand. At the time of writing this book, 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, though on the other hand understanding what does this code do and how does it do it is extremely hard.

⁴⁴Link to the project: github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

⁴⁵The problem is defined as: For numbers 1 through 100 if the number is divisible by 3 print Fizz; if the number is divisible by 5 print Buzz; if the number is divisible by 3 and 5 (15) print FizzBuzz; else, print the number.



Part of FizzBuzz Enterprise Edition structure of classes. In the description of this project, you can find the 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.”

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.** One might use the `showMessage` function thinking that it still displays toast, and we might be surprised when it displays a snack bar. One seeing that unintended toast message is displayed 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 are the consequences of our actions.

To understand abstractions, examples are very helpful. Unit test or examples in the documentation that shows how an element can be used, make abstractions more real for us. For the same reason, I filled this book with concrete examples for most ideas I present. It is hard to understand abstract descriptions. It is also easy to misunderstand them.

Where is the 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 between, and where is it exactly, it 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, a separation between modules or parts is especially useful then.
- We care less about how hard 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. Although when it gets serious, change it as soon as possible.

Another thing that we need to constantly think about is what might change and what are the odds for each change. For instance, there is a very small chance that the API for 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? 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 is harder. They are something we need to learn and understand. It is also harder to understand the consequences when we use abstract structures. We need to understand both the importance and risk of using abstractions, and we need to search for a balance in every project. Having too many or too little abstractions would not be an ideal situation.

Item 28: Specify API stability

Life would be much harder if every car was 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 the API changes and developers get the update, they will need to manually update their code.** This point can be especially problematic when many elements depend on this API. Fixing its use or giving an alternative might be hard. Especially if our API was used by another developer in a part of our project we are not familiar with. If it is a public library, we cannot adjust these uses ourselves, but instead, our users have to make the changes. From a user's perspective, it is not a comfortable situation. Small changes in a library might require many changes in different parts of the codebase. When users are afraid of such changes, they stay on 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 spend. What's more, **they need to update knowledge that changed.** This is painful for them as well, so they avoid it. Not healthy either: outdated

knowledge can lead to security issues and learning changes the hard way.

On the other hand, designing a good API is very hard, and so creators want to make changes to improve it. The solution that we (the programming community) developed is that we specify API stability.

The simplest way is that creators should specify in the documentation to make it clear if an API or some of its parts 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), and in this system, we compose 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:

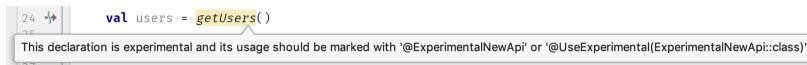
- MAJOR version when you make incompatible API changes.
- MINOR version when you add functionality in a backward-compatible manner.
- PATCH version when you make backward-compatible bug fixes.

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, and 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, and 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 it (by merging code into the main branch and releasing it), you can use the `Experimental` meta-annotation to warn them that they are not yet stable. It makes elements visible, but their use displays a warning or an error (depending on set level).

```
1 @Experimental(level = Experimental.Level.WARNING)
2 annotation class ExperimentalNewApi
3
4 @ExperimentalNewApi
5 suspend fun getUsers(): List<User> {
6     //...
7 }
```



We should expect that such elements might change at any moment. Again, don't worry to keep elements experimental for a long time. Doing that slows down adoption, but also helps us design good API for longer.

When we need to change something that is part of a stable API, to help users deal with this transition, we start with annotating this element with the `Deprecated` annotation:

```
1 @Deprecated("Use suspending getUsers instead")
2 fun getUsers(callback: (List<User>)->Unit) {
3     //...
4 }
```

Also, when there is a direct alternative, specify it using `ReplaceWith` to allow the IDE to make automatic transition:

```
1 @Deprecated("Use suspending getUsers instead",
2 ReplaceWith("getUsers()"))
3 fun getUsers(callback: (List<User>)->Unit) {
4     //...
5 }
```

An example from the stdlib:

```
1 @Deprecated("Use readBytes() overload without "+
2 "estimatedSize parameter",
3 ReplaceWith("readBytes()"))
4 public fun InputStream.readBytes(
5     estimatedSize: Int = DEFAULT_BUFFER_SIZE
6 ): ByteArray {
7     //...
8 }
```

Then we need to give users time to adjust. This should be a long time because users have other responsibilities than adjusting to new versions of libraries they use. In widely used APIs, this takes years. Finally after this time, in some 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 supposed to be stable. Such changes can be really painful for users. Correct communication between module or library creators and their users is important. We achieve that by using version names, documentation, and annotations. Also, each change in a stable API needs to follow a long process of deprecation.

Item 29: Consider wrapping external API

Not Kotlin-specific

It is risky to heavily use an API that might be unstable. Both when creators clarify that it is unstable, and when we do not trust those creators to keep it stable. Remembering that we need to adjust every use in case of inevitable API change, 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 gives us a lot of freedom and stability:

- We are not afraid of API changes because we would only need to change a single usage inside the wrapper.
- We can adjust the API to our project style and logic.
- We can replace it with a different library in case of some problems with this one.
- We can change the behavior of these objects if we need to (of course, do it responsibly).

There are also counterarguments to this approach:

- We need to define all those wrappers.
- Our internal API is internal, and developers need to learn it just for this project.
- There are no courses teaching how our internal API works. We should also not expect answers on Stack Overflow.

Knowing both sides, you need to decide which APIs should be wrapped. A good heuristics that tells us how stable a library is are version number and the number of users. Generally, the more users the library has, the more stable it is. Creators are more careful with changes when they know that their small change

might require corrections in many projects. The riskiest libraries are new ones with small popularity. Use them wisely and consider wrapping them into your own classes and functions to control them internally.

Item 30: Minimize elements visibility

When we design an API, there are many reasons why we prefer it as lean as possible. Let's name the most important reasons.

It is easier to learn and maintain a smaller interface. Understanding a class is easier when there are 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 to test.

When we want to make changes, it is way easier to expose something new, rather than to hide an existing element. All publicly visible elements are part of our public API, and they can be used externally. The longer an element is visible, the more external uses it will have. As such, changing these elements will be harder because they will require updating all usages. Restricting visibility would be even more of a challenge. If you do, you'll need to carefully consider each usage and provide an alternative. Giving an alternative might not be simple, especially if it was implemented by another developer. Finding out now what were the business requirements might be tough as well. If it is a public library, restricting some elements' visibility might make some users angry. They will need to adjust their implementation and they will face the same problems - they will need to implement alternative solutions probably years after code was developed. 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 on a class state that class 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 not knowing about our internal contract. Take a look at CounterSet from *Chapter 2*. We correctly

restricted the visibility of `elementsAdded` setter. Without it, someone might change it to any value from outside and we wouldn't be able to trust that this value really represents how many elements were added. Notice that only setter is private. This is a very useful trick.

```
1 class CounterSet<T>(
2     private val innerSet: MutableSet<T> = mutableSetOf\
3 ()
4 ) : MutableSet<T> by innerSet {
5
6     var elementsAdded: Int = 0
7     private set
8
9     override fun add(element: T): Boolean {
10         elementsAdded++
11         return innerSet.add(element)
12     }
13
14     override fun addAll(elements: Collection<T>): Boolean {
15         elementsAdded += elements.size
16         return innerSet.addAll(elements)
17     }
18 }
```

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 internal object state is especially important when we have properties depending on each other. For instance, in the below `mutableLazy` delegate implementation, we expect that if `initialized` is `true`, the value is initialized and it contains a value of type `T`. Whatever we do, setter of the `initialized` should not be exposed, because otherwise it cannot be trusted and that can lead to an ugly exception on a different property.

```
1  class MutableLazyHolder<T>(val initializer: () -> T) {  
2      private var value: Any? = Any()  
3      private var initialized = false  
4  
5      fun get(): T {  
6          if (!initialized) {  
7              value = initializer()  
8              initialized = true  
9          }  
10         return value as T  
11     }  
12  
13     fun set(value: T) {  
14         this.value = value  
15         initialized = true  
16     }  
17 }
```

It is easier to track how class changes when they have more restricted visibility. This makes the property state easier to understand. It is especially important when we are dealing with concurrency. State changes are a problem for parallel programming, and it is better to control and restrict it 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 have it hidden. This is why if there is no good reason to have less restrictive visibility type, it is a good practice to make the visibility of classes and elements as restrictive as possible. We do that using visibility modifiers.

For class members, these are 4 visibility modifiers we can use together with their behavior:

- `public` (default) - visible everywhere, for clients who 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 the module is not the same as package. In Kotlin it is defined as a set of Kotlin sources compiled together. It might mean:

- 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 your public elements that you don't want to expose to `internal`. If an element is designed for inheritance and it is only used in a class and subclasses, make it `protected`. If you use element only in the same file or class, make it `private`. This convention is supported by Kotlin as it suggests to restrict visibility to private if an element is used only locally:



This rule should not be applied to properties in the classes that were designed primarily to hold data (data model classes, DTO). If your server returns a user with an age, and you decided 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.

```
1 class User(  
2     val name: String,  
3     val surname: String,  
4     val age: Int  
5 )
```

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**. Visible elements constitute the public API, 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 something.
- 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 the API changes when they have more restricted visibility.

Item 31: Define contract with documentation

Think again about the function to display a message from *Item 27*:
Use abstraction to protect code against changes:

```
1 fun Context.showMessage(  
2     message: String,  
3     length: MessageLength = MessageLength.LONG  
4 ) {  
5     val toastLength = when(length) {  
6         SHORT -> Toast.LENGTH_SHORT  
7         LONG -> Toast.LENGTH_LONG  
8     }  
9     Toast.makeText(this, message, toastLength).show()  
10 }  
11  
12 enum class MessageLength { SHORT, LONG }
```

We extracted it to give ourselves the freedom to change how the message is displayed. However, it is not well documented. Another developer might read its code and assume that it always displays a toast. This is the opposite of what we wanted to achieve by naming it in a way not suggesting concrete message type. To make it clear it would be better to add a meaningful KDoc comment explaining what should be expected from this function.

```

1  /**
2  * Universal way for the project to display a short
3  * message to a user.
4  * @param message The text that should be shown to
5  * the user
6  * @param length How long to display the message.
7  */
8  fun Context.showMessage(
9      message: String,
10     duration: MessageLength = MessageLength.LONG
11 ) {
12     val toastDuration = when(duration) {
13         SHORT -> Toast.LENGTH_SHORT
14         LONG -> Toast.LENGTH_LONG
15     }
16     Toast.makeText(this, message, toastDuration).show()
17 }
18
19 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 so well known and interpretation is not clear enough:

```

1  /**
2  * Powerset returns a set of all subsets of the receiver
3  * including itself and the empty set
4  */
5  fun <T> Collection<T>.powerset(): Set<Set<T>> =
6      if (isEmpty()) setOf(emptySet())
7      else take(size - 1)
8          .powerset()
9          .let { it + it.map { it + last() } }
```

Notice that this description gives us some freedom. It does not

specify the order of those elements. As a user, we should not depend on how those elements are ordered. Implementation hidden behind this abstraction can be optimized without changing how this function looks from the outside:

```
1  /**
2  * Powerset returns a set of all subsets of the receiver
3  * including itself and empty set
4  */
5  fun <T> Collection<T>.powerset(): Set<Set<T>> =
6      powerset(this, setOf(setOf()))
7
8  private tailrec fun <T> powerset(
9      left: Collection<T>,
10     acc: Set<Set<T>>
11 ): Set<Set<T>> = when {
12     left.isEmpty() -> acc
13     else -> {
14         val head = left.first()
15         val tail = left.drop(1)
16         powerset(tail, acc + acc.map { it + head })
17     }
18 }
```

The general problem is that **when the behavior is not documented and the element name is not clear, developers will depend on current implementation instead of on the abstraction we intended to create.** We solve this problem by describing what behavior can be expected.

Contract

Whenever we describe some behavior, users treat it as a promise and based on that they adjust their expectations. We call all such expected behaviors a contract of an element. Just like in a real-life

contract another side expects us to honor it, here as well users will expect us to keep this contract once it is stable (*Item 28: 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, the contract gives freedom to change everything as long as the contract is satisfied. **Both users and creators depend on abstractions defined in the contract, and so they can work independently.** Everything will work perfectly fine as long as the contract is respected. This is a comfort and freedom for both sides.

What if we don't set a contract? Without users knowing what they can and cannot do, they'll depend on implementation details instead. A creator without knowing what users depend on would be either blocked or they would risk breaking users implementations. As you can see, it is important to specify a contract.

Defining a contract

How do we define a contract? There are various ways, including:

- Names - when a name is connected to some more general concept, we expect this element to be consistent with this concept. For instance, when you see `sum` method, you don't need to read its comment to know how it will behave. It is because the summation is a well defined mathematical concept.
- Comments and documentation - 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 additionally have set-up responsibilities in their documentation. When we see a function, information about return type and arguments types are very meaningful.

Do we need comments?

Looking at history, it is amazing to see how opinions in the community fluctuate. When Java was still young, there was a very popular concept of literate programming. It suggested explaining everything in comments⁴⁶. A decade later we can hear a very strong critique of comments and strong voices that we should omit comments and concentrate on writing readable code instead (I believe that the most influential book was the Clean Code by Robert C. Martin).

No extreme is healthy. I absolutely agree that we should first concentrate on writing readable code. Though what needs to be understood is that comments before elements (functions or classes) can describe it at a higher level, and set their contract. 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 they don't need any special description. We might, for instance, assume that product is a clear mathematical concept that is known by programmers, and leave it without any comment:

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

⁴⁶Read more about this concept in the book Literate Programming by Donald Knuth.

Obvious comments are a 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 the functionality can be inferred from the method's name and parameter type:

```
1 // Product of all numbers in a list
2 fun List<Int>.product() = fold(1) { acc, i -> acc * i }
```

I also agree that when we just need to organize our code, instead of comments in the implementation, we should extract a function. Take a look at the example below:

```
1 fun update() {
2     // Update users
3     for (user in users) {
4         user.update()
5     }
6
7     // Update books
8     for (book in books) {
9         updateBook(book)
10    }
11 }
```

Function `update` is clearly composed of extractable parts, and comment suggests that those parts can be described with a different explanation. Therefore it is better to extract those parts into separate abstractions like for instance methods, and their names are clear enough to explain what they mean (just like it in *Item 26: Each function should be written in terms of a single level of abstraction*).

```
1 fun update() {  
2     updateUsers()  
3     updateBooks()  
4 }  
5  
6 private fun updateBooks() {  
7     for (book in books) {  
8         updateBook(book)  
9     }  
10 }  
11  
12 private fun updateUsers() {  
13     for (user in users) {  
14         user.update()  
15     }  
16 }
```

Although 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 function `listOf`:

```
1 /**
2 * Returns a new read-only list of given elements.
3 * The returned list is serializable (JVM).
4 * @sample samples.collections.Collections.Lists.
5 readOnlyList
6 */
7 public fun <T> listOf(vararg elements: T): List<T> =  
8     if (elements.size > 0) elements.asList()  
9     else emptyList()
```

All it promises is that it returns `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

satisfactory 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.

KDoc format

When we document functions using comments, the official format in which we present that comment is called KDoc. All KDoc comments start with `/**` and end with `*/`, and internally all lines generally start with `*`. Descriptions there are written in KDoc markdown.

The structure of this 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. Those tags are used to reference an element to describe it.

Here are tags that are supported:

- `@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 a class which has the specified name. Used for properties defined on the primary constructor.
- `@throws <class>, @exception <class>` - Documents an exception which can be thrown by a method.
- `@sample <identifier>` - Embeds the body of the function with the specified qualified name 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 element being documented.
- @since - Specifies the version of the software in which the element being documented 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.

Both in descriptions and in texts describing tags we can link elements, concretely classes, methods, properties or parameters. Links are in square brackets or with double square brackets when we want to have different description than the name of the linked element.

```
1  /**
2  * This is an example descriptions linking to [element1],
3  * [com.package.SomeClass.element2] and
4  * [this element with custom description][element3]
5  */
```

All those 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 shortened description:

```
1  /**
2  * Immutable tree data structure.
3  *
4  * Class represents immutable tree having from 1 to
5  * infinitive number of elements. In the tree we hold
6  * elements on each node and nodes can have left and
7  * right subtrees...
8  *
9  * @param T the type of elements this tree holds.
10 * @property value the value kept in this node of the tree.
11 * @property left the left subtree.
12 * @property right the right subtree.
13 */
14 class Tree<T>(
15     val value: T,
16     val left: Tree<T>? = null,
17     val right: Tree<T>? = null
18 ) {
19     /**
20      * Creates a new tree based on the current but with
21      * [element] added.
22      * @return newly created tree with additional element.
23     */
24     operator fun plus(element: T): Tree { ... }
25 }
```

Notice that not everything needs to be described. The best documentation is short and on-point describes what might be unclear.

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 an expectation, all of its subclasses should

guarantee that too. This principle is known as *Liskov substitution principle*, and it is one of the most important rules in the *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 why it is important is that every class can be used as a superclass, and so if it 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:

```
1  interface Car {  
2      fun setWheelPosition(angle: Float)  
3      fun setBreakPedal(pressure: Double)  
4      fun setGasPedal(pressure: Double)  
5  }  
6  
7  class GasolineCar: Car {  
8      // ...  
9  }  
10  
11 class GasCar: Car {  
12     // ...  
13 }  
14  
15 class ElectricCar: Car {  
16     // ...  
17 }
```

The problem with this interface is that it leaves a lot of questions. What does `angle` in the `setWheelPosition` function means? In what units it is 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 brands should behave similarly when they are used as a Car. We can address those concerns with documentation:

```
1 interface Car {  
2     /**  
3         * Changes car direction.  
4         *  
5         * @param angle Represents position of wheels in  
6         * radians relatively to car axis. 0 means driving  
7         * straight, pi/2 means driving maximally right,  
8         * -pi/2 maximally left.  
9         * Value needs to be in (-pi/2, pi/2)  
10        */  
11    fun setWheelPosition(angle: Float)  
12  
13    /**  
14        * Decelerates vehicle speed until 0.  
15        *  
16        * @param pressure The percentage of brake pedal use.  
17        * Number from 0 to 1 where 0 means not using break  
18        * at all, and 1 means maximal pedal pedal use.  
19        */  
20    fun setBreakPedal(pressure: Double)  
21  
22    /**  
23        * Accelerates vehicle speed until max speed possible  
24        * for user.  
25        *  
26        * @param pressure The percentage of gas pedal use.  
27        * Number from 0 to 1 where 0 means not using gas at  
28        * all, and 1 means maximal gas pedal use.  
29        */  
30    fun setGasPedal(pressure: Double)  
31 }
```

Now all cars have set a 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 define contracts for our elements as well. Those contracts will make those interfaced truly useful. They will give us the freedom to use classes that implement those interfaces in the way their contract guarantees.

Leaking implementation

Implementation details always leak. In a car, different kinds of engines behave a bit differently. We are still able to drive the car, but we can feel a difference. It is fine as this 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. Though as long as a language works as it promises, everything is fine. We just need to remember and apply good practices.

In our abstractions, implementation will leak as well, but still, we should protect it as much as we can. We protect it by encapsulation, which can be described as “You can do what I allow, and nothing more”. The more encapsulated classes and functions are, the more freedom we have inside them because we don’t need to think about how one might depend on our implementation.

Summary

When we define an element, especially parts of external API, we should define a contract. We do that through names, documentation, comments, and types. The contract specifies what the

expectations are on those elements. 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, and 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 32: Respect abstraction contracts

Both contract and visibility are kind of an agreement between developers. This agreement nearly always can 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:

```
1 class Employee {
2     private val id: Int = 2
3     override fun toString() = "User(id=$id)"
4
5     private fun privateFunction() {
6         println("Private function called")
7     }
8 }
9
10 fun callPrivateFunction(employee: Employee) {
11     employee::class.declaredMemberFunctions
12         .first { it.name == "privateFunction" }
13         .apply { isAccessible = true }
14         .call(employee)
15 }
16
17 fun changeEmployeeId(employee: Employee, newId: Int) {
18     employee::class.java.getDeclaredField("id")
19         .apply { isAccessible = true }
20         .set(employee, newId)
21 }
22
23 fun main() {
24     val employee = Employee()
25     callPrivateFunction(employee)
```

```
26 // Prints: Private function called  
27  
28     changeEmployeeId(employee, 1)  
29     print(employee) // Prints: User(id=1)  
30 }
```

Just because you can do something, doesn't mean that it is fine to do it. Here we very strongly depend on the implementation details like the names of the private property and the private function. They are not part of a contract at all, and so they might change at any moment. This is like a ticking 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 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 your object should respect their contracts. For instance, every class extends Any that have equals and hashCode methods. They both 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. Below behavior is incorrect because a set should not allow duplicates:

```
1 class Id(val id: Int) {  
2     override fun equals(other: Any?) =  
3         other is Id && other.id == id  
4 }  
5  
6 val mutableSet = mutableSetOf(Id(1))  
7 mutableSet.add(Id(1))  
8 mutableSet.add(Id(1))  
9 print(mutableSet.size) // 3
```

In this case, it is that `hashCode` do not have implementation consistent with `equals`. We will discuss some important Kotlin contracts in *Chapter 6: Class design*. For now, remember to check the expectations on functions you override, and respect those.

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 will maintain your code. Maybe that will be 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 (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 ways have different characteristics. It is important to know what options do we have. This is why this chapter shows different ways how we can define 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. It is no coincidence. This chapter relates to the first chapter of Effective Java. Although Kotlin is very different from Java, and there are only morsels of knowledge that can be used. For instance, static methods are not allowed in Kotlin, but we have very good alternatives like top-level functions and companion object functions. They don't work the same way as static functions, so it is important to understand them. Similarly, with other items, you can notice similarities, but the changes that Kotlin has introduced are important. To cheer you up: these changes are mostly to provide more possibilities or force better style. Kotlin is a powerful and really well-designed language, and this chapter should mainly open your eyes to these new possibilities.

Item 33: Consider factory functions instead of constructors

The most common way for a class to allow a client to obtain an instance in Kotlin is to provide a primary constructor:

```
1 class MyLinkedList<T>(
2     val head: T,
3     val tail: MyLinkedList<T>?
4 )
5
6 val list = MyLinkedList(1, MyLinkedList(2, null))
```

Though constructors are not the only way to create objects. There are many creational design patterns for object instantiation. Most of them revolve around the idea that instead of directly creating an object, a function can create the object for us. For instance, the following top-level function creates an instance of `MyLinkedList`:

```
1 fun <T> myLinkedListOf(
2     vararg elements: T
3 ): MyLinkedList<T>? {
4     if(elements.isEmpty()) return null
5     val head = elements.first()
6     val elementsTail = elements
7         .copyOfRange(1, elements.size)
8     val tail = myLinkedListOf(*elementsTail)
9     return MyLinkedList(head, tail)
10 }
11
12 val list = myLinkedListOf(1, 2)
```

Functions used as an alternative to constructors are called factory functions because they produce an object. Using factory functions instead of a constructor has many advantages, including:

- **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 on the newly created list, or is it the size of the list? It is definitely not self-explanatory. In such situation a name, like `ArrayList.withSize(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 a conflict between constructors with the same parameter types.
- **Unlike constructors, functions can return an object of any subtype of their return type.** This can be used to provide a better object for different cases. It is especially important when we want to hide actual object implementations behind an interface. Think of `listOf` from `stdlib` (standard library). Its declared return type is `List` which is an interface. What does it 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 made by the Kotlin team. It also gives Kotlin creators much more freedom because the actual type of list might change over time and as long as new objects still implement interface `List` and act the same way, everything will be fine.
- **Unlike constructors, functions are not required to create a new object each time they're invoked.** It 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. This way, programmers can operate on objects that will be generated or used via proxy without building the project.

- When we define a factory function outside of an object, we can control its visibility. For instance, we can make a top-level factory function accessible only in the same file or in the same module.
- Factory functions can be inline and so their type parameters can be reified.
- Factory functions can construct objects which might otherwise be complicated to construct.
- 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:

```
1 fun makeListView(config: Config) : ListView {  
2     val items = ... // Here we read items from config  
3     return ListView(items) // We call actual constructor  
4 }
```

There is a limitation on factory functions usage: it cannot be used in subclass construction. This is because in subclass construction, we need to call the superclass constructor.

```
1 class IntLinkedList: MyLinkedList<Int>() {  
2     // Supposing that MyLinkedList is open  
3  
4     constructor(vararg ints: Int): myLinkedListOf(*ints)  
5     // Error  
6 }
```

That's generally not a problem, since if we have decided that we want to create a superclass using a factory function, why would we use a constructor for its subclass? We should rather consider implementing a factory function for such class as well.

```
1 class MyLinkedList<Int>(head: Int, tail: MyLinkedList<Int>?) :  
2     MyLinkedList<Int>(head, tail)  
3  
4 fun myLinkedListOf(vararg elements: Int) :  
5     MyLinkedList<Int>? {  
6     if(elements.isEmpty()) return null  
7     val head = elements.first()  
8     val elementsTail = elements  
9         .copyOfRange(1, elements.size)  
10    val tail = myLinkedListOf(*elementsTail)  
11    return MyLinkedList(head, tail)  
12 }
```

The above function is longer than the previous constructor, but it has better characteristics - flexibility, independence of class, and the ability to declare a nullable return type.

There are strong reasons standing behind factory functions, though what needs to be understood is that **they are not a competition to the primary constructor**⁴⁷. Factory functions still need to use a constructor in their body, so constructor must exist. It can be private if we really want to force creation using factory functions, but we rarely do (*Item 34: Consider primary constructor with named optional arguments*). **Factory functions are mainly a competition to secondary constructors**, and looking at Kotlin projects they generally win as secondary constructors are used rather rarely. **They are also a competition to themselves as there are variety of different kinds of factory functions**. Let's discuss different Kotlin factory functions:

1. Companion object factory function
2. Extension factory function
3. Top-level factory functions
4. Fake constructors
5. Methods on a factory classes

⁴⁷See section about primary/secondary constructor in the dictionary

Companion Object Factory Function

The most popular way to define a factory function is to define it in a companion object:

```
1 class MyLinkedList<T>(
2     val head: T,
3     val tail: MyLinkedList<T>?
4 ) {
5
6     companion object {
7         fun <T> of(vararg elements: T): MyLinkedList<T>? {
8             /*...*/
9         }
10    }
11 }
12
13 // Usage
14 val list = MyLinkedList.of(1, 2)
```

Such approach should be very familiar to Java developers because it is a direct equivalent to a static factory method. Though developers of other languages might be familiar with it as well. In some languages, like C++, it is called a *Named Constructor Idiom* as its usage is similar to a constructor, but with a name.

In Kotlin, this approach works with interfaces too:

```
1  class MyLinkedList<T>(
2      val head: T,
3      val tail: MyLinkedList<T>?
4  ): MutableList<T> {
5      // ...
6  }
7
8  interface MutableList<T> {
9      // ...
10
11     companion object {
12         fun <T> of(vararg elements: T): MutableList<T>? {
13             // ...
14         }
15     }
16 }
17
18 // Usage
19 val list = MutableList.of(1, 2)
```

Notice that the name of the above function is not really descriptive, and yet it should be understandable for most developers. The reason is that there are some conventions that come from Java and thanks to them, a short word like `of` is enough to understand what the arguments mean. Here are some common names with their descriptions:

- `from` - A type-conversion function that takes a single parameter and returns a corresponding instance of the same type, for example:

```
val date: Date = Date.from(instant)
```

- `of` - An aggregation function that takes multiple parameters 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 only instance. When parameterized, will return an instance parameterized by arguments. Often we can expect that returned instance to always be the same when 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)
```

- `getType` - Like `getInstance`, 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 fs: FileStore = Files.getFileStore(path)
```

- `newType` - 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)
```

Many less-experienced Kotlin developers treat companion object members like static members which need to be grouped in a single block. However, companion objects are actually much more powerful: for example, companion objects can implement interfaces and extend classes. So, we can implement general companion object factory functions like the one below:

```
1 abstract class ActivityFactory {
2     abstract fun getIntent(context: Context): Intent
3
4     fun start(context: Context) {
5         val intent = getIntent(context)
6         context.startActivity(intent)
7     }
8
9     fun startForResult(activity: Activity, requestCode: Int) {
10        val intent = getIntent(activity)
11        activity.startActivityForResult(intent,
12            requestCode)
13    }
14 }
15
16
17 class MainActivity : AppCompatActivity() {
18     //...
19
20     companion object: ActivityFactory() {
21         override fun getIntent(context: Context): Intent =
22             Intent(context, MainActivity::class.java)
23     }
24 }
25
26 // Usage
27 val intent = MainActivity.getIntent(context)
```

```
28 MainActivity.start(context)
29 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 products, you will see that companion objects are strongly used. For instance in the Kotlin Coroutines library, nearly every companion object of coroutine context implements an interface `CoroutineContext.Key` as they all serve as a key we use to identify this context.

Extension factory functions

Sometimes we want to create a factory function that acts like an existing companion object function, and we either cannot modify this companion object or we just want to specify a new function in a separate file. In such a case we can use another advantage of companion objects: we can define extension functions for them.

Suppose that we cannot change the `Tool` interface:

```
1 interface Tool {
2     companion object { /*...*/ }
3 }
```

Nevertheless, we can define an extension function on its companion object:

```
1 fun Tool.Companion.createBigTool( /*...*/ ) : BigTool {
2     //...
3 }
```

At the call site we can then write:

```
1 Tool.createBigTool()
```

This is a powerful possibility that lets us extend external libraries with our own factory methods. One catch is that to make an extension on companion object, there must be some (even empty) companion object:

```
1 interface Tool {  
2     companion object {}  
3 }
```

Top-level functions

One popular way to create an object is by using top-level factory functions. Some common examples are `listOf`, `setOf`, and `mapOf`. Similarly, library designers specify top-level functions that are used to create objects. Top-level factory functions are used widely. For example, in Android, we have the tradition of defining a function to create an `Intent` to start an Activity. In Kotlin, the `getIntent()` can be written as a companion object function:

```
1 class MainActivity: Activity {  
2  
3     companion object {  
4         fun getIntent(context: Context) =  
5             Intent(context, MainActivity::class.java)  
6     }  
7 }
```

In the Kotlin Anko library, we can use the top-level function `intentFor` with reified type instead:

```
1 intentFor<MainActivity>()
```

This function can be also used to pass arguments:

```
1 intentFor<MainActivity>("page" to 2, "row" to 10)
```

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. It is easy to clutter up the developer's IDE tips. The problem becomes more serious when top-level functions are named like class methods and they are confused with them. This is why top-level functions should be named wisely.

Fake constructors

Constructors in Kotlin are used the same way as top-level functions:

```
1 class A
2 val a = A()
```

They are also referenced the same as top-level functions (and constructor reference implements function interface):

```
1 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; functions a lower case letter. Although technically functions can begin with an uppercase. This fact is used in different places, for example, in case of the Kotlin standard library. `List` and `MutableList` are interfaces. They cannot have constructors, but Kotlin developers wanted to allow the following `List` construction:

```
1 List(4) { "User$it" } // [User0, User1, User2, User3]
```

This is why the following functions are included (since Kotlin 1.1) in the Kotlin stdlib:

```
1 public inline fun <T> List(
2     size: Int,
3     init: (index: Int) -> T
4 ): List<T> = MutableList(size, init)
5
6 public inline fun <T> MutableList(
7     size: Int,
8     init: (index: Int) -> T
9 ): MutableList<T> {
10     val list = ArrayList<T>(size)
11     repeat(size) { index -> list.add(init(index)) }
12     return list
13 }
```

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*.

Two main reasons why developers choose fake constructors over the real ones are:

- To have “constructor” for an interface
- To have reified type arguments

Except for that, fake constructors should behave like normal constructors. They look like constructors and they should behave this way. If you want to include caching, returning a nullable type or returning a subclass of a class that can be created, consider using

a factory function with a name, like a companion object factory method.

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:

```
1 class Tree<T> {  
2  
3     companion object {  
4         operator fun <T> invoke(size: Int, generator:  
5             (Int)->T): Tree<T>{  
6             // ...  
7         }  
8     }  
9 }  
10  
11 // Usage  
12 Tree(10) { "$it" }
```

However, implementing `invoke` in a companion object to make a fake constructor is very rarely used and I do not recommend it. First of all, because it breaks *Item 12: Use operator methods according to their names*. What does it mean to invoke a companion object? Remember that the name can be used instead of the operator:

```
1 Tree.invoke(10) { "$it" }
```

Invocation is a different operation to object construction. Using the operator in this way is inconsistent with its name. More importantly, this approach is more complicated than just a top-level function. Looking at their reflection shows this complexity. Just compare how reflection looks like when we reference a constructor, fake constructor, and `invoke` function in a companion object:

Constructor:

```
1 val f: ()->Tree = ::Tree
```

Fake constructor:

```
1 val f: ()->Tree = ::Tree
```

Invoke in companion object:

```
1 val f: ()->Tree = Tree.Companion::invoke
```

I recommend using standard top-level functions when you need a fake constructor. 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 a factory class

There are many creational patterns associated with factory classes. For instance, abstract factory or prototype. Every one of them has some advantages.

We will see that some of these approaches are not reasonable in Kotlin. In the next item, we will see that the telescoping constructor and builder pattern rarely make sense in Kotlin.

Factory classes hold advantages over factory functions because classes can have a state. For instance, this very simple factory class that produces students with next id numbers:

```
1  data class Student(  
2      val id: Int,  
3      val name: String,  
4      val surname: String  
5  )  
6  
7  class StudentsFactory {  
8      var nextId = 0  
9      fun next(name: String, surname: String) =  
10         Student(nextId++, name, surname)  
11    }  
12  
13  val factory = StudentsFactory()  
14  val s1 = factory.next("Marcin", "Moskala")  
15  println(s1) // Student(id=0, name=Marcin, Surname=Moskala)  
16  val s2 = factory.next("Igor", "Wojda")  
17  println(s2) // Student(id=1, name=Igor, Surname=Wojda)
```

Factory classes can have properties and those properties 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.

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. Some of them should preferably be used with caution: Fake Constructors, Top-Level Factory Method, and Extension Factory Function. The most universal way to define a factory function is by using a Companion Object. It is safe and very intuitive for most developers since usage is very similar to

Java Static Factory Methods, and Kotlin mainly inherits its style and practices from Java.

Item 34: 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 a primary constructor:

```
1 class User(var name: String, var surname: String)  
2 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 very common that we need to pass arguments that determine the object's initial state, as illustrated by the following examples. Starting from the most obvious one: data model objects representing data⁴⁸. For such object, whole state is initialized using constructor and then hold as properties.

```
1 data class Student(  
2     val name: String,  
3     val surname: String,  
4     val age: Int  
5 )
```

Here's another common example, in which we create a presenter⁴⁹ for displaying a sequence of indexed quotes. There we inject dependencies using a primary constructor:

⁴⁸A data model is not necessarily a data class, and vice versa. The first concept represents classes in our project that represent data, while the latter is a special support for such classes. This special support is a set of functions that we might not need or that we might need for other classes.

⁴⁹A Presenter is a kind of object used in the Model-View-Presenter (MVP) architecture that is very popular in Android

```
1 class QuotationPresenter(  
2     private val view: QuotationView,  
3     private val repo: QuotationRepository  
4 ) {  
5     private var nextQuoteId = -1  
6  
7     fun onStart() {  
8         onNext()  
9     }  
10  
11    fun onNext() {  
12        nextQuoteId = (nextQuoteId + 1) % repo.quotesNumber  
13        val quote = repo.getQuote(nextQuoteId)  
14        view.showQuote(quote)  
15    }  
16 }
```

Note that `QuotationPresenter` has more properties than those declared on a primary constructor. In here `nextQuoteId` is a property always initialized with the value `-1`. This is perfectly fine, especially when the initial state is set up using default values or using primary constructor parameters.

To better understand why the primary constructor is such a good alternative 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 what problems they do solve, and what better alternatives Kotlin offers.

Telescoping constructor pattern

The telescoping constructor pattern is nothing more than a set of constructors for different possible sets of arguments:

```
1 class Pizza {
2     val size: String
3     val cheese: Int
4     val olives: Int
5     val bacon: Int
6
7     constructor(size: String, cheese: Int, olives: Int,
8 bacon: Int) {
9         this.size = size
10        this.cheese = cheese
11        this.olives = olives
12        this.bacon = bacon
13    }
14    constructor(size: String, cheese: Int, olives: Int):
15 this(size, cheese, olives, 0)
16    constructor(size: String, cheese: Int):
17 this(size, cheese, 0)
18    constructor(size: String): this(size, 0)
19 }
```

Well, this code doesn't really make any sense in Kotlin, because instead we can use default arguments:

```
1 class Pizza(
2     val size: String,
3     val cheese: Int = 0,
4     val olives: Int = 0,
5     val bacon: Int = 0
6 )
```

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`:

```
1 val myFavorite = Pizza("L", olives = 3)
```

We can also add another named argument either before or after olives:

```
1 val myFavorite = Pizza("L", olives = 3, cheese = 1)
```

As you can see, default arguments are more powerful than the telescoping constructor because:

- We can set any subset of parameters with 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:

```
1 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 in which position bacon is, and in which position cheese can be found. 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 say what their names are using *named arguments*:

```
1 val villagePizza = Pizza(  
2     size = "L",  
3     cheese = 1,  
4     olives = 2,  
5     bacon = 3  
6 )
```

As you can see, **constructors with default arguments surpass the telescoping constructor pattern**. Though there are more popular construction patterns in Java, and one of them is the Builder Pattern.

Builder pattern

Named parameters and default arguments are not allowed in Java. This is why Java developers mainly use the builder pattern. It allows them to:

- name parameters,
- specify parameters in any order,
- have default values.

Here is an example of a builder defined in Kotlin:

```
1 class Pizza private constructor(  
2     val size: String,  
3     val cheese: Int,  
4     val olives: Int,  
5     val bacon: Int  
6 ) {  
7     class Builder(private val size: String) {  
8         private var cheese: Int = 0  
9         private var olives: Int = 0  
10        private var bacon: Int = 0
```

```
11
12     fun setCheese(value: Int): Builder = apply {
13         cheese = value
14     }
15     fun setOlives(value: Int): Builder = apply {
16         olives = value
17     }
18
19     fun setBacon(value: Int): Builder = apply {
20         bacon = value
21     }
22
23     fun build() = Pizza(size, cheese, olives, bacon)
24 }
25 }
```

With the builder pattern, we can set those parameters as we want, using their names:

```
1 val myFavorite = Pizza.Builder("L").setOlives(3).build()
2
3 val villagePizza = Pizza.Builder("L")
4     .setCheese(1)
5     .setOlives(2)
6     .setBacon(3)
7     .build()
```

As we've already mentioned, these two advantages are fully satisfied by Kotlin default arguments and named parameters:

```
1 val villagePizza = Pizza(  
2     size = "L",  
3     cheese = 1,  
4     olives = 2,  
5     bacon = 3  
6 )
```

Comparing these two simple usages, you can see the advantages of named parameters over the builder:

- It's 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. It is a significant difference because the builder pattern implementation can be time-consuming. Any builder modification is hard as well, for instance, changing a name of a parameter requires not only name change of the function used to set it, but also name of parameter in this function, body of this function, internal field used to keep it, parameter name in the private constructor etc.
- It's cleaner—when you want to see how an object is constructed, all you need is in a single method instead of being spread 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. On the other hand, class creation is usually clear on a factory method.
- Offers simpler usage - the primary constructor is a built-in concept. The builder pattern is an artificial concept and it requires some knowledge about it. For instance, a developer can easily forget to call the `build` function (or in other cases `create`).
- 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.

It doesn't mean that we should always use a constructor instead of a builder. Let's see cases where different advantages of this pattern shine.

Builders can require a set of values for a name (`setPositiveButton`, `setNegativeButton`, and `addRoute`), and allows us to aggregate (`addRoute`):

```
1 val dialog = AlertDialog.Builder(context)
2     .setMessage(R.string.fire_missiles)
3     .setPositiveButton(R.string.fire, { d, id ->
4         // FIRE MISSILES!
5     })
6     .setNegativeButton(R.string.cancel, { d, id ->
7         // User cancelled the dialog
8     })
9     .create()
10
11 val router = Router.Builder()
12     .addRoute(path = "/home", ::showHome)
13     .addRoute(path = "/users", ::showUsers)
14     .build()
```

To achieve similar behavior with a constructor we would need to introduce special types to hold more data in a single argument:

```
1 val dialog = AlertDialog(context,
2     message = R.string.fire_missiles,
3     positiveButtonDescription =
4         ButtonDescription(R.string.fire, { d, id ->
5             // FIRE MISSILES!
6         }),
7     negativeButtonDescription =
8         ButtonDescription(R.string.cancel, { d, id ->
9             // User cancelled the dialog
10        })
```

```
11  )
12
13 val router = Router(
14     routes = listOf(
15         Route("/home", ::showHome),
16         Route("/users", ::showUsers)
17     )
18 )
```

This notation is generally badly received in the Kotlin community, and we tend to prefer using DSL (Domain Specific Language) builder for such cases:

```
1 val dialog = context.alert(R.string.fire_missiles) {
2     positiveButton(R.string.fire) {
3         // FIRE ZE MISSILES!
4     }
5     negativeButton {
6         // User cancelled the dialog
7     }
8 }
9
10 val route = router {
11     "/home" directsTo ::showHome
12     "/users" directsTo ::showUsers
13 }
```

These kinds of DSL builders are generally preferred over classic builder pattern, since they give more flexibility and cleaner notation. 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 a 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 are going to talk more about using DSLs for object creation.

Another advantage of the classic builder pattern is that it can be used as a factory. It might be filled partially and passed further, for example a default dialog in our application:

```
1 fun Context.makeDefaultDialogBuilder() =  
2     AlertDialog.Builder(this)  
3         .setIcon(R.drawable.ic_dialog)  
4         .setTitle(R.string.dialog_title)  
5         .setOnCancelListener { it.cancel() }
```

To have a similar possibility in a constructor or factory method, we would need currying, which is not supported in Kotlin. Alternatively we could keep object configuration in a data class and use copy to modify an existing one:

```
1 data class DialogConfig(  
2     val icon: Int = -1,  
3     val title: Int = -1,  
4     val onCancelListener: (() -> Unit)? = null  
5     //...  
6 )  
7  
8 fun makeDefaultDialogConfig() = DialogConfig(  
9     icon = R.drawable.ic_dialog,  
10    title = R.string.dialog_title,  
11    onCancelListener = { it.cancel() }  
12 )
```

Although both options are rarely seen as an option. If we want to define, let's say, a default dialog for an application, we can create it using a function and pass all customization elements as optional arguments. Such a method would have more control over dialog creation. This is why this advantage of the builder pattern I treat as minor.

In the end, the 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 builder pattern,
- when we design API to be easily used in other languages that do not support default arguments or DSLs.

Except of that, we rather prefer either a primary constructor with default arguments, or an expressive DSL.

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 builder pattern is very rarely reasonable either as in simpler cases we can just use a primary constructor with named arguments, and when we need to create more complex object we can use define a DSL for that.

Item 35: Consider defining a DSL for complex object creation

Edu

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 they are defined they hide boilerplate and complexity, and a developer can express his or her intentions clearly.

For instance, Kotlin DSL is a popular way to express HTML: both classic HTML, and React HTML. This is how it can look like:

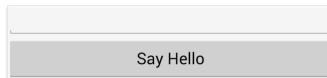
```
1 body {  
2     div {  
3         a("https://kotlinlang.org") {  
4             target = ATarget.blank  
5             +"Main site"  
6         }  
7     }  
8     +"Some content"  
9 }
```

Main site
Some content

View from the above HTML DSL

Views on other platforms can be defined using DSLs as well. Here is a simple Android view defined using the Anko library:

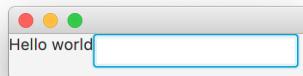
```
1 verticalLayout {  
2     val name = editText()  
3     button("Say Hello") {  
4         onClick { toast("Hello, ${name.text}!") }  
5     }  
6 }
```



View from the above Android View DSL

Similarly with desktop applications. Here is a view defined on TornadoFX that is built on top of the JavaFX:

```
1 class HelloWorld : View() {  
2     override val root = hbox {  
3         label("Hello world") {  
4             addClass(heading)  
5         }  
6  
7         textfield {  
8             promptText = "Enter your name"  
9         }  
10    }  
11 }
```



View from the above TornadoFX DSL

DSLs are also often used to define data or configurations. Here is API definition in Ktor, also a DSL:

```
1 fun Routing.api() {
2     route("news") {
3         get {
4             val newsData = NewsUseCase.getAcceptedNews()
5             call.respond(newsData)
6         }
7         get("propositions") {
8             requireSecret()
9             val newsData = NewsUseCase.getPropositions()
10            call.respond(newsData)
11        }
12    }
13    // ...
14 }
```

And here are test case specifications defined in Kotlin Test:

```
1 class MyTests : StringSpec({
2     "length should return size of string" {
3         "hello".length shouldBe 5
4     }
5     "startsWith should test for a prefix" {
6         "world" should startWith("wor")
7     }
8 })
```

We can even use Gradle DSL to define Gradle configuration:

```
1  plugins {
2      `java-library`
3  }
4
5  dependencies {
6      api("junit:junit:4.12")
7      implementation("junit:junit:4.12")
8      testImplementation("junit:junit:4.12")
9  }
10
11 configurations {
12     implementation {
13         resolutionStrategy.failOnVersionConflict()
14     }
15 }
16
17 sourceSets {
18     main {
19         java.srcDir("src/core/java")
20     }
21 }
22
23 java {
24     sourceCompatibility = JavaVersion.VERSION_11
25     targetCompatibility = JavaVersion.VERSION_11
26 }
27
28 tasks {
29     test {
30         testLogging.showExceptions = true
31     }
32 }
```

Creating complex and hierarchical data structures become easier with DSLs. Inside those 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 already used some Kotlin DSL, but it is also important to know how to define them yourself.

Defining your own DSL

To understand how to make own DSLs, it is important to understand the notion of function types with a receiver. But before that, we'll first briefly review the notion of function types themselves. The function type is a type that represents an object that can be used as a function. For instance, in the `filter` function, it is there to represent a predicate that decides if an element can be accepted or not.

```
1 inline fun <T> Iterable<T>.filter(  
2     predicate: (T) -> Boolean  
3 ): List<T> {  
4     val list = arrayListOf<T>()  
5     for (elem in this) {  
6         if (predicate(elem)) {  
7             list.add(elem)  
8         }  
9     }  
10    return list  
11 }
```

Here are a few examples of function types:

- `()->Unit` - Function with no arguments and 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 to create instances of function types are:

- Using lambda expressions
- Using anonymous functions
- Using function references

For instance, think of the following function:

```
1 fun plus(a: Int, b: Int) = a + b
```

Analogical function can be created in the following ways:

```
1 val plus1: (Int, Int)->Int = { a, b -> a + b }
2 val plus2: (Int, Int)->Int = fun(a, b) = a + b
3 val plus3: (Int, Int)->Int = ::plus
```

In the above example, property types are specified and so argument types in the lambda expression and in the anonymous function can be inferred. It could be the other way around. If we specify argument types, then the function type can be inferred.

```
1 val plus4 = { a: Int, b: Int -> a + b }
2 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 without a name. A lambda expression is a shorter notation for an anonymous function.

Although if we have function types to represent functions, what about extension functions? Can we express them as well?

```
1 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. And so anonymous extension functions are defined the same way:

```
1 val myPlus = fun Int.(other: Int) = this + other
```

What type does it have? The answer is that there is a special type to represent extension functions. It is called *function type with 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:

```
1 val myPlus: Int.(Int)->Int =
2     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):

```
1 val myPlus: Int.(Int)->Int = { this + it }
```

Object created using anonymous extension function or lambda expression with receiver can be invoked in 3 ways:

- Like a standard object, using `invoke` method.
- Like a non-extension function.
- Same as a normal extension function.

```
1 myPlus.invoke(1, 2)
2 myPlus(1, 2)
3 1.myPlus(2)
```

The most important trait of the function type with receiver is that it changes what `this` refers to. `this` is used for instance in the `apply` function to make it easier to reference the receiver object's methods and properties:

```
1 inline fun <T> T.apply(block: T.() -> Unit): T {
2     this.block()
3     return this
4 }
5
6 class User {
7     var name: String = ""
8     var surname: String = ""
9 }
10
11 val user = User().apply {
12     name = "Marcin"
13     surname = "Moskała"
14 }
```

Function type with a receiver is the most basic building block of Kotlin DSL. Let's create a very simple DSL that would allow us to make the following HTML table:

```
1 fun createTable(): TableDsl = table {  
2     tr {  
3         for (i in 1..2) {  
4             td {  
5                 +"This is column $i"  
6             }  
7         }  
8     }  
9 }
```

Starting from the beginning of this DSL, we can see a function **table**. We are at top-level without any receivers so it needs to be a top-level function. Although 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.

```
1 fun table(init: TableBuilder.()>Unit): TableBuilder {  
2     //...  
3 }  
4  
5 class TableBuilder {  
6     fun tr(init: TrBuilder.() -> Unit) { /*...*/ }  
7 }  
8  
9 class TrBuilder {  
10     fun td(init: TdBuilder.()>Unit) { /*...*/ }  
11 }  
12  
13 class TdBuilder
```

How about this statement:

```
1 +"This is row $i"
```

What is that? This is nothing else but a unary plus operator on String. It needs to be defined inside TdBuilder:

```
1 class TdBuilder {
2     var text = ""
3
4     operator fun String.unaryPlus() {
5         text += this
6     }
7 }
```

Now our DSL is well defined. To make it work fine, at every step we need to create a builder and initialize it using a function from parameter (`init` in the example below). After that, 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 holding this data. In this example, we'll just return builder. This is how the `table` function could be defined:

```
1 fun table(init: TableBuilder.() -> Unit): TableBuilder {
2     val tableBuilder = TableBuilder()
3     init.invoke(tableBuilder)
4     return tableBuilder
5 }
```

Notice that we can use the `apply` function, as shown before, to shorten this function:

```
1 fun table(init: TableBuilder.() -> Unit) =
2     TableBuilder().apply(init)
```

Similarly we can use it in other parts of this DSL to be more concise:

```
1  class TableBuilder {
2      var trs = listOf<TrBuilder>()
3
4      fun tr(init: TrBuilder.() -> Unit) {
5          trs = trs + TrBuilder().apply(init)
6      }
7  }
8
9  class TrBuilder {
10     var tds = listOf<TdBuilder>()
11
12    fun td(init: TdBuilder.() -> Unit) {
13        tds = tds + TdBuilder().apply(init)
14    }
15 }
```

This is a fully functional DSL builder for HTML table creation. It could be improved using a `DslMarker` explained in Item 15: Consider referencing receiver explicitly.

When should we use it?

DSLs give us a way to define information. It can be used to express any kind of information you want, but it is never clear to users how this information will be later used. 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 how exactly. Some more complicated uses can be hard to discover. Usage can be also confusing to those not used to them. Not to mention maintenance. The way how they are defined can be a cost - both in developer confusion and in performance. DSLs are an overkill when we can use other, simpler features instead. Though they are really useful when we need to express:

- complicated data structures,

- hierarchical structures,
- huge amount of data.

Everything can be expressed without DSL-like structure, by using builders or just constructors instead. **DSLs are about boilerplate elimination for such structures.** You should consider using DSL when you see repeatable boilerplate code⁵⁰ and there are no simpler Kotlin features that can help.

Summary

A DSL is a special language inside of a language. It can make it really simple to create complex object, and even whole object hierarchies, like HTML code or complex configuration files. On the other hand DSL implementations might be confusing or hard for new developers. They are also hard to define. This is why they should be only used when they offer real value. For instance, for the creation of a really complex object, or possibly 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 DSL can make our project much better.

⁵⁰Repeatable code not containing any important information for a reader

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 very important for us as well. This chapter is about class design. Not about system design, since it 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 talk about 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 kind of questions we will answer here. They are all important because breaking them 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 the “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, and we should instead prefer a lighter alternative: class composition.

Simple behavior reuse

Let’s start with a simple problem: we have two classes with partially similar behavior - progress bar display before and hide after logic.

```
1  class ProfileLoader {  
2  
3      fun load() {  
4          // show progress  
5          // load profile  
6          // hide progress  
7      }  
8  }  
9  
10 class ImageLoader {  
11  
12     fun load() {  
13         // show progress  
14         // load image  
15         // hide progress  
16     }  
17 }
```

From my experience, many developers would extract this common behavior by extracting a common superclass:

```
1 abstract class LoaderWithProgress {
2
3     fun load() {
4         // show progress
5         innerLoad()
6         // hide progress
7     }
8
9     abstract fun innerLoad()
10 }
11
12 class ProfileLoader: LoaderWithProgress() {
13
14     override fun innerLoad() {
15         // load profile
16     }
17 }
18
19 class ImageLoader: LoaderWithProgress() {
20
21     override fun innerLoad() {
22         // load image
23     }
24 }
```

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 huge BaseXXX classes that accumulate many functionalities or too deep and complex hierarchies of types.

- 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 the method works.

Those 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 reusing its functionalities. This is an example of how we can use composition instead of inheritance to solve our problem:

```
1  class Progress {
2      fun showProgress() { /* show progress */ }
3      fun hideProgress() { /* hide progress */ }
4  }
5
6  class ProfileLoader {
7      val progress = Progress()
8
9      fun load() {
10          progress.showProgress()
11          // load profile
12          progress.hideProgress()
13      }
14  }
15
16 class ImageLoader {
17     val progress = Progress()
18
19     fun load() {
20         progress.showProgress()
21         // load image
```

```
22     progress.hideProgress()  
23 }  
24 }
```

Notice that composition is harder, as 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 progress is used and how it is used. It also gives the developer more power over how progress works.

Another thing to note is that composition is better in a case when we want to extract multiple pieces of functionality. For instance, information that loading has finished:

```
1 class ImageLoader {  
2     private val progress = Progress()  
3     private val finishedAlert = FinishedAlert()  
4  
5     fun load() {  
6         progress.showProgress()  
7         // load image  
8         progress.hideProgress()  
9         finishedAlert.show()  
10    }  
11 }
```

We cannot extend more than a single class, so if we would want to use inheritance instead, we would be forced to place both functionalities in a single superclass. This often leads to a complex hierarchy of types 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 alert in two subclasses, but not in the third one? One way to deal with this problem is to use a parameterized constructor:

```
1 abstract class InternetLoader(val showAlert: Boolean) {  
2  
3     fun load() {  
4         // show progress  
5         innerLoad()  
6         // hide progress  
7         if (showAlert) {  
8             // show alert  
9         }  
10    }  
11  
12    abstract fun innerLoad()  
13}  
14  
15 class ProfileLoader : InternetLoader(showAlert = true) {  
16  
17     override fun innerLoad() {  
18         // load profile  
19     }  
20}  
21  
22 class ImageLoader : InternetLoader(showAlert = false) {  
23  
24     override fun innerLoad() {  
25         // load image  
26     }  
27}
```

This is a bad solution. It takes functionality a subclass doesn't need and blocks it. The problem is compounded when the subclass cannot block other unneeded functionality. When we use inheritance we take everything from the superclass, not only what we need.

Taking the whole package

When we use inheritance, we take from superclass everything - both methods, expectations (contract) and behavior. Therefore inheritance is a great tool to represent the hierarchy of objects, but not necessarily to just reuse some common parts. For such cases, the composition is better because we can choose what behavior do we need. To think of an example, let's say that in our system we decided to represent a Dog that can bark and sniff:

```
1 abstract class Dog {  
2     open fun bark() { /*...*/ }  
3     open fun sniff() { /*...*/ }  
4 }
```

What if then we need to create a robot dog that can bark but can't sniff?

```
1 class Labrador: Dog()  
2  
3 class RobotDog : Dog() {  
4     override fun sniff() {  
5         throw Error("Operation not supported")  
6         // Do you really want that?  
7     }  
8 }
```

Notice that such a solution violates *interface-segregation principle* - RobotDog has a method it doesn't need. It also violates the Liskov Substitution Principle by breaking superclass behavior. On the other hand, what if your RobotDog needs to be a Robot class as well because Robot can calculate (have calculate method)? *Multiple inheritance* is not supported in Kotlin.

```
1 abstract class Robot {  
2     open fun calculate() { /*...*/ }  
3 }  
4  
5 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 was not yet 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 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 were added to it during its lifetime. This set can be created using inheritance from HashSet:

```
1 class CounterSet<T>: HashSet<T>() {  
2     var elementsAdded: Int = 0  
3         private set  
4  
5     override fun add(element: T): Boolean {  
6         elementsAdded++  
7         return super.add(element)  
8     }  
9  
10    override fun addAll(elements: Collection<T>): Boolean {  
11        elementsAdded += elements.size  
12        return super.addAll(elements)
```

```
13     }
14 }
```

This implementation might look good, but it doesn't work correctly:

```
1 val counterList = CounterSet<String>()
2 counterList.addAll(listOf("A", "B", "C"))
3 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`. The problem can be naively solved by removing custom `addAll` function:

```
1 class CounterSet<T>: HashSet<T>() {
2     var elementsAdded: Int = 0
3     private set
4
5     override fun add(element: T): Boolean {
6         elementsAdded++
7         return super.add(element)
8     }
9 }
```

Although 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 would do that, this implementation would break with a Java update. Together with this implementation, any other libraries which depend on our current implementation will 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. How can we solve this problem? We should use composition instead of inheritance:

```
1 class CounterSet<T> {
2     private val innerSet = HashSet<T>()
3     var elementsAdded: Int = 0
4         private set
5
6     fun add(element: T) {
7         elementsAdded++
8         innerSet.add(element)
9     }
10
11    fun addAll(elements: Collection<T>) {
12        elementsAdded += elements.size
13        innerSet.addAll(elements)
14    }
15 }
16
17 val counterList = CounterSet<String>()
18 counterList.addAll(listOf("A", "B", "C"))
19 print(counterList.elementsAdded) // 3
```

One problem is that in this case, we lose polymorphic behavior: `CounterSet` is not a `Set` anymore. To keep it, we can use the delegation pattern. The delegation pattern is when 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:

```
1  class CounterSet<T> : MutableSet<T> {
2      private val innerSet = HashSet<T>()
3      var elementsAdded: Int = 0
4          private set
5
6      override fun add(element: T): Boolean {
7          elementsAdded++
8          return innerSet.add(element)
9      }
10
11     override fun addAll(elements: Collection<T>): Boolean {
12         elementsAdded += elements.size
13         return innerSet.addAll(elements)
14     }
15
16     override val size: Int
17         get() = innerSet.size
18
19     override fun contains(element: T): Boolean =
20         innerSet.contains(element)
21
22     override fun containsAll(elements: Collection<T>):
23 Boolean = innerSet.containsAll(elements)
24
25     override fun isEmpty(): Boolean = innerSet.isEmpty()
26
27     override fun iterator() =
28         innerSet.iterator()
29
30     override fun clear() =
31         innerSet.clear()
32
33     override fun remove(element: T): Boolean =
34         innerSet.remove(element)
35
```

```
36     override fun removeAll(elements: Collection<T>):  
37     Boolean = innerSet.removeAll(elements)  
38  
39     override fun retainAll(elements: Collection<T>):  
40     Boolean = innerSet.retainAll(elements)  
41 }
```

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 interface delegation presented in action:

```
1 class CounterSet<T>(  
2     private val innerSet: MutableSet<T> = mutableSetOf()  
3 ) : MutableSet<T> by innerSet {  
4  
5     var elementsAdded: Int = 0  
6         private set  
7  
8     override fun add(element: T): Boolean {  
9         elementsAdded++  
10        return innerSet.add(element)  
11    }  
12  
13    override fun addAll(elements: Collection<T>): Boolean {  
14        elementsAdded += elements.size  
15        return innerSet.addAll(elements)  
16    }  
17 }
```

This is a case where delegation is a good choice: we need polymorphic behavior and inheritance would be dangerous. In most cases, polymorphic behavior is not needed or we use it in a different way.

In such a case composition without delegation is more suitable. It is easier to understand and more flexible.

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. The composition is easier to reuse and gives us more flexibility.

Restricting overriding

To prevent developers from extending classes that are not designed for an inheritance, we can just keep them final. Though if for a reason we need to allow inheritance, still all methods are final by default. To let developers override them, they must be set to open:

```
1 open class Parent {
2     fun a() {}
3     open fun b() {}
4 }
5
6 class Child: Parent() {
7     override fun a() {} // Error
8     override fun b() {}
9 }
```

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:

```
1 open class ProfileLoader: InternetLoader() {  
2  
3     final override fun loadFromInternet() {  
4         // load profile  
5     }  
6 }
```

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 do not depend on how a class is implemented, but only on its externally observable behavior.
- **Composition is more flexible** - We can only extend a single class, while we can compose many. When we inherit, we take everything, while 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 composed changes, it will only change our behavior if it changed its contract to the outside world.
- **Composition is more explicit** - This is both an advantage and a disadvantage. When we use a method from a superclass we don't need to reference any receiver (we don't need to use `this` keyword). It is less explicit, which means that it 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, superclass, 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 composed object explicitly. When we add some functionalities to a superclass we often do not need to modify subclasses. When we use composition we more often need to adjust usages.
- **Inheritance gives us a strong polymorphic behavior** - This is also a double-edged sword. From one side, it is comfortable that a dog can be treated like an animal. On the other side, it is very constraining. It must be an animal. Every subclass of the animal should be consistent with animal behavior. Superclass set contract and 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.

When is composition more reasonable then? The rule of thumb: **we should use inheritance when there is a definite “is a” relationship**. Not only linguistically, but meaning that every class that inherits from a superclass needs to “be” its superclass. All unit tests for superclasses should always pass for their subclasses (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, keep methods that are not designed for inheritance final.

Item 37: Use the data modifier to represent a bundle of data

Sometimes we just need to pass around a bundle of data. This is what we use data classes for. These are classes with the `data` modifier. From my experience, developers quickly introduce it into their data model classes:

```
1 data class Player(  
2     val id: Int,  
3     val name: String,  
4     val points: Int  
5 )  
6  
7 val player = Player(0, "Gecko", 9999)
```

When we add the `data` modifier, it generates a few useful functions:

- `toString`
- `equals` and `hashCode`
- `copy`
- `componentN` (`component1`, `component2`, etc.)

Let's discuss them one after another in terms of data classes.

`toString` displays the name of the class and the values of all primary constructor properties with their names. Useful for logging and debugging.

```
1 print(player) // Player(id=0, name=Gecko, points=9999)
```



`equals` checks if all primary constructor properties are equal, and `hashCode` is coherent with it (see 41: Respect the contract of `hashCode`).

```
1 player == Player(0, "Gecko", 9999) // true
2 player == Player(0, "Ross", 9999) // false
```

`copy` is especially useful for immutable data classes. It creates a new object where each primary constructor properties have the same value by default, but each of them can be changed using named arguments.

```
1 val newObj = player.copy(name = "Thor")
2 print(newObj) // Player(id=0, name=Thor, points=9999)
```

We cannot see the `copy` method because it is generated under the hood just like the other methods generated thanks to the `data` modifier. If we were able to see it, this is what it would look like for the class `Person`:

```
1 // This is how `copy` is generated under the hood by
2 // data modifier for `Person` class looks like
3 fun copy(
4     id: Int = this.id,
5     name: String = this.name,
6     points: Int = this.points
7 ) = Player(id, name, points)
```

Notice that `copy` method makes a shallow copy of an object, but it is not a problem when object is immutable - for immutable objects we do not need deep copies.

`componentN` functions (`component1`, `component2`, etc.) allow position-based destructuring. Like in the below example:

```
1 val (id, name, pts) = player
```

Destructuring in Kotlin translates directly into variable definitions using the `componentN` functions, so this is what the above line will be compiled to under the hood:

```
1 // After compilation
2 val id: Int = player.component1()
3 val name: String = player.component2()
4 val pts: Int = player.component3()
```

Position-based destructuring has its 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`:

```
1 val visited = listOf("China", "Russia", "India")
2 val (first, second, third) = visited
3 println("$first $second $third")
4 // China Russia India
5
6 val trip = mapOf(
7     "China" to "Tianjin",
8     "Russia" to "Petersburg",
9     "India" to "Rishikesh"
10 )
11 for ((country, city) in trip) {
12     println("We loved $city in $country")
13     // We loved Tianjin in China
14     // We loved Petersburg in Russia
15     // We loved Rishikesh in India
16 }
```

On the other hand, it is dangerous. We need to adjust every destructuring when the order of elements in data class change. It is also easy to destructure incorrectly by confusing order:

```
1 data class FullName(  
2     val firstName: String,  
3     val secondName: String,  
4     val lastName: String  
5 )  
6  
7 val elon = FullName("Elon", "Reeve", "Musk")  
8 val (name, surname) = elon  
9 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. Then in case of an incorrect order, an IntelliJ/Android Studio warning will be shown. It might be even useful to upgrade this warning into 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("It is $name $lastName") // It is Elon Reeve!  
Variable name 'lastName' matches the name of a different component more... (⌘F1)
```

Do not destructure to get just the first value like in the example below:

```
1 data class User(val name: String)  
2 val (name) = User("John")
```

This might be really confusing and misleading for a reader. Especially when you destructure in lambda expressions:

```
1 data class User(val name: String)  
2  
3 fun main() {  
4     val user = User("John")  
5     user.let { a -> print(a) } // User(name=John)  
6     // Don't do that  
7     user.let { (a) -> print(a) } // John  
8 }
```

It is problematic because in some languages parentheses around arguments in lambda expressions are optional or required.

Prefer data classes instead of tuples

Data classes offer more than what is generally offered by tuples. More concretely, Kotlin tuples are just generic data classes which are `Serializable` and have a custom `toString` method:

```
1 public data class Pair<out A, out B>(  
2     public val first: A,  
3     public val second: B  
4 ) : Serializable {  
5  
6     public override fun toString(): String =  
7         "($first, $second)"  
8 }  
9  
10 public data class Triple<out A, out B, out C>(  
11     public val first: A,  
12     public val second: B,  
13     public val third: C  
14 ) : Serializable {  
15  
16     public override fun toString(): String =  
17         "($first, $second, $third)"  
18 }
```

Why do I show only `Pair` and `Triple`? It is because they are the last tuples that are left in Kotlin. 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: (`Int`, `String`, `String`, `Long`). What we achieved, in the end, behaved the same as data classes, but 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. They stayed because they are used for local purposes:

- When we immediately name values:

```
1 val (description, color) = when {  
2     degrees < 5 -> "cold" to Color.BLUE  
3     degrees < 23 -> "mild" to Color.YELLOW  
4     else -> "hot" to Color.RED  
5 }
```

- To represent an aggregate not known in advance — as is commonly found in standard library functions:

```
1 val (odd, even) = numbers.partition { it % 2 == 1 }  
2 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 name and surname. One might represent this name and surname as a `Pair<String, String>`:

```
1 fun String.parseName(): Pair<String, String>? {  
2     val index0fLastSpace = this.trim().lastIndex0f(' ')  
3     if(index0fLastSpace < 0) return null  
4     val firstName = this.take(index0fLastSpace)  
5     val lastName = this.drop(index0fLastSpace)  
6     return Pair(firstName, lastName)  
7 }  
8  
9 // Usage  
10 val fullName = "Marcin Moskała"  
11 val (firstName, lastName) = fullName.parseName() ?: return
```

The problem is that when someone reads it, it is not clear that `Pair<String, String>` represents a full name. What is more, it is not clear what is the order of the values. Someone could think that surname goes first:

```
1 val fullName = "Marcin Moskała"  
2 val (lastName, firstName) = fullName.parseName() ?: return  
3 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:

```
1 data class FullName(  
2     val firstName: String,  
3     val lastName: String  
4 )  
5  
6 fun String.parseName(): FullName? {  
7     val index0fLastSpace = this.trim().lastIndex0f(' ')  
8     if(index0fLastSpace < 0) return null  
9     val firstName = this.take(index0fLastSpace)  
10    val lastName = this.drop(index0fLastSpace)  
11    return FullName(firstName, lastName)
```

```
12  }
13
14 // Usage
15 val fullName = "Marcin Moskała"
16 val (firstName, lastName) = fullName.parseName() ?: return
```

It costs nearly nothing, and improves the function significantly:

- The return type of this function is clear.
- The return type is shorter and easier to pass forward.
- If a user destructures to variables with different names than those described in the data class, 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, do not be afraid to use them.

Item 38: Use function types instead of 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 SAM's (Single-Abstract Method). Here is an example SAM used to pass information about what should happen when a view is clicked:

```
1 interface OnClickListener {
2     fun clicked(view: View)
3 }
```

When a function expects a SAM, we must pass an instance of an object that implements this interface⁵¹.

```
1 fun setOnClickListener(listener: OnClickListener) {
2     //...
3 }
4
5 setOnClickListener(object : OnClickListener {
6     override fun clicked(view: View) {
7         // ...
8     }
9 })
```

However, notice that declaring a parameter with a function type gives us much more freedom:

⁵¹Unless it is Java SAM and Java function: since in such cases there is special support and we can pass a function type instead

```
1 fun setOnClickListener(listener: (View) -> Unit) {  
2     //...  
3 }
```

Now, we can pass the parameter as:

- A lambda expression or an anonymous function

```
1 setOnClickListener { /*...*/ }  
2 setOnClickListener(fun(view) { /*...*/ })
```

- A function reference or bounded function reference

```
1 setOnClickListener(::println)  
2 setOnClickListener(this::showUsers)
```

- Objects that implement the declared function type

```
1 class ClickListener: (View)->Unit {  
2     override fun invoke(view: View) {  
3         // ...  
4     }  
5 }  
6  
7 setOnClickListener(ClickListener())
```

These options can cover a wider spectrum of use cases. On the other hand, one might argue that the advantage of a SAM is that it and its arguments are named. Notice that we can name function types using type aliases as well.

```
1 typealias OnClick = (View) -> Unit
```

Parameters can also be named. The advantage of naming them is that these names can then be suggested by default by an IDE.

```
1 fun setOnClickListener(listener: OnClick) { /*...*/ }
2 typealias OnClick = (view: View)->Unit
```

A screenshot of an IDE interface showing code completion. The code is:

```
fun main() {
    setOnClickListener { } // Completion dropdown is open
}
```

The completion dropdown shows three suggestions:

- view, id ->
- view: View, id: Int ->
- f b println(message: String) (ko)

Notice that when we use lambda expressions, we can also destructure arguments. Together, this makes function types generally a better option than SAMs.

This argument is especially true when we set many observers. The classic Java way often is to collect them in a single listener interface:

```
1 class CalendarView {
2     var listener: Listener? = null
3
4     interface Listener {
5         fun onDateClicked(date: Date)
6         fun onPageChanged(date: Date)
7     }
8 }
```

I believe this is largely a result of laziness. From an API consumer's point of view, it is better to set them as separate properties holding function types:

```
1 class CalendarView {  
2     var onDateClicked: ((date: Date) -> Unit)? = null  
3     var onPageChanged: ((date: Date) -> Unit)? = null  
4 }
```

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.

If you don't have a good reason to define an interface, prefer using function types. They are well supported and are used frequently by Kotlin developers.

When should we prefer a SAM?

There is one case when we prefer a SAM: When we design a class to be used from another language than Kotlin. Interfaces are cleaner for Java clients. They cannot see type aliases nor name suggestions. Finally, Kotlin function types when used from some languages (especially Java) require functions to return `Unit` explicitly:

```
1 // Kotlin  
2 class CalendarView() {  
3     var onDateClicked: ((date: Date) -> Unit)? = null  
4     var onPageChanged: OnDateClicked? = null  
5 }  
6  
7 interface OnDateClicked {  
8     fun onClick(date: Date)  
9 }  
10  
11 // Java  
12 CalendarView c = new CalendarView();  
13 c.setOnDateClicked(date -> Unit.INSTANCE);  
14 c.setOnPageChanged(date -> {});
```

This is why it might be reasonable to use SAM instead of function types when we design API for use from Java. Though in other cases, prefer function types.

Item 39: Prefer class hierarchies to tagged classes

It is not uncommon in large projects to find classes with a constant “mode” that specifies how the class should behave. We call such classes tagged as they contain a tag that specifies their mode of operation. There are many problems with them, and most of those problems originate from the fact that different responsibilities from different modes are fighting 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’s a real example from a large project⁵².

```
1  class ValueMatcher<T> private constructor(
2      private val value: T? = null,
3      private val matcher: Matcher
4  ){
5
6      fun match(value: T?) = when(matcher) {
7          Matcher.EQUAL -> value == this.value
8          Matcher.NOT_EQUAL -> value != this.value
9          Matcher.LIST_EMPTY -> value is List<*> &&
10         value.isEmpty()
11         Matcher.LIST_NOT_EMPTY -> value is List<*> &&
12         value.isNotEmpty()
13     }
14
15     enum class Matcher {
16         EQUAL,
17         NOT_EQUAL,
18         LIST_EMPTY,
19         LIST_NOT_EMPTY
```

⁵²In the full version, it contained many more modes.

```
20     }
21
22     companion object {
23         fun <T> equal(value: T) =
24             ValueMatcher<T>(value = value, matcher =
25             Matcher.EQUAL)
26
27         fun <T> notEqual(value: T) =
28             ValueMatcher<T>(value = value, matcher =
29             Matcher.NOT_EQUAL)
30
31         fun <T> emptyList() =
32             ValueMatcher<T>(matcher = Matcher.LIST_EMPTY)
33
34         fun <T> notEmptyList() =
35             ValueMatcher<T>(matcher =
36             Matcher.LIST_NOT_EMPTY)
37     }
38 }
```

There are many downsides to this approach:

- Additional boilerplate from handling multiple modes in a single class
- Inconsistently used properties, as they are used for different purposes. Also, the object generally has more properties than they need as they 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 set in a few ways.
- It is often required to use a factory method 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 those classes as a set of alternatives. Here is how it could have been implemented:

```
1 sealed class ValueMatcher<T> {
2     abstract fun match(value: T): Boolean
3
4     class Equal<T>(val value: T) : ValueMatcher<T>() {
5         override fun match(value: T): Boolean =
6             value == this.value
7     }
8
9     class NotEqual<T>(val value: T) : ValueMatcher<T>() {
10        override fun match(value: T): Boolean =
11            value != this.value
12    }
13
14    class EmptyList<T>() : ValueMatcher<T>() {
15        override fun match(value: T) =
16            value is List<*> && value.isEmpty()
17    }
18
19    class NotEmptyList<T>() : ValueMatcher<T>() {
20        override fun match(value: T) =
21            value is List<*> && value.isNotEmpty()
22    }
23 }
```

Such implementation is much cleaner as there are not multiple responsibilities tangled with each other. Every object has only the data it requires and can define what parameters it is willing to have. Using a type hierarchy allows to eliminate all shortcomings of tagged classes.

Sealed modifier

We do not necessarily need to use the `sealed` modifier. We could use `abstract` instead, but `sealed` forbids any subclasses to be defined outside of that file. Thanks to that, if we cover all those types in `when`, we do not need to use an `else` branch, as it's guaranteed to be exhaustive. Using this advantage, we can easily add new functionalities and know that we won't forget to cover them in these `when` statements.

This is a convenient way to define operations that behave differently for different modes. For instance, we can define `reversed` as an extension function using `when`, instead of by defining how it should behave in all subclasses. New functions can be added this way even as extensions.

```
1 fun <T> ValueMatcher<T>.reversed(): ValueMatcher<T> =
2     when (this) {
3         is ValueMatcher.EmptyList ->
4             ValueMatcher.NotEmptyList<T>()
5         is ValueMatcher.NotEmptyList ->
6             ValueMatcher.EmptyList<T>()
7         is ValueMatcher.Equal -> ValueMatcher.NotEqual(value)
8         is ValueMatcher.NotEqual -> ValueMatcher.Equal(value)
9     }
```

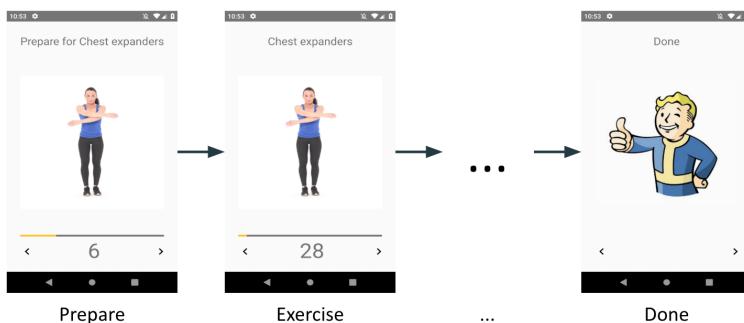
On the other hand, when we use `abstract` we leave the door open for other developers to create new instances outside of our control. In such a case, we should declare our functions as `abstract` and implement them in the subclasses, because if we use `when`, our function cannot work properly when new classes are added outside of our project.

Sealed modifier makes it easier to add new functions to the class using extensions or to handle different variants of this class. Abstract classes leave space for new classes joining this

hierarchy. If we want to control what are the subclasses of a class, we should use the `sealed` modifier. We use `abstract` mainly when we design for inheritance.

Tagged classes are not the same as state pattern

Tagged classes should not be confused with the *state pattern*, 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 (respectively from MVC, MVP, and MVVM architectures). For instance, let's say that you write an application for morning exercises. Before every exercise, there is preparation time, and in the end, there is a screen that states that you are done with the training.



When we use the state pattern, we have a hierarchy of classes representing different states, and a read-write property we use to represent which state is the current one:

```
1 sealed class WorkoutState
2
3 class PrepareState(val exercise: Exercise) :
4 WorkoutState()
5
6 class ExerciseState(val exercise: Exercise) :
7 WorkoutState()
8
9 object DoneState : WorkoutState()
10
11 fun List<Exercise>.toStates(): List<WorkoutState> =
12     flatMap { exercise ->
13         listOf(PrepareState(exercise),
14             ExerciseState(exercise))
15     } + DoneState
16
17 class WorkoutPresenter( /*...*/ ) {
18     private var state: WorkoutState = states.first()
19
20     //...
21 }
```

The difference here is that:

- The state is a part of a bigger class with more responsibilities
- The state changes

The state is generally kept in a single read-write property `state`. The concrete state is represented with an object, and we prefer this object to be a sealed class hierarchy instead of a tagged class. We also prefer it as an immutable object and whenever we need to change it, we change `state` property. It is not uncommon for `state` to later be observed to update the view every time it changes:

```
1 private var state: WorkoutState by
2     Delegates.observable(states.first()) { _, _, _ ->
3         updateView()
4     }
```

Summary

In Kotlin, we use type hierarchies instead of tagged classes. We most often represent those type hierarchies as sealed classes as they represent a sum type (a type collecting alternative class options). It doesn't collide with the state pattern, which is a popular and useful pattern in Kotlin. They actually cooperate as when we implement state, we prefer to use sealed hierarchies instead of tagged classes. This is especially true when we implement complex yet separable states on a single view.

Item 40: 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 elaborated in the official documentation, and as I described in *Item 32: Respect abstraction contracts*, every subtype of a type with a set contract should respect this contract. These methods have an important position in Kotlin, as they have been defined since the beginning of Java, and therefore many objects and functions depend on their contract. Breaking their contract will often lead to some objects or functions not working properly. This is why in the current and 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 by the `equals` method or `==` operator (and its negated counterpart `!=`) which is based on the `equals` method. `a == b` translates to `a.equals(b)` when `a` is not nullable, or otherwise to `a?.equals(b) ?: (b === null)`.
- Referential equality - checked by 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. Although using operators to check equality is not allowed when objects are not of the same type:

```
1 open class Animal
2 class Book
3 Animal() == Book() // Error: Operator == cannot be
4 // applied to Animal and Book
5 Animal() === Book() // Error: Operator === cannot be
6 // applied to Animal and Book
```

Objects either need to have the same type or one needs to be a subtype of another:

```
1 class Cat: Animal()
2 Animal() == Cat() // OK, because Cat is a subclass of
3 // Animal
4 Animal() === Cat() // OK, because Cat is a subclass of
5 // Animal
```

It is because it does not make sense to check equality of two objects of a different type. It will get clear when we will explain the contract of `equals`.

Why do we need `equals`?

The default implementation of `equals` coming from `Any` checks if another object is exactly the same instance. Just like referential equality (`==`). It means that every object by default is unique:

```
1 class Name(val name: String)
2 val name1 = Name("Marcin")
3 val name2 = Name("Marcin")
4 val name1Ref = name1
5
6 name1 == name1 // true
7 name1 == name2 // false
8 name1 == name1Ref // true
9
10 name1 === name1 // true
11 name1 === name2 // false
12 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 where we need to represent equality differently. A popular alternative is a data class equality that checks if all primary constructor properties are equal:

```
1 data class FullName(val name: String, val surname: String)
2 val name1 = FullName("Marcin", "Moskała")
3 val name2 = FullName("Marcin", "Moskała")
4 val name3 = FullName("Maja", "Moskała")
5
6 name1 == name1 // true
7 name1 == name2 // true, because data are the same
8 name1 == name3 // false
9
10 name1 === name1 // true
11 name1 === name2 // false
12 name1 === name3 // false
```

Such behavior is perfect for classes that are represented by the data they hold, and 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. For instance when we want to skip cache or other redundant properties. Here is an example of an object representing date and time having properties `asStringCache` and `changed` that should not be compared by equality check:

```
1 class DateTime(  
2     /** The millis from 1970-01-01T00:00:00Z */  
3     private var millis: Long = 0L,  
4     private var timeZone: TimeZone? = null  
5 ) {  
6     private var asStringCache = ""  
7     private var changed = false  
8  
9     override fun equals(other: Any?): Boolean =  
10        other is DateTime &&  
11            other.millis == millis &&  
12            other.timeZone == timeZone  
13  
14        //...  
15 }
```

The same can be achieved by using data modifier:

```
1 data class DateTime(  
2     private var millis: Long = 0L,  
3     private var timeZone: TimeZone? = null  
4 ) {  
5     private var asStringCache = ""  
6     private var changed = false  
7  
8     //...  
9 }
```

Just notice that copy in such case will not copy those properties that are not declared in the primary constructor. Such behavior is

correct only when those additional properties are truly redundant (the object will behave correctly when they will be lost).

Thanks to those two alternatives, default and data class equality, **we rarely need to implement equality ourselves in Kotlin**. Although there are cases where we need to implement equals ourselves.

Another example is when just concrete property decides if two objects are equal. For instance, a `User` class might have an assumption that two users are equal when their `id` is equal.

```
1  class User(
2      val id: Int,
3      val name: String,
4      val surname: String
5  ) {
6      override fun equals(other: Any?): Boolean =
7          other is User && other.id == id
8
9      override fun hashCode(): Int = id
10 }
```

As you can see, we implement `equals` ourselves when:

- We need its logic to differ from the default one
- We need to compare only a subset of properties
- We do not want our object to be a data class or 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 1.3.11, formatted):

Indicates whether some other object is “equal to” this one. Implementations must fulfill 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. It 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 those requirements are important. They are assumed from the beginning, also in Java, and so now many objects depend on those assumptions. Don't worry if they sound confusing right now, we'll describe them in detail.

- Object equality should be **reflexive**, meaning that `x.equals(x)` returns `true`. Sounds obvious, but this can be violated. For instance, someone might want to make a `Time` object that represents the current time, and compares milliseconds:

```
1 // DO NOT DO THIS!
2 class Time(
3     val millisArg: Long = -1,
4     val isNow: Boolean = false
5 ) {
6     val millis: Long get() =
7         if (isNow) System.currentTimeMillis()
8         else millisArg
9
10    override fun equals(other: Any?): Boolean =
11        other is Time && millis == other.millis
12 }
13
14 val now = Time(isNow = true)
15 now == now // Sometimes true, sometimes false
16 List(100000) { now }.all { it == now }
17 // 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. It will not work correctly in most unit tests assertions either.

```
1 val now1 = Time(isNow = true)
2 val now2 = Time(isNow = true)
3 assertEquals(now1, now2)
4 // Sometimes passes, sometimes not
```

When the result is not constant, we cannot trust it. We can never be sure if the result is correct or is it just a result of inconsistency.

How should we improve it? A simple solution is checking separately if the object represents the current time and if not, then

whether it has the same timestamp. Though it is a typical example of tagged class, and as described in *Item 39: Prefer class hierarchies to tagged classes*, it would be even better to use class hierarchy instead:

```
1 sealed class Time
2 data class TimePoint(val millis: Long): Time()
3 object Now: Time()
```

- Object equality should be **symmetric**, meaning that the result of `x == y` and `y == x` should always be the same. It can be easily violated when in our equality we accept objects of a different type. For instance, let's say that we implemented a class to represent complex numbers and made its equality accept `Double`:

```
1 class Complex(
2     val real: Double,
3     val imaginary: Double
4 ) {
5     // DO NOT DO THIS, violates symmetry
6     override fun equals(other: Any?): Boolean {
7         if (other is Double) {
8             return imaginary == 0.0 && real == other
9         }
10        return other is Complex &&
11            real == other.real &&
12            imaginary == other.imaginary
13    }
14 }
```

The problem is that `Double` does not accept equality with `Complex`. Therefore the result depends on the order of the elements:

```
1 Complex(1.0, 0.0).equals(1.0) // true
2 1.0.equals(Complex(1.0, 0.0)) // false
```

Lack of symmetry means, for instance, unexpected results on collections contains or on unit tests assertions.

```
1 val list = listOf<Any>(Complex(1.0, 0.0))
2 list.contains(1.0) // Currently on the JVM this is false,
3 // but it depends on the collection's implementation
4 // 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 the same (they assume symmetry). It can also change at any moment - creators during some refactoring might change the order. 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 equality.

The general solution is that we should not accept equality between different classes. I've never seen a case where it would be reasonable. Notice that in Kotlin similar classes are not equal to each other. `1` is not equal to `1.0`, and `1.0` is not equal to `1.0F`. Those are different types and they are not even comparable. Also, in Kotlin we cannot use the `==` operator between two different types that do not have a common superclass other than `Any`:

```
1 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:

```
1  open class Date(
2      val year: Int,
3      val month: Int,
4      val day: Int
5  ) {
6      // DO NOT DO THIS, symmetric but not transitive
7      override fun equals(o: Any?): Boolean = when (o) {
8          is DateTime -> this == o.date
9          is Date -> o.day == day && o.month == month &&
10         o.year == year
11         else -> false
12     }
13
14     // ...
15 }
16
17 class DateTime(
18     val date: Date,
19     val hour: Int,
20     val minute: Int,
21     val second: Int
22 ): Date(date.year, date.month, date.day) {
23     // DO NOT DO THIS, symmetric but not transitive
24     override fun equals(o: Any?): Boolean = when (o) {
25         is DateTime -> o.date == date && o.hour == hour &&
26         o.minute == minute && o.second == second
27         is Date -> date == o
28         else -> false
29     }
30 }
```

```
31     // ...
32 }
```

The problem with the above implementation is that when we compare two `DateTime`, we check more properties than when we compare `DateTime` and `Date`. Therefore two `DateTime` 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:

```
1 val o1 = DateTime(Date(1992, 10, 20), 12, 30, 0)
2 val o2 = Date(1992, 10, 20)
3 val o3 = DateTime(Date(1992, 10, 20), 14, 45, 30)
4
5 o1 == o2 // true
6 o2 == o3 // true
7 o1 == o3 // false <- So equality is not transitive
```

Notice that here the restriction to compare only objects of the same type didn't help 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, do not compare two objects of different types. These classes are perfect examples of objects holding data and representing them this way is a good choice:

```
1  data class Date(
2    val year: Int,
3    val month: Int,
4    val day: Int
5  )
6
7  data class DateTime(
8    val date: Date,
9    val hour: Int,
10   val minute: Int,
11   val second: Int
12 )
13
14 val o1 = DateTime(Date(1992, 10, 20), 12, 30, 0)
15 val o2 = Date(1992, 10, 20)
16 val o3 = DateTime(Date(1992, 10, 20), 14, 45, 30)
17
18 o1.equals(o2) // false
19 o2.equals(o3) // false
20 o1 == o3 // false
21
22 o1.date.equals(o2) // true
23 o2.equals(o3.date) // true
24 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 those objects was modified. For immutable objects, the result should be always the same. In other words, we expect `equals` to be a pure function (do not modify the state of an object) for which result always depends only on input and state of its receiver. We've seen the `Time` class that violated this principle. This rule was also famously violated in `java.net.URL.equals()`.
- Never equal to null: for any non-null value `x`, `x.equals(null)`

must return `false`. It is important because `null` should be unique and no object should be equal to it.

Problem with equals in URL

One example of a really poorly designed `equals` is the one from `java.net.URL`. Equality of two `java.net.URL` objects depends on a network operation as two hosts are considered equivalent if both hostnames can be resolved into the same IP addresses. Take a look at the following example:

```
1 import java.net.URL  
2  
3 fun main() {  
4     val enWiki = URL("https://en.wikipedia.org/")  
5     val wiki = URL("https://wikipedia.org/")  
6     println(enWiki == wiki)  
7 }
```

The result is not consistent. In normal conditions, it should print `true` because those two addresses are considered equal according to `equals` implementation, although if you have your internet turned off, it will print `false`. You can check it 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 a network is available and unequal when it is not. Also, the network may change. The IP address for a given hostname varies over time and by the network. Two URLs could be equal on some networks and unequal on others.
- **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 on the list. Also on some platforms, like Android, network operations are prohibited on the main thread. As a result, even adding to a set of URLs 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 permits 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 default or data class equality. If you do need custom equality, always consider if your implementation is reflexive, symmetric, transitive, and consistent. Make such class final, or beware that subclasses should not change how equality behaves. It is hard to make custom equality and support inheritance at the same time. Some even say it is impossible⁵³. Data classes are final.

⁵³As 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.”.

Item 41: 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 *hash table*, which is used in a variety of different collections or algorithms under the hood.

Hash table

Let's start with the problem 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 map, neither of which allow for 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 for checking if it contains an element, because to check that we need to compare this element with all elements on 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 will be really time-consuming to compare your text one after another with those 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
- Ideally 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. Those buckets are kept in a structure called hash table, which is an array with a size 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 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. 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 1,000 buckets, searching for duplicates only requires to compare about 1,000 elements on average, and the performance cost of this improvement is really small.

To see a more concrete example, let's say that we have the following strings and a hash function that splits 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 the following hash table built:

Index	Object to which 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”]

Index	Object to which hash table points
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 on 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 also in standard library collections in many languages. In Kotlin/JVM both the default set (`LinkedHashSet`) and default map (`LinkedHashMap`) use it. To produce a hash code, we use the `hashCode` function⁵⁴.

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. This is why both `LinkedHashSet` and `LinkedHashMap` key will not behave properly when an object mutates after it has been added:

⁵⁴Often, after some transformations, as `hashCode` returns Int, that is 32-bit signed integer, meaning 4294967296 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 makes the hash table bigger through a transformation and assigning all elements to new buckets.

```
1 data class FullName(  
2     var name: String,  
3     var surname: String  
4 )  
5  
6 val person = FullName("Maja", "Markiewicz")  
7 val s = mutableSetOf<FullName>()  
8 s.add(person)  
9 person.surname = "Moskała"  
10 print(person) // FullName(name=Maja, surname=Moskała)  
11 print(person in s) // false  
12 print(s.first() == person) // true
```

This problem was already noted on *Item 1: Limit mutability*: Mutable objects are not to be used in data structures based on hashes or on 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

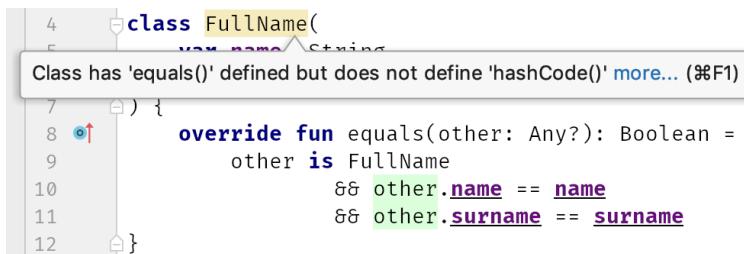
Knowing what we need `hashCode` for, it should be clear how we expect it to behave. The formal contract is as follows (Kotlin 1.3.11):

- 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 one is the one that developers often forget about, and that needs to be highlighted: `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 using a hash table under the hood:

```
1 class FullName(
2     var name: String,
3     var surname: String
4 ) {
5     override fun equals(other: Any?): Boolean =
6         other is FullName
7             && other.name == name
8             && other.surname == surname
9 }
10
11 val s = mutableSetOf<FullName>()
12 s.add(FullName("Marcin", "Moskała"))
13 val p = FullName("Marcin", "Moskała")
14 print(p in s) // false
15 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 required, but very important if we want this function to be useful. `hashCode` should spread

elements as widely as possible. Different elements should have the highest possible probability of having different hash values.

Think about what happens when many different 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 is fulfilling 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 added a counter that counts how many times it was used. You can see that when we operate on sets with values of both types, the second one, named `Terrible`, requires many more comparisons:

```
1 class Proper(val name: String) {
2
3     override fun equals(other: Any?): Boolean {
4         equalsCounter++
5         return other is Proper && name == other.name
6     }
7
8     override fun hashCode(): Int {
9         return name.hashCode()
10    }
11
12    companion object {
13        var equalsCounter = 0
14    }
15 }
16
17 class Terrible(val name: String) {
18     override fun equals(other: Any?): Boolean {
19         equalsCounter++
```

```
20     return other is Terrible && name == other.name
21 }
22
23 // Terrible choice, DO NOT DO THAT
24 override fun hashCode() = 0
25
26 companion object {
27     var equalsCounter = 0
28 }
29 }
30
31 val properSet = List(10000) { Proper("9999") }.toSet()
32 println(Precise.equalsCounter) // 0
33 val terribleSet = List(10000) { Terrible("A") }.toSet()
34 println(Terrible.equalsCounter) // 50116683
35
36 Proper.equalsCounter = 0
37 println(Precise("A") in properSet) // true
38 println(Precise.equalsCounter) // 1
39
40 Proper.equalsCounter = 0
41 println(Precise("A") in properSet) // false
42 println(Precise.equalsCounter) // 0
43
44 Terrible.equalsCounter = 0
45 println(Terrible("A") in terribleSet) // true
46 println(Terrible.equalsCounter) // 4324
47
48 Terrible.equalsCounter = 0
49 println(Terrible("A") in terribleSet) // false
50 println(Terrible.equalsCounter) // 10001
```

Implementing hashCode

We define `hashCode` in Kotlin practically only when we define 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. When you have a custom `equals`, implement `hashCode` that always returns the same value for equal elements.

If you implemented typical `equals` that checks equality of significant properties, then a typical `hashCode` should be calculated using the hash codes of those properties. How can we make a single hash code out of those many hash codes? A typical way is that we 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 its characteristics make it a good number 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`:

```
1 class DateTime(
2     private var millis: Long = 0L,
3     private var timeZone: TimeZone? = null
4 ) {
5     private var asStringCache = ""
6     private var changed = false
7
8     override fun equals(other: Any?): Boolean =
9         other is DateTime &&
10            other.millis == millis &&
11            other.timeZone == timeZone
12
13    override fun hashCode(): Int {
14        var result = millis.hashCode()
```

```
15     result = result * 31 + timeZone.hashCode()
16     return result
17 }
18 }
```

One helpful function on Kotlin/JVM is `Objects.hashCode` that calculates hash for us:

```
1 override fun hashCode(): Int =
2     Objects.hash(timeZone, millis)
```

There is no such function in the Kotlin stdlib, but if you need it on other platforms you can implement it yourself:

```
1 override fun hashCode(): Int =
2     hashCodeFrom(timeZone, millis)
3
4 inline fun hashCodeOf(vararg values: Any?) =
5     values.fold(0) { acc, value ->
6         (acc * 31) + value.hashCode()
7     }
```

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 `data` modifier:

```
1 data class DateTime2(  
2     private var millis: Long = 0L,  
3     private var timeZone: TimeZone? = null  
4 ) {  
5     private var asStringCache = ""  
6     private var changed = false  
7 }
```

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.

Item 42: 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 inequality signs:

```
1 obj1 > obj2 // Translates to obj1.compareTo(obj2) > 0
2 obj1 < obj2 // Translates to obj1.compareTo(obj2) < 0
3 obj1 >= obj2 // Translates to obj1.compareTo(obj2) >= 0
4 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`, it means that this object has a natural order. Such an order needs to be:

- **Antisymmetric**, meaning that 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 that 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 without it, sorting of elements might take literally forever in some sorting algorithms.
- **Connex**, meaning that there must be a relationship between every two elements. So either $a \geq b$, or $b \geq a$. In Kotlin, it is guaranteed by typing system for `compareTo` because it 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 surname:

```
1 class User(val name: String, val surname: String)
2 val names = listOf<User>(/*...*/)
3
4 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 that sorts elements using a comparator. This comparator can be produced using a function `compareBy`. So in the following example, we first sort users comparing them by their `surname`, and if they match, we compare them by their `name`:

```
1 val sorted = names
2     .sortedWith(compareBy({ it.surname }, { it.name }))
```

Surely, we might make `User` implement `Comparable<User>`, but what order should it choose? We might need to sort them by any property. When this is not absolutely clear, it is better to not make such objects comparable.

`String` has a natural order, which is an alphanumerical order, and so it implements `Comparable<String>`. This fact is very useful because we often do need to sort text alphanumerically. However, it also has its downsides. For instance, we can compare two strings using an inequality sign, which seems highly unintuitive. Most people seeing two strings comparison using inequality signs will be rather confused.

```
1 // DON'T DO THIS!
2 print("Kotlin" > "Java") // true
```

Surely there are objects with a clear natural order. Units of measure, date and time are all perfect examples. Although 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:

```
1 class User(val name: String, val surname: String) {
2     // ...
3
4     companion object {
5         val DISPLAY_ORDER =
6             compareBy(User::surname, User::name)
7     }
8 }
9
10 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:

```
1 class User(
2     val name: String,
3     val surname: String
4 ): Comparable<User> {
5     override fun compareTo(other: User): Int =
6         compareValues(surname, other.surname)
7 }
```

If you need to use more values, or if you need to compare them using selectors, use `compareValuesBy`:

```
1 class User(  
2     val name: String,  
3     val surname: String  
4 ): Comparable<User> {  
5     override fun compareTo(other: User): Int =  
6         compareValuesBy(this, other, { it.surname }, {  
7             it.name })  
8 }
```

This function helps us create most comparators we might need. If you need to implement some 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`

Once you did that, don't forget to verify that your comparison is antisymmetric, transitive and connex.

Item 43: Consider extracting non-essential parts of your API into extensions

When we define final methods in a class, we need to make a decision if we want to define them as members, or if we want to define them as extension functions.

```
1 // Defining methods as members
2 class Workshop(/*...*/) {
3     //...
4
5     fun makeEvent(date: DateTime): Event = //...
6
7     val permalink
8         get() = "/workshop/$name"
9 }
```

```
1 // Defining methods as extensions
2 class Workshop(/*...*/) {
3     //...
4
5     fun Workshop.makeEvent(date: DateTime): Event = //...
6
7     val Workshop_permalink
8         get() = "/workshop/$name"
```

Both approaches are similar in many ways. Their use and even referencing them via reflection is very similar:

```
1 fun useWorkshop(workshop: Workshop) {  
2     val event = workshop.makeEvent(date)  
3     val permalink = workshop permalink  
4  
5     val makeEventRef = Workshop::makeEvent  
6     val permalinkPropRef = Workshop::permalink  
7 }
```

There are also significant differences though. They both have their pros and cons, and one way does not dominate over another. This is why my suggestion is to consider such extraction, not necessarily to do it. The point is to make smart decisions, and for that, we need to understand the differences.

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 public API of the class.

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 and we won't have a conflict. On the other hand, it would be dangerous to have two extensions with the same name, but having different behavior. For such cases, we can cut the Gordian knot by making a member function. The compiler always chooses member functions over extensions⁵⁵.

Another significant difference is that **extensions are not virtual**, meaning that they cannot be redefined in derived classes. The extension function to call is selected statically during compilation.

⁵⁵The only exception is when an extension in the Kotlin stdlib has `kotlin.internal.HidesMembers` internal annotation

This is different behavior from member elements that are virtual in Kotlin. Therefore we should not use extensions for elements that are designed for inheritance.

```
1 open class C
2 class D: C()
3 fun C.foo() = "c"
4 fun D.foo() = "d"
5
6 fun main() {
7     val d = D()
8     print(d.foo()) // d
9     val c: C = d
10    print(c.foo()) // c
11
12    print(D().foo()) // d
13    print((D() as C).foo()) // c
14 }
```

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:

```
1 fun foo(`this$receiver`: C) = "c"
2 fun foo(`this$receiver`: D) = "d"
3
4 fun main() {
5     val d = D()
6     print(foo(d)) // d
7     val c: C = d
8     print(foo(c)) // c
9
10    print(foo(D())) // d
11    print(foo(D() as C)) // c
12 }
```

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 a concrete substitution of a generic type:

```
1 inline fun CharSequence?.isNullOrBlank(): Boolean {
2     contract {
3         returns(false) implies (this@isNullOrBlank != null)
4     }
5
6     return this == null || this.isBlank()
7 }
8
9 public fun Iterable<Int>.sum(): Int {
10     var sum: Int = 0
11     for (element in this) {
12         sum += element
13     }
14     return sum
15 }
```

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 why, when we process a class using annotation processing, we cannot extract elements that should be processed into extension functions. On the other hand, if we extract non-essential elements into extensions, we don't need to worry about them being seen by those processors. We don't need to hide them, because they are not in the class anyway.

Summary

The most important differences between members and extensions are:

- Extensions need to be imported
- Extensions are not virtual
- Member has a higher priority
- Extensions are on a type, not on a class
- Extensions are not listed in the class reference

To summarize it, extensions give us more freedom and flexibility. They are more noncommittal. Although they do not support inheritance, annotation processing, and it might be confusing that they are not present in the class. Essential parts of our API should rather stay as members, but there are good reasons to extract non-essential parts of your API as extensions.

Item 44: 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, 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:

```
1 fun String.isPhoneNumber(): Boolean =  
2     length == 7 && all { it.isDigit() }
```

Under the hood is compiled to a function similar to this one:

```
1 fun isPhoneNumber(`$this`: String): Boolean =  
2     `$this`.length == 7 && `$this`.all { it.isDigit() }
```

One of the consequences of how they are implemented is that we can have member extensions or even define extensions in interfaces:

```
1 interface PhoneBook {  
2     fun String.isPhoneNumber(): Boolean  
3 }  
4  
5 class Fizz: PhoneBook {  
6     override fun String.isPhoneNumber(): Boolean =  
7         length == 7 && all { it.isDigit() }  
8 }
```

Even though it is possible, there are good reasons to avoid defining member extensions (except for DSLs). **Especially, do not define extension as members just to restrict visibility.**

```
1 // Bad practice, do not do this
2 class PhoneBookIncorrect {
3     // ...
4
5     fun String.isPhoneNumber() =
6         length == 7 && all { it.isDigit() }
7 }
```

One big reason is that it does not really restrict visibility. It only makes it more complicated to use the extension function since the user would need to provide both the extension and dispatch receivers:

```
1 PhoneBookIncorrect().apply { "1234567890".test() }
```

You should restrict the extension visibility using a visibility modifier and not by making it a member.

```
1 // This is how we limit extension functions visibility
2 class PhoneBookCorrect {
3     // ...
4
5     private fun String.isPhoneNumber() =
6         length == 7 && all { it.isDigit() }
7 }
```

There are a few good reasons why we prefer to avoid member extensions:

- Reference is not supported:

```

1 val ref = String::isPhoneNumber
2 val str = "1234567890"
3 val boundedRef = str::isPhoneNumber
4
5 val refX = PhoneBookIncorrect::isPhoneNumber // ERROR
6 val book = PhoneBookIncorrect()
7 val boundedRefX = book::isPhoneNumber // ERROR

```

- Implicit access to both receivers might be confusing:

```

1 class A {
2     val a = 10
3 }
4 class B {
5     val a = 20
6     val b = 30
7
8     fun A.test() = a + b // Is it 40 or 50?
9 }

```

- When we expect an extension to modify or reference a receiver, it is not clear if we modify the extension or dispatch receiver (the class in which the extension is defined):

```

1 class A {
2     //...
3 }
4 class B {
5     //...
6
7     fun A.update() = ... // Shell is update A or B?
8 }

```

- For less experienced developers it might be counterintuitive or scary to see member extensions.

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 indulgently. To a certain degree, it is reasonable. Memory is cheap and developers are expensive. Though if your application is running on millions of devices, it consumes a lot of energy, and some optimization of battery use might save enough energy to power a small city. Or maybe your company is paying lots of money for servers and their maintenance, and some optimization might make it significantly cheaper. Or maybe your application works well for a small number of requests but does not scale well and on the day of the trial, it shuts down. 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. Those are the cheap wins: they cost nearly nothing, but still, they can 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 more 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.

They focus on general rules to cheaply optimize every-day development. But they also give some Kotlin-specific suggestions on

how performance might be optimized in the critical parts of your program. They should also deepen your understanding of thinking about performance in general.

Please, remember that when there is a tradeoff between readability and performance, you need to answer yourself what is more important in the component you develop. I included some suggestions, but there is no universal answer.

Item 45: Avoid unnecessary object creation

Object creation can sometimes be expensive and always costs something. 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 any other code running in the same virtual machine that happens to contain the same string literal⁵⁶:

```
1 val str1 = "Lorem ipsum dolor sit amet"
2 val str2 = "Lorem ipsum dolor sit amet"
3 print(str1 == str2) // true
4 print(str1 === str2) // true
```

Boxed primitives (Integer, Long) are also reused in JVM when they are small (by default Integer Cache holds numbers in a range from -128 to 127).

```
1 val i1: Int? = 1
2 val i2: Int? = 1
3 print(i1 == i2) // true
4 print(i1 === i2) // true, because i2 was taken from cache
```

Reference equality shows that this is the same object. If we use number that is either smaller than -128 or bigger than 127 though, different objects will be produced:

⁵⁶Java Language Specification, Java SE 8 edition, 3.10.5

```
1 val j1: Int? = 1234
2 val j2: Int? = 1234
3 print(j1 == j2) // true
4 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 when we make it nullable or when we use it as a type argument, `Integer` is used instead. It is because primitive cannot be `null` and cannot be used as a type argument. Knowing that such mechanisms were introduced in the language, you might wonder how significant they are. Is object creation expensive?

Is object creation expensive?

Wrapping something into an object has 3 parts of cost:

- **Objects take additional space.** In a modern 64-bit JDK, an object has a 12-byte header, 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 take space as well. Typically, references are 4 bytes on 32bit platforms or on 64bit platforms up to `-Xmx32G`, and 8 bytes above `32Gb` (`-Xmx32G`). Those are relatively small numbers, but they can add up to a significant cost. When we think about such small elements like integers, they make a difference. `Int` as a primitive fit in 4 bytes, when as a wrapped type on 64-bit JDK we mainly use today, it requires 16 bytes (it fits in the 4 bytes after the header) + its reference requires 4 or 8 bytes. In the end, it takes 5 or 6 times more space⁵⁷.
- **Access requires an additional function call when elements are encapsulated.** That is again a small cost as function use

⁵⁷To measure the size of concrete fields on JVM objects, use Java Object Layout.

is really fast, but it can add-up when we need to operate on a huge pool of objects. Do not limit encapsulation, avoid creating unnecessary objects especially in performance critical parts of your code.

- **Objects need to be created.** An object needs to be created, allocated in the memory, a reference needs to be created, etc. It is also a really small cost, but it can add up.

```
1  class A
2  private val a = A()
3
4  // Benchmark result: 2.698 ns/op
5  fun accessA(blackhole: Blackhole) {
6      blackhole.consume(a)
7  }
8
9  // Benchmark result: 3.814 ns/op
10 fun createA(blackhole: Blackhole) {
11     blackhole.consume(A())
12 }
13
14 // Benchmark result: 3828.540 ns/op
15 fun createListAccessA(blackhole: Blackhole) {
16     blackhole.consume(List(1000) { a })
17 }
18
19 // Benchmark result: 5322.857 ns/op
20 fun createListCreateA(blackhole: Blackhole) {
21     blackhole.consume(List(1000) { A() })
22 }
```

By eliminating objects, we can avoid all three costs. By reusing objects, we can eliminate the first and the third one. Knowing that, you might start thinking about limiting the number of unnecessary objects in your code. Let's see some ways we can do that.

Object declaration

A very simple way to reuse an object instead of creating it every time is using object declaration (singleton). To see an example, let's imagine that you need to implement a linked list. The linked list can be either empty, or it can be a node containing an element and pointing to the rest. This is how it can be implemented:

```
1 sealed class LinkedList<T>
2
3 class Node<T>(
4     val head: T,
5     val tail: LinkedList<T>
6 ): LinkedList<T>()
7
8 class Empty<T>: LinkedList<T>()
9
10 // Usage
11 val list: LinkedList<Int> =
12     Node(1, Node(2, Node(3, Empty())))
13 val list2: LinkedList<String> =
14     Node("A", Node("B", Empty()))
```

One problem with this implementation is that we need to create an instance of `Empty` every time we create a list. Instead, we should just have one instance and use it universally. The only problem is the generic type. What generic type should we use? We want this empty list to be subtype of all lists. We cannot use all types, but we also don't need to. A solution is that we can make it a list of `Nothing`. `Nothing` is a subtype of every type, and so `LinkedList<Nothing>` will be a subtype of every `LinkedList` once this list is covariant (`out` modifier). Making type arguments covariant truly makes sense here since the list is immutable and this type is used only in out positions (Item 24: Consider variance for generic types). So this is the improved code:

```
1 sealed class LinkedList<out T>
2
3 class Node<out T>(
4     val head: T,
5     val tail: LinkedList<T>
6 ) : LinkedList<T>()
7
8 object Empty : LinkedList<Nothing>()
9
10 // Usage
11 val list: LinkedList<Int> =
12     Node(1, Node(2, Node(3, Empty)))
13
14 val list2: LinkedList<String> =
15     Node("A", Node("B", Empty))
```

This is a useful trick that is often used, especially when we define immutable sealed classes. Immutable, because using it for mutable objects can lead to subtle and hard to detect bugs connected to shared state management. The general rule is that mutable objects should not be cached (*Item 1: Limit mutability*). Apart from object declaration, there are more ways to reuse objects. Another one is a factory function with a cache.

Factory function with a cache

Every time we use a constructor, we have a new object. Though it is not necessarily true when you use a factory method. Factory functions can have cache. The simplest case is when a factory function always returns the same object. This is, for instance, how `emptyList` from stdlib is implemented:

```
1 fun <T> emptyList(): List<T> = EmptyList
```

Sometimes we have a set of objects, and we return one of them. For instance, when we use the default dispatcher in the Kotlin

coroutines library `Dispatchers.Default`, it has a pool of threads, and whenever we start anything using it, it will start it in one that is not in use. Similarly, we might have a pool of connections with a database. Having a pool of objects is a good solution when object creation is heavy, and we might need to use a few mutable objects at the same time.

Caching can also be done for parameterized factory methods. In such a case, we might keep our objects in a map:

```
1 private val connections: MutableMap<String, Connection> =
2     mutableMapOf<String, Connection>()
3
4 fun getConnection(host: String) =
5     connections.getOrPut(host) { createConnection(host) }
```

Caching can be used for all pure functions. In such a case, we call it memorization. Here is, for instance, a function that calculates the Fibonacci number at a position based on the definition:

```
1 private val FIB_CACHE: MutableMap<Int, BigInteger> =
2     mutableMapOf<Int, BigInteger>()
3
4 fun fib(n: Int): BigInteger = FIB_CACHE.getOrPut(n) {
5     if (n <= 1) BigInteger.ONE else fib(n - 1) + fib(n - 2)
6 }
```

Now our method during the first run is nearly as efficient as a linear solution, and later it gives result immediately if it was already calculated. Comparison between this and classic linear fibonacci implementation on an example machine is presented in the below table. Also the iterative implementation we compare it to is presented below.

	n = 100	n = 200	n = 300	n = 400
fibIter	1997 ns	5234 ns	7008 ns	9727 ns
fib (first)	4413 ns	9815 ns	15484 ns	22205 ns
fib (later)	8 ns	8 ns	8 ns	8 ns

```

1 fun fibIter(n: Int): BigInteger {
2     if(n <= 1) return BigInteger.ONE
3     var p = BigInteger.ONE
4     var pp = BigInteger.ONE
5     for (i in 2..n) {
6         val temp = p + pp
7         pp = p
8         p = temp
9     }
10    return p
11 }
```

You can see that using this function for the first time is slower than using the classic approach as there is additional overhead on checking if the value is in the cache and adding it there. Once values are added, the retrieval is nearly instantaneous.

It has a significant drawback though: we are reserving and using more memory since the Map needs to be stored somewhere. Everything would be fine if this was cleared at some point. But take into account that for the Garbage Collector (GC), there is no difference between a cache and any other static field that might be necessary in the future. It will hold this data as long as possible, even if we never use the `fib` function again. One thing that helps is using a soft reference that can be removed by the GC when memory is needed. It should not be confused with weak reference. In simple words, the difference is:

- Weak references do not prevent Garbage Collector from cleaning-up the value. So once no other reference (variable) is using it, the value will be cleaned.

- Soft references are not guaranteeing that the value won't be cleaned up by the GC either, but in most JVM implementations, this value won't be cleaned unless memory is needed. Soft references are perfect when we implement a cache.

This is an example property delegate (details in *Item 21: Use property delegation to extract common property patterns*) that on-demand creates a map and lets us use it, but does not stop Garbage Collector from recycling this map when memory is needed (full implementation should include thread synchronization):

```
1 private val FIB_CACHE: MutableMap<Int, BigInteger> by
2     SoftReferenceDelegate { mutableMapOf<Int, BigInteger>() }
3
4 fun fib(n: Int): BigInteger = FIB_CACHE.getOrPut(n) {
5     if (n <= 1) BigInteger.ONE else fib(n - 1) + fib(n - 2)
6 }
7
8 class SoftReferenceDelegate<T: Any>(
9     val initialization: ()->T
10 ) {
11     private var reference: SoftReference<T>? = null
12
13     operator fun getValue(
14         thisRef: Any?,
15         property: KProperty<*>
16     ): T {
17         val stored = reference?.get()
18         if (stored != null) return stored
19         val new = initialization()
20         reference = SoftReference(new)
21         return new
22     }
23 }
```

Designing a cache well is not easy, and in the end, caching is

always a tradeoff: performance for memory. Remember this, and use caches wisely. No-one wants to move from performance issues to lack of memory issues.

Heavy object lifting

A very useful trick for performance is lifting a heavy object to an outer scope. Surely, we should lift if possible all heavy operations from collection processing functions to a general processing scope. For instance, in this function we can see that we will need to find the maximal element for every element in the Iterable:

```
1 fun <T: Comparable<T>> Iterable<T>.countMax(): Int =  
2     count { it == this.max() }
```

A better solution is to extract the maximal element to the level of countMax function:

```
1 fun <T: Comparable<T>> Iterable<T>.countMax(): Int {  
2     val max = this.max()  
3     return count { it == max }  
4 }
```

This solution is better for performance because we will not need to find the maximal element on the receiver on every iteration. Notice that it also improves readability by making it visible that max is called on the extension receiver and so it is the same through all iterations.

Extracting a value calculation to an outer scope to not calculate it is an important practice. It might sound obvious, but it is not always so clear. Just take a look at this function where we use a regex to validate if a string contains a valid IP address:

```
1 fun String.isValidIpAddress(): Boolean {
2     return this.matches("\\"A(?:(?:25[0-5]|2[0-4])[0-9]
3 | [01]?[0-9][0-9]?)\\\.){3}(?:25[0-5]|2[0-4][0-9]| [01]
4 ?[0-9][0-9]?)\\z".toRegex())
5 }
6
7 // Usage
8 print("5.173.80.254".isValidIpAddress()) // true
```

The problem with this function is that `Regex` object needs to be created every time we use it. It is a serious disadvantage since regex pattern compilation is a complex operation. This is why this function is not suitable to be used repeatedly in performance-constrained parts of our code. Though we can improve it by lifting regex up to the top-level:

```
1 private val IS_VALID_EMAIL_REGEX = "\\"A(?:(?:25[0-5]
2 | 2[0-4][0-9]| [01]?[0-9][0-9]?)\\\.){3}(?:25[0-5]|2[0-4]
3 | [01]?[0-9][0-9]?)\\z".toRegex()
4
5 fun String.isValidIpAddress(): Boolean =
6     matches(IS_VALID_EMAIL_REGEX)
```

If this function is in a file together with some other functions and we don't want to create this object if it is not used, we can even initialize the regex lazily:

```
1 private val IS_VALID_EMAIL_REGEX by lazy {
2     "\\"A(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\\\.)
3 {3}(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\\z".toRegex()
4 }
```

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 that lazily. For instance, imagine that class A needs instances of B, C, and D that are heavy. If we just create them during class creation, A creation will be very heavy, because it will need to create B, C and D and then the rest of its body. The heaviness of objects creation will just accumulates.

```
1  class A {  
2      val b = B()  
3      val c = D()  
4      val d = D()  
5  
6      //...  
7 }
```

There is a cure though. We can just initialize those heavy objects lazily:

```
1  class A {  
2      val b by lazy { B() }  
3      val c by lazy { C() }  
4      val d by lazy { D() }  
5  
6      //...  
7 }
```

Each object will then be initialized just before its first usage. The cost of those objects creation will be spread instead of accumulated.

Keep in mind that this sword is double-edged. You might have a case where object creation can be heavy, but you need methods to be as fast as possible. Imagine that A is a controller in a backend application that responds to HTTP calls. It starts quickly, but the

first call requires all heavy objects initialization. So the first call needs to wait significantly longer for the response, and it doesn't matter for how long our application runs. This is not the desired behavior. This is also something that might clutter our performance tests.

Using primitives

In JVM we have a special built-in type to represent basic elements like number or character. They are called primitives, and they are used by Kotlin/JVM compiler under the hood wherever possible. Although there are some cases where a wrapped class 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 the type as a generic

So in short:

Kotlin type	Java type
<code>Int</code>	<code>int</code>
<code>Int?</code>	<code>Integer</code>
<code>List<Int></code>	<code>List<Integer></code>

Knowing 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 flavours of Kotlin/Native. Not at all on Kotlin/JS. It also needs to be remembered that it makes sense only when operations on a number are repeated many times. Access of both primitive and wrapped types are relatively really fast compared to other operations. The difference manifests itself when we deal with a really big collections (we will discuss it in the *Item 51: 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 our code and in libraries.** You can find out what is performance critical using a profiler.

To see an example, imagine that you implement a standard library for Kotlin, and you want to introduce a function that will return the maximal element or `null` if this iterable is empty. You don't want to iterate over the iterable more than once. This is not a trivial problem, but it can be solved with the following function:

```
1 fun Iterable<Int>.maxOrNull(): Int? {  
2     var max: Int? = null  
3     for (i in this) {  
4         max = if(i > (max ?: Int.MIN_VALUE)) i else max  
5     }  
6     return max  
7 }
```

This implementation has serious disadvantages:

1. We need to use an Elvis operator in every step
2. We use a nullable value, so under the hood in JVM, there will be an `Integer` instead of an `int`.

Resolving these two problems requires us to implement the iteration using while loop:

```
1 fun Iterable<Int>.maxOrNull(): Int? {  
2     val iterator = iterator()  
3     if (!iterator.hasNext()) return null  
4     var max: Int = iterator.next()  
5     while (iterator.hasNext()) {  
6         val e = iterator.next()  
7         if (max < e) max = e  
8     }  
9     return max  
10 }
```

For a collection of elements from 1 to 10 million, in my computer, the optimized implementation took 289 ms, while the previous one took 518 ms. This is nearly 2 times faster, but remember that this is an extreme case designed to show the difference. Such optimization is rarely reasonable in a code that is not performance-critical. Though if you implement a Kotlin standard library, everything is performance critical. This is why the second approach is chosen here:

```
1 /**  
2 * Returns the largest element or `null` if there are  
3 * no elements.  
4 */  
5 public fun <T : Comparable<T>> Iterable<T>.max(): T? {  
6     val iterator = iterator()  
7     if (!iterator.hasNext()) return null  
8     var max = iterator.next()  
9     while (iterator.hasNext()) {  
10         val e = iterator.next()  
11         if (max < e) max = e  
12     }  
13     return max  
14 }
```

Summary

In this item, you've seen different ways to avoid object creation. Some of them are cheap in terms of readability: those should be used freely. For instance, heavy object lifting out of a loop or function is generally a good idea. It is a good practice for performance, and also thanks to this extraction, we can name this object so we make our function easier to read. Those optimizations that are tougher or require bigger changes should possibly be skipped. We should avoid premature optimization unless we have such guidelines in our project or we develop a library that might be used in who-knows-what-way. We've also learned some optimizations that might be used to optimize the performance critical parts of our code.

Item 46: Use inline modifier for functions with parameters of functional types

You might have noticed that nearly all Kotlin stdlib higher-order functions have an inline modifier. Have you ever asked yourself why are they defined in this way? Here is, for instance, how the `repeat` function from Kotlin stdlib is implemented:

```
1 inline fun repeat(times: Int, action: (Int) -> Unit) {
2     for (index in 0 until times) {
3         action(index)
4     }
5 }
```

What this `inline` modifier does is that during compilation, all uses of this function are replaced with its body. Also, all calls of function arguments inside `repeat` are replaced with those functions bodies. So the following `repeat` function call:

```
1 repeat(10) {
2     print(it)
3 }
```

During compilation will be replaced with the following:

```
1 for (index in 0 until 10) {
2     print(index)
3 }
```

It 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 where the

function was invoked. Replacing calls with bodies is a significantly different behavior.

There are a few advantages to 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 both all the advantages and costs of `inline` modifier.

A type argument can be reified

Java does not have generics in older versions. They were added to the Java programming language in 2004 within version J2SE 5.0. They are still not present in the JVM bytecode though. Therefore during compilation, generic types are erased. For instance `List<Int>` compiles to `List`. This is why we cannot check if an object is `List<Int>`. We can only check it is a `List`.

```
1 any is List<Int> // Error  
2 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:

```
1 fun <T> printTypeName() {  
2     print(T::class.simpleName) // ERROR  
3 }
```

We can overcome this limitation by making a function inline. Function calls are replaced with its body, so type parameters uses can be replaced with type arguments, by using the *reified* modifier:

```
1 inline fun <reified T> printTypeName() {  
2     print(T::class.simpleName)  
3 }  
4  
5 // Usage  
6 printTypeName<Int>() // Int  
7 printTypeName<Char>() // Char  
8 printTypeName<String>() // String
```

During compilation, the body of `printTypeName` replaces usages, and the type argument replaces the type parameter:

```
1 print(Int::class.simpleName) // Int  
2 print(Char::class.simpleName) // Char  
3 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:

```
1 class Worker
2 class Manager
3
4 val employees: List<Any> =
5     listOf(Worker(), Manager(), Worker())
6
7 val workers: List<Worker> =
8     employees.filterIsInstance<Worker>()
```

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 to track the back-stack. This is why small functions that are used very often in the stdlib are often inlined:

```
1 inline fun print(message: Any?) {
2     System.out.print(message)
3 }
```

This difference is most likely insignificant when a function does not have any functional parameter though. This is the reason why IntelliJ gives such a warning:



The screenshot shows a code editor with the following code:

```
91 inline fun printThree() {
```

A yellow tooltip window is open over the word "printThree". The tooltip contains the text:

Expected performance impact of inlining 'public inline fun printThree(): Unit defined in org.kotinacademy in file InlineRepeatBenchmark.kt' is insignificant. Inlining works best for functions with parameters of functional types

To understand why, we first need to understand what the problem is with operating on functions as objects. These kinds of objects, created using function literals, need to be held somehow. In Kotlin/JS, it is simple since JavaScript treats functions as first-class citizens. Therefore in Kotlin/JS, it is either a function or a function reference. On Kotlin/JVM, some object needs to be created, either using a JVM anonymous class or a normal class. Therefore the following lambda expression:

```
1 val lambda: ()->Unit = {  
2     // code  
3 }
```

Will be compiled to a class. Either a JVM anonymous class:

```
1 // Java  
2 Function0<Unit> lambda = new Function0<Unit>() {  
3     public Unit invoke() {  
4         // code  
5     }  
6 };
```

Or it can be compiled into a normal class defined in a separate file:

```
1 // Java  
2 // Additional class in separate file  
3 public class Test$lambda implements Function0<Unit> {  
4     public Unit invoke() {  
5         // code  
6     }  
7 }  
8  
9 // Usage  
10 Function0 lambda = new Test$lambda()
```

There is no significant difference between those two options.

Notice that the function type translates to the `Function0` type. This is what a function type with no arguments in JVM is compiled to. And the same is true with other function types:

- `()->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 those interfaces are generated by the Kotlin compiler. You cannot use them explicitly in Kotlin though because they are generated on demand. We should use function types instead. Although knowing that function types are just interfaces opens your eyes to some new possibilities:

```
1 class OnClickListener: ()->Unit {  
2     override fun invoke() {  
3         // ...  
4     }  
5 }
```

As illustrated in *Item 45: Avoid unnecessary object creation*, wrapping the body of a function into an object will slow down the code. This is why among the two functions below, the first one will be faster:

```
1 inline fun repeat(times: Int, action: (Int) -> Unit) {  
2     for (index in 0 until times) {  
3         action(index)  
4     }  
5 }  
6  
7 fun repeatNoinline(times: Int, action: (Int) -> Unit) {  
8     for (index in 0 until times) {  
9         action(index)  
10    }  
11 }
```

The difference is visible, but rarely significant in real-life examples. Although if we design our test well, you can see this difference clearly:

```
1 @Benchmark
2 fun nothingInline(blackhole: Blackhole) {
3     repeat(100_000_000) {
4         blackhole.consume(it)
5     }
6 }
7
8 @Benchmark
9 fun nothingNoninline(blackhole: Blackhole) {
10    noinlineRepeat(100_000_000) {
11        blackhole.consume(it)
12    }
13 }
```

The first one takes on my computer on average 189 ms. The second one takes on average 447 ms. This difference comes from the fact that in the first function we only iterate over numbers and call an empty function. In the second function, we call a method that iterates over numbers and calls an object, and this object calls an empty function. All that difference comes from the fact that we use an extra object (*Item 45: 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 were bought. We can do it simply by:

```
1 users.filter { it.bought }.sumByDouble { it.price }
```

In my machine, it takes 38 ms to calculate on average. How much would it be if `filter` and `sumByDouble` functions were not inline? 42 ms on average on my machine. This doesn't look like a lot, but this is $\sim 10\%$ difference every time when you use 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 as well into some object and whenever it is used, it needs to be done using this object. For instance, in the following code:

```
1 var l = 1L
2 noinlineRepeat(100_000_000) {
3     l += it
4 }
```

A local variable cannot be really used directly in non-inlines lambda. This is why during compilation time, the value of a will be wrapped into a reference object:

```
1 val a = Ref.LongRef()
2 a.element = 1L
3 noinlineRepeat(100_000_000) {
4     a.element = a.element + it
5 }
```

This is a more significant difference because often, 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 two times. Therefore extra object use will happen $2 * 100\ 000\ 000$. To see this difference, let's compare the following functions:

```
1 @Benchmark
2 // On average 30 ms
3 fun nothingInline(blackhole: Blackhole) {
4     var l = 0L
5     repeat(100_000_000) {
6         l += it
7     }
8     blackhole.consume(l)
9 }
10
```

```
11 @Benchmark
12 // On average 274 ms
13 fun nothingNoninline(blackhole: Blackhole) {
14     var l = 0L
15     noinlineRepeat(100_000_000) {
16         l += it
17     }
18     blackhole.consume(l)
19 }
```

The first one on my machine takes 30 ms. The second one 274 ms. This comes from the accumulated effects of the facts that the function is compiled to be an object, and the local variable needs to be wrapped. This is a significant difference. 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 inlined.

Non-local return is allowed

Previously defined `repeatNoninline` looks much like a control structure. Just compare it with a if or for-loop:

```
1 if(value != null) {
2     print(value)
3 }
4
5 for (i in 1..10) {
6     print(i)
7 }
8
9 repeatNoninline(10) {
10     print(it)
11 }
```

Although one significant difference is that `return` is not allowed inside:

```
1 fun main() {  
2     repeatNoinline(10) {  
3         print(it)  
4         return // ERROR: Not allowed  
5     }  
6 }
```

This is the result of what function literals are compiled to. We cannot return from `main`, when our code is located in another class. There is no such limitation when a function literal is inlined. The code will be located in the `main` function anyway.

```
1 fun main() {  
2     repeat(10) {  
3         print(it)  
4         return // OK  
5     }  
6 }
```

Thanks to that, functions can look and behave more like control structures:

```
1 fun getSomeMoney(): Money? {  
2     repeat(100) {  
3         val money = searchForMoney()  
4         if(money != null) return money  
5     }  
6     return null  
7 }
```

Costs of inline modifier

Inline is a useful modifier but it should not be used everywhere. **Inline functions cannot be recursive.** Otherwise, they would replace their calls infinitely. Recurrent cycles are especially dangerous because, at the moment, it does not show an error in the IntelliJ:

```
1 inline fun a() { b() }
2 inline fun b() { c() }
3 inline fun c() { a() }
```

Inline functions cannot use elements with more restrictive visibility. We cannot use private or internal functions or properties in a public inline function. In fact, an inline function cannot use anything with more restrictive visibility:

```
1 internal inline fun read() {
2     val reader = Reader() // Error
3     // ...
4 }
5
6 private class Reader {
7     // ...
8 }
```

This is why they cannot be used to hide implementation, and so they are rarely used in classes.

Finally, they make our code grow. To see the scale, let's say that I really like printing 3. I first defined the following function:

```
1 inline fun printThree() {  
2     print(3)  
3 }
```

I liked to call it 3 times, so I added this function:

```
1 inline fun threePrintThree() {  
2     printThree()  
3     printThree()  
4     printThree()  
5 }
```

Still not satisfied. I've defined the following functions:

```
1 inline fun threeThreePrintThree() {  
2     threePrintThree()  
3     threePrintThree()  
4     threePrintThree()  
5 }  
6  
7 inline fun threeThreeThreePrintThree() {  
8     threeThreePrintThree()  
9     threeThreePrintThree()  
10    threeThreePrintThree()  
11 }
```

What are they all compiled to? First two are very readable:

```
1 inline fun printThree() {
2     print(3)
3 }
4
5 inline fun threePrintThree() {
6     print(3)
7     print(3)
8     print(3)
9 }
```

Next two were compiled to the following functions:

```
1 inline fun threeThreePrintThree() {
2     print(3)
3     print(3)
4     print(3)
5     print(3)
6     print(3)
7     print(3)
8     print(3)
9     print(3)
10    print(3)
11 }
12
13 inline fun threeThreeThreePrintThree() {
14     print(3)
15     print(3)
16     print(3)
17     print(3)
18     print(3)
19     print(3)
20     print(3)
21     print(3)
22     print(3)
23     print(3)
```

```
24     print(3)
25     print(3)
26     print(3)
27     print(3)
28     print(3)
29     print(3)
30     print(3)
31     print(3)
32     print(3)
33     print(3)
34     print(3)
35     print(3)
36     print(3)
37     print(3)
38     print(3)
39     print(3)
40     print(3)
41 }
```

This is an abstract example, but it shows a big problem with inline functions: code grows really quickly when we overuse them. I met with 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 when we want to inline a function, but for some reason, we cannot inline all function type arguments. In such cases we can use the following modifiers:

- `crossinline` - it 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` - it 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.

```
1 inline fun requestNewToken(  
2     hasToken: Boolean,  
3     crossinline onRefresh: ()->Unit,  
4     noinline onGenerate: ()->Unit  
5 ) {  
6     if (hasToken) {  
7         httpCall("get-token", onGenerate) // We must use  
8         // noinline to pass function as an argument to a  
9         // function that is not inlined  
10    } else {  
11        httpCall("refresh-token") {  
12            onRefresh() // We must use crossinline to  
13            // inline function in a context where  
14            // non-local return is not allowed  
15            onGenerate()  
16        }  
17    }  
18 }  
19  
20 fun httpCall(url: String, callback: ()->Unit) {  
21     /*...*/  
22 }
```

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:

```
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 where we use inline functions are:

- Very often 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 with cases when one inline function calls some other inline functions. Remember that code growth accumulates.

Item 47: Consider using inline classes

Not only functions can be inlined, but also objects holding a single value can be replaced with this value. Such possibility was introduced as experimental in Kotlin 1.3, and to make it possible, we need to place the `inline` modifier before a class with a single primary constructor property:

```
1 inline class Name(private val value: String) {  
2     // ...  
3 }
```

Such a class will be replaced with the value it holds whenever possible:

```
1 // Code  
2 val name: Name = Name("Marcin")  
3  
4 // During compilation replaced with code similar to:  
5 val name: String = "Marcin"
```

Methods from such a class will be evaluated as static methods:

```
1 inline class Name(private val value: String) {  
2     // ...  
3  
4     fun greet() {  
5         print("Hello, I am $value")  
6     }  
7 }  
8  
9 // Code  
10 val name: Name = Name("Marcin")
```

```
11 name.greet()  
12  
13 // During compilation replaced with code similar to:  
14 val name: String = "Marcin"  
15 Name.`greet-impl`(name)
```

We can use inline classes to make a wrapper around some type (like `String` in the above example) with no performance overhead (*Item 45: Avoid unnecessary object creation*). Two especially popular uses of inline classes are:

- To indicate a unit of measure
- To use types to protect user from misuse

Let's discuss them separately.

Indicate unit of measure

Imagine that you need to use a method to set up timer:

```
1 interface Timer {  
2     fun callAfter(time: Int, callback: ()->Unit)  
3 }
```

What is this `time`? Might be time in milliseconds, seconds, minutes... it is not clear at this point and it is easy to make a mistake. A serious mistake. One famous example of such a mistake is Mars Climate Orbiter that crashed into Mars atmosphere. The reason behind that was that the software used to control it was developed by an external company and it produced outputs in different units of measure than the ones 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 unit of measure is by including it in the parameter name:

```
1 interface Timer {  
2     fun callAfter(timeMillis: Int, callback: ()->Unit)  
3 }
```

It is better but still leaves some space for mistakes. The property name is often not visible when a function is used. Another problem is that indicating the type this way is harder when the type is returned. In the example below, time is returned from `decideAboutTime` and its unit of measure is not indicated at all. It might return time in minutes and time setting would not work correctly then.

```
1 interface User {  
2     fun decideAboutTime(): Int  
3     fun wakeUp()  
4 }  
5  
6 interface Timer {  
7     fun callAfter(timeMillis: Int, callback: ()->Unit)  
8 }  
9  
10 fun setUpUserWakeUpUser(user: User, timer: Timer) {  
11     val time: Int = user.decideAboutTime()  
12     timer.callAfter(time) {  
13         user.wakeUp()  
14     }  
15 }
```

We might introduce the unit of measure of the returned value in the function name, for instance by naming it `decideAboutTimeMillis`,

but such a solution is rather rare as it makes a function longer every time we use it, and it states this low-level information even when we don't need to know about it.

A better way to solve this problem is to introduce stricter types that will protect us from misusing more generic types, and to make them efficient we can use inline classes:

```
1 inline class Minutes(val minutes: Int) {  
2     fun toMillis(): Millis = Millis(minutes * 60 * 1000)  
3     // ...  
4 }  
5  
6 inline class Millis(val milliseconds: Int) {  
7     // ...  
8 }  
9  
10 interface User {  
11     fun decideAboutTime(): Minutes  
12     fun wakeUp()  
13 }  
14  
15 interface Timer {  
16     fun callAfter(timeMillis: Millis, callback: ()->Unit)  
17 }  
18  
19 fun setUpUserWakeUpUser(user: User, timer: Timer) {  
20     val time: Minutes = user.decideAboutTime()  
21     timer.callAfter(time) { // ERROR: Type mismatch  
22         user.wakeUp()  
23     }  
24 }
```

This would force us to use the correct type:

```
1 fun setUpUserWakeUpUser(user: User, timer: Timer) {  
2     val time = user.decideAboutTime()  
3     timer.callAfter(time.toMillis()) {  
4         user.wakeUp()  
5     }  
6 }
```

It is especially useful for metric units, as in frontend, we often use a variety of units like pixels, millimeters, dp, etc. To support object creation, we can define DSL-like extension properties (and you can make them inline functions):

```
1 inline val Int.min  
2     get() = Minutes(this)  
3  
4 inline val Int.ms  
5     get() = Millis(this)  
6  
7 val timeMin: Minutes = 10.min
```

Protect us from type misuse

In SQL databases, we often identify elements by their IDs, which are all just numbers. Therefore, let's say that you have a student grade in a system. It will probably need to reference the id of a student, teacher, school etc.:

```
1 @Entity(tableName = "grades")
2 class Grades(
3     @ColumnInfo(name = "studentId")
4     val studentId: Int,
5     @ColumnInfo(name = "teacherId")
6     val teacherId: Int,
7     @ColumnInfo(name = "schoolId")
8     val schoolId: Int,
9     // ...
10 )
```

The problem is that it is really easy to later misuse all those ids, and the typing system does not protect us because they are all of type `Int`. The solution is to wrap all those integers into separate inline classes:

```
1 inline class StudentId(val studentId: Int)
2 inline class TeacherId(val teacherId: Int)
3 inline class SchoolId(val studentId: Int)
4
5 @Entity(tableName = "grades")
6 class Grades(
7     @ColumnInfo(name = "studentId")
8     val studentId: StudentId,
9     @ColumnInfo(name = "teacherId")
10    val teacherId: TeacherId,
11    @ColumnInfo(name = "schoolId")
12    val schoolId: SchoolId,
13    // ...
14 )
```

Now those id uses will be safe, and at the same time, the database will be generated correctly because during compilation, all those types will be replaced with `Int` anyway. This way, inline classes allow us to introduce types where they were not allowed before,

and thanks to that, we have safer code with no performance overhead.

Inline classes and interfaces

Inline classes just like other classes can implement interfaces. Those interfaces could let us properly pass time in any unit of measure we want.

```
1 interface TimeUnit {  
2     val millis: Long  
3 }  
4  
5 inline class Minutes(val minutes: Long): TimeUnit {  
6     override val millis: Long get() = minutes * 60 * 1000  
7     // ...  
8 }  
9  
10 inline class Millis(val milliseconds: Long): TimeUnit {  
11     override val millis: Long get() = milliseconds  
12 }  
13  
14 fun setUpTimer(time: TimeUnit) {  
15     val millis = time.millis  
16     // ...  
17 }  
18  
19 setUpTimer(Minutes(123))  
20 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 classes, since wrapped objects need to be created to let us present a type through this interface. **When we present inline classes through an interface, such classes are not inlined.**

Typealias

Kotlin typealias lets us create another name for a type:

```
1 typealias NewName = Int
2 val n: NewName = 10
```

Naming types is a useful capability used especially when we deal with long and repeatable types. For instance, it is popular practice to name repeatable function types:

```
1 typealias ClickListener =
2     (view: View, event: Event) -> Unit
3
4 class View {
5     fun addClickListener(listener: ClickListener) {}
6     fun removeClickListener(listener: ClickListener) {}
7     //...
8 }
```

What needs to be understood though is that **typealiases do not protect us in any way from type misuse**. They are just adding a new name for a type. If we would name `Int` as both `Millis` and `Seconds`, we would make an illusion that the typing system protects us while it does not:

```
1 typealias Seconds = Int
2 typealias Millis = Int
3
4 fun getTime(): Millis = 10
5 fun setUpTimer(time: Seconds) {}
6
7 fun main() {
8     val seconds: Seconds = 10
9     val millis: Millis = seconds // No compiler error
10
11     setUpTimer(getTime())
12 }
```

In the above example it would be easier to find what is wrong without 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 classes, we take the best from both options - it is both cheap and safe.

Summary

Inline classes let us wrap a type without performance overhead. Thanks to that, we improve safety by making our typing system protect us from value misuse. If you use a type with unclear meaning, especially a type that might have different units of measure, consider wrapping it with inline classes.

Item 48: Eliminate obsolete object references

Programmers that are used to languages with automatic memory management rarely think about freeing objects. In Java, for example, the Garbage Collector (GC) does all the job. Although forgetting about memory management altogether leads to memory leaks - unnecessary memory consumption - and in some cases to `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 beginner's mistake, where a developer willing to freely access an `Activity` (a concept similar to a window in a desktop application) from any place, stores it in a static or companion object property (classically, in a static field):

```
1 class MainActivity : Activity() {  
2  
3     override fun onCreate(savedInstanceState: Bundle?) {  
4         super.onCreate(savedInstanceState)  
5         //...  
6         activity = this  
7     }  
8  
9     //...  
10  
11    companion object {  
12        // DON'T DO THIS! It is a huge memory leak  
13        var activity: MainActivity? = null  
14    }  
15 }
```

Holding a reference to an activity in a companion object does

not let Garbage Collector release it as long as our application is running. Activities are heavy objects, and so it is a huge memory leak. There are some ways to improve it, 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 deal with a memory leak when we hold an object storing a reference to another one. Like in the example below, we hold a lambda function that captures a reference to the `MainActivity`:

```
1 class MainActivity : Activity() {  
2  
3     override fun onCreate(savedInstanceState: Bundle?) {  
4         super.onCreate(savedInstanceState)  
5         //...  
6  
7         // Be careful, we leak a reference to `this`  
8         logError = { Log.e(this::class.simpleName,  
9             it.message) }  
10    }  
11  
12    //...  
13  
14    companion object {  
15        // DON'T DO THIS! A memory leak  
16        var logError: ((Throwable)->Unit)? = null  
17    }  
18 }
```

Although problems with memory can be much more subtle. Take a look at the stack implementation below⁵⁸:

⁵⁸Example inspired by the book Effective Java by Joshua Bloch.

```
1  class Stack {
2      private var elements: Array<Any?> =
3          arrayOfNulls(DEFAULT_INITIAL_CAPACITY)
4      private var size = 0
5
6      fun push(e: Any) {
7          ensureCapacity()
8          elements[size++] = e
9      }
10
11     fun pop(): Any? {
12         if (size == 0) {
13             throw EmptyStackException()
14         }
15         return elements[--size]
16     }
17
18     private fun ensureCapacity() {
19         if (elements.size == size) {
20             elements = elements.copyOf(2 * size + 1)
21         }
22     }
23
24     companion object {
25         private const val DEFAULT_INITIAL_CAPACITY = 16
26     }
27 }
```

Can you spot a problem here? Take a minute to think about it.

The problem is that when we pop, we just decrement size, but we don't free an element on the array. Let's say that we had 1000 elements on the stack, and we popped nearly all of them one after another and our size is now equal to 1. We should have only one element. We can access only one element. Although our stack still holds 1000 elements and doesn't allow GC to destroy them.

All those objects are wasting our memory. This is why they are called memory leaks. If this 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:

```
1 fun pop(): Any? {
2     if (size == 0)
3         throw EmptyStackException()
4     val elem = elements[--size]
5     elements[size] = null
6     return elem
7 }
```

This is a rare example and a costly mistake, but there are everyday objects we use that profit, or can profit, from this rule as well. 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:

```
1 fun <T> mutableLazy(initializer: () -> T):
2     ReadWriteProperty<Any?, T> = MutableLazy(initializer)
3
4 private class MutableLazy<T>(
5     val initializer: () -> T
6 ) : ReadWriteProperty<Any?, T> {
7
8     private var value: T? = null
9     private var initialized = false
10
11    override fun getValue(
12        thisRef: Any?,
13        property: KProperty<*>
14    ): T {
15        synchronized(this) {
```

```
16         if (!initialized) {
17             value = initializer()
18             initialized = true
19         }
20         return value as T
21     }
22 }
23
24 override fun setValue(
25     thisRef: Any?,
26     property: KProperty<*>,
27     value: T
28 ) {
29     synchronized(this) {
30         this.value = value
31         initialized = true
32     }
33 }
34 }
```

Usage:

```
1 var game: Game? by mutableLazy { readGameFromSave() }
2
3 fun setUpActions() {
4     startNewGameButton.setOnClickListener {
5         game = makeNewGame()
6         startGame()
7     }
8     resumeGameButton.setOnClickListener {
9         startGame()
10    }
11 }
```

The above implementation of `mutableLazy` is correct, but has one flaw: `initializer` is not cleaned after usage. It means that it is

held as long as the reference to an instance of `MutableLazy` exist even though it is not useful anymore. This is how `MutableLazy` implementation can be improved:

```
1  fun <T> mutableLazy(initializer: () -> T):  
2  ReadWriteProperty<Any?, T> = MutableLazy(initializer)  
3  
4  private class MutableLazy<T>(  
5      var initializer: (() -> T)?  
6  ) : ReadWriteProperty<Any?, T> {  
7  
8      private var value: T? = null  
9  
10     override fun getValue(  
11         thisRef: Any?,  
12         property: KProperty<*>  
13     ): T {  
14         synchronized(this) {  
15             val initializer = initializer  
16             if (initializer != null) {  
17                 value = initializer()  
18                 this.initializer = null  
19             }  
20             return value as T  
21         }  
22     }  
23  
24     override fun setValue(  
25         thisRef: Any?,  
26         property: KProperty<*>,  
27         value: T  
28     ) {  
29         synchronized(this) {  
30             this.value = value  
31             this.initializer = null  
32         }  
33     }  
34 }
```

```
33     }
34 }
```

When we set initializer to `null`, the previous value can be recycled by the GC.

How important is this optimization? Not so important to bother for rarely used objects. There is a saying that premature optimization is the root of all evil. Although, it is good to set `null` to unused objects when it doesn't cost you much to do it. Especially when it is a function type which can capture many variables, or when it is an unknown class like `Any` or a generic type. For example, `Stack` from the above example might be used by someone to hold heavy objects. This is a general purpose tool and we don't know how it will be used. For such tools, we should care more about optimizations. This is especially true when we are 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:

```
1 private class SynchronizedLazyImpl<out T>(
2     initializer: () -> T, lock: Any? = null
3 ) : Lazy<T>, Serializable {
4     private var initializer: ((() -> T)? = initializer
5     private var _value: Any? = UNINITIALIZED_VALUE
6     private val lock = lock ?: this
7
8     override val value: T
9         get() {
10         val _v1 = _value
11         if (_v1 !== UNINITIALIZED_VALUE) {
12             @Suppress("UNCHECKED_CAST")
13             return _v1 as T
14         }
15
16         return synchronized(lock) {
17             val _v2 = _value
```

```
18         if (_v2 !== UNINITIALIZED_VALUE) {
19             @Suppress("UNCHECKED_CAST") (_v2 as T)
20         } else {
21             val typedValue = initializer!!()
22             _value = typedValue
23             initializer = null
24             typedValue
25         }
26     }
27 }
28
29 override fun isInitialized(): Boolean =
30     _value !== UNINITIALIZED_VALUE
31
32 override fun toString(): String =
33     if (isInitialized()) value.toString()
34     else "Lazy value not initialized yet."
35
36 private fun writeReplace(): Any =
37     InitializedLazyImpl(value)
38 }
```

The general rule is that **when we hold state, we should have memory management in our minds**. Before changing implementation, we should always think of the best trade-off for our project, having in mind not only memory and performance, but also the readability and scalability of our solution. Generally, readable code will also be doing fairly well in terms of performance or memory. Unreadable code is more likely to hide memory leaks or wasted CPU power. Though sometimes those two values stand in opposition and then, in most cases, readability is more important. When we develop a library, performance and memory are often more important.

There are a few common sources of memory leaks we need to discuss. First of all, caches hold objects that might never be used.

This is the idea behind caches, but it won't help us when we suffer from out-of-memory-error. The solution is to use soft references. Objects can still be collected by the GC if memory is needed, but often they will exist and will be used.

Some objects can be referenced using weak reference. For instance a Dialog on a screen. As long as it is displayed, it won't be Garbage Collected anyway. Once it vanishes we do not need to reference it anyway. It is a perfect candidate for an object being referenced using a weak reference.

The big problem is that memory leaks are sometimes hard to predict and do not manifest themselves until some point when the application crashes. Which is especially true for Android apps, as there are stricter limits on memory usage than for other kinds of clients, like desktop. This is why we should search for them using special tools. The most basic tool is the heap profiler. We also have some libraries that help in the search for data leaks. For instance, a popular library for Android is LeakCanary which shows a notification whenever a memory leak is detected.

It is important to remember that a need to free objects manually is pretty rare. In most cases, those objects are freed anyway thanks to the scope, or when objects holding them are cleaned. The most important way to avoid cluttering our memory is having variables defined in a local scope (*Item 2: Minimize the scope of variables*) and not storing possibly heavy data in top-level properties or object declarations (including companion objects).

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`. When you need to write a portal with news, you will have a list of news. Each of them will probably have a list of authors and a list of tags. When you make an online shop, you start from a list of products. They will most likely have a list of categories and a list of different variants. When a user buys, they use some basket which is probably a collection of products and amounts. Then they need to choose from a list of delivery options and a list of payment methods. In some languages, `String` is just a list of characters. Collections are everywhere in programming! Just think about your application and you will quickly see lots of collections.

This fact can be reflected also in programming languages. Most modern languages have some collection literals:

```
1 // Python  
2 primes = [2, 3, 5, 7, 13]  
3 // Swift  
4 let primes = [2, 3, 5, 7, 13]
```

Good collection processing was a flag functionality of functional programming languages. Name of the Lisp programming lan-

guage⁵⁹ stands for “list processing”. Most modern languages have good support for collection processing. This statement includes Kotlin which has one of the most powerful sets of tools for collection processing. Just think of the following collection processing:

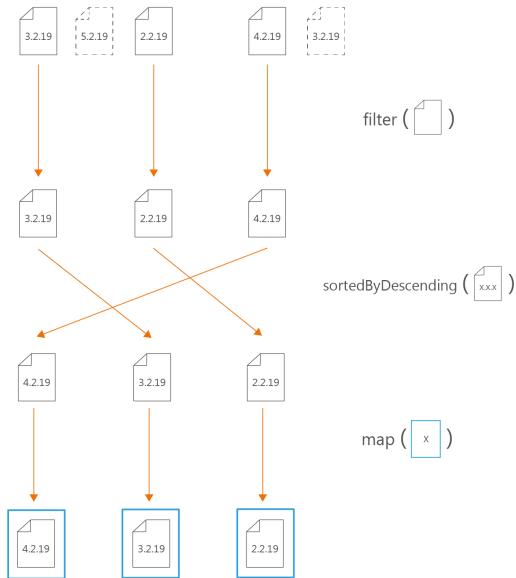
```
1 val visibleNews = mutableListOf<News>()
2 for (n in news) {
3     if(n.visible) {
4         visibleNews.add(n)
5     }
6 }
7
8 Collections.sort(visibleNews,
9     { n1, n2 -> n2.publishedAt - n1.publishedAt })
10 val newsItemAdapters = mutableListOf<NewsItemAdapter>()
11 for (n in visibleNews) {
12     newsItemAdapters.add(NewsItemAdapter(n))
13 }
```

In Kotlin it can be replaced with the following notation:

```
1 val newsItemAdapters = news
2     .filter { it.visible }
3     .sortedByDescending { it.publishedAt }
4     .map(::NewsItemAdapter)
```

Such notation is not only shorter, but it is also more readable. Every step makes a concrete transformation on the list of elements. Here is a visualization of the above processing:

⁵⁹Lisp is one of the oldest programming languages still in widespread use today. Often known as the father of all functional programming languages. Today, the best known general-purpose Lisp dialects are Clojure, Common Lisp, and Scheme.



The performance of the above examples is very similar. It is not always so simple though. Kotlin has a lot of collection processing methods and we can do the same processing in a lot of different ways. For instance, the below processing implementations have the same result but different performance:

```
1  fun productsListProcessing(): String {  
2      return clientsList  
3          .filter { it.adult }  
4          .flatMap { it.products }  
5          .filter { it.bought }  
6          .map { it.price }  
7          .filterNotNull()  
8          .map { "$$it" }  
9          .joinToString(separator = " + ")  
10     }  
11  
12     fun productsSequenceProcessing(): String {  
13         return clientsList.asSequence()  
14             .filter { it.adult }  
15             .flatMap { it.products.asSequence() }  
16             .filter { it.bought }  
17             .mapNotNull { it.price }  
18             .joinToString(separator = " + ") { "$$it" }  
19     }
```

Collection processing optimization is much more than just a brain-puzzle. Collection processing is extremely important and, in big systems, often performance-critical. This is why it is often key to improve the performance of our program. Especially if we do backend application development or data analysis. Although, when we are implementing front-end clients, we can face collection processing that limits the performance of our application as well. 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 ignored so easily.

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 49: Prefer Sequence for big collections with more than one processing step

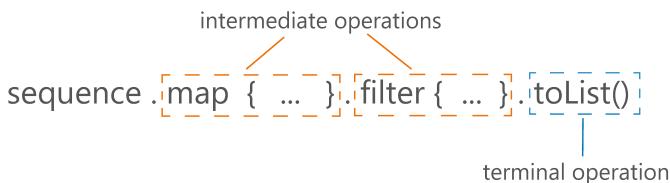
People often miss the difference between `Iterable` and `Sequence`. It is understandable since even their definitions are nearly identical:

```
1 interface Iterable<out T> {  
2     operator fun iterator(): Iterator<T>  
3 }  
4  
5 interface Sequence<out T> {  
6     operator fun iterator(): Iterator<T>  
7 }
```

You can say that the only formal difference between them is the name. Although `Iterable` and `Sequence` are associated with totally different usages (have different contracts), so nearly all their processing functions work in a different way. Sequences are lazy, so 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` on every step.

```
1  public inline fun <T> Iterable<T>.filter(  
2      predicate: (T) -> Boolean  
3  ): List<T> {  
4      return filterTo(ArrayList<T>(), predicate)  
5  }  
6  
7  public fun <T> Sequence<T>.filter(  
8      predicate: (T) -> Boolean  
9  ): Sequence<T> {  
10     return FilteringSequence(this, true, predicate)  
11 }
```

As a result, collection processing operations are invoked once they are used. Sequence processing functions are not invoked until the terminal operation (an operation that returns something else but Sequence). For instance, for Sequence, filter is an intermediate operation, so it doesn't do any calculations, but instead, it decorates the sequence with the new processing step. Calculations are done in a terminal operation like `toList`.



```
1 val seq = sequenceOf(1,2,3)
2 val filtered = seq.filter { print("f$it "); it % 2 == 1 }
3 println(filtered) // FilteringSequence@...
4
5 val asList = filtered.toList()
6 // f1 f2 f3
7 println(asList) // [1, 3]
8
9 val list = listOf(1,2,3)
10 val listFiltered = list
11     .filter { print("f$it "); it % 2 == 1 }
12 // f1 f2 f3
13 println(listFiltered) // [1, 3]
```

There are a few important advantages of 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 those advantages one-by-one.

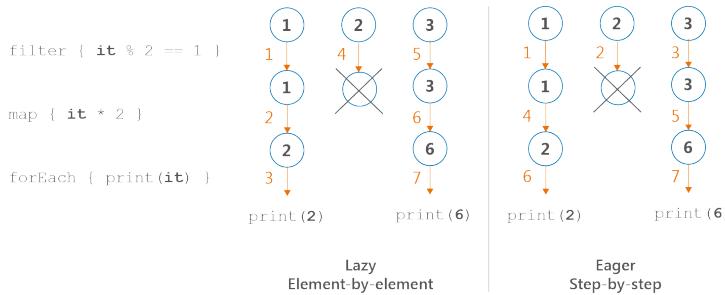
Order is important

Because of how iterable and sequence processing are 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 will call it an element-by-element or lazy order. In iterable processing, we take the first operation and we apply it to the whole collection, then move to the next operation, etc.. We will call it step-by-step or eager order.

```

1  sequenceOf(1,2,3)
2      .filter { print("F$it, "); it % 2 == 1 }
3      .map { print("M$it, "); it * 2 }
4      .foreach { print("E$it, ") }
5 // Prints: F1, M1, E2, F2, F3, M3, E6,
6
7 listOf(1,2,3)
8      .filter { print("F$it, "); it % 2 == 1 }
9      .map { print("M$it, "); it * 2 }
10     .foreach { print("E$it, ") }
11 // Prints: F1, F2, F3, M1, M3, E2, E6,

```



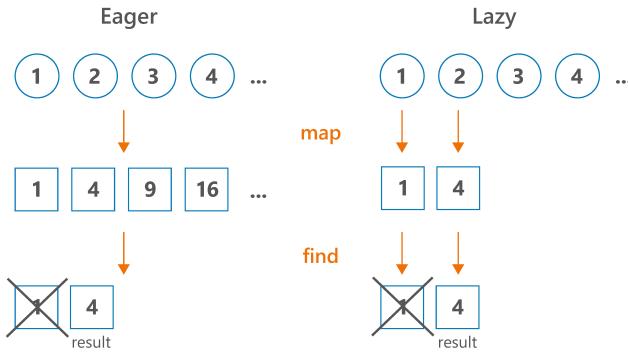
Notice that, if we were to implement those operations without any collection processing functions, but instead using classic loops and conditions, we would have element-by-element order like in sequence processing:

```
1  for (e in listOf(1,2,3)) {  
2      print("F$e, ")  
3      if(e % 2 == 1) {  
4          print("M$e, ")  
5          val mapped = e * 2  
6          print("E$mapped, ")  
7      }  
8  }  
9 // Prints: F1, M1, E2, F2, F3, M3, E6,
```

Therefore element-by-element order that is used in sequence processing is more natural. It also opens the door for low-level compiler optimizations - sequence processing can be optimized to basic loops and conditions. Maybe in the future, it will be.

Sequences do the minimal number of operations

Often we do not need to process the 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 the whole collection is processed as if it were to be returned on every operation. Sequences do not need that, and so they will do the minimal number of operations required to get the result.



Take a look at the example, where we have a few processing steps and we end our processing with `find`:

```

1  (1..10).asSequence()
2    .filter { print("F$it, "); it % 2 == 1 }
3    .map { print("M$it, "); it * 2 }
4    .find { it > 5 }
5  // Prints: F1, M1, F2, M3,
6
7  (1..10)
8    .filter { print("F$it, "); it % 2 == 1 }
9    .map { print("M$it, "); it * 2 }
10   .find { it > 5 }
11 // Prints: F1, F2, F3, F4, F5, F6, F7, F8, F9, F10,
12 // M1, M3, M5, M7, M9,
```

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 the performance of your processing. All that while looking nearly the same as the standard collection processing. Examples of such operations 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`. The first one needs the first element and a function specifying how to calculate the next one:

```
1 generateSequence(1) { it + 1 }
2     .map { it * 2 }
3     .take(10)
4     .foreach { print("$it, ") }
5 // Prints: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
```

The second mentioned sequence generator - `sequence` - uses a suspending function (coroutine⁶⁰) 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 the next Fibonacci numbers:

```
1 val fibonacci = sequence {
2     yield(1)
3     var current = 1
4     var prev = 1
5     while (true) {
6         yield(current)
7         val temp = prev
8         prev = current
9         current += temp
10    }
```

⁶⁰These 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 we need it.

```
11  }
12
13 print(fibonacci.take(10).toList())
14 // [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Notice that infinite sequences need to have a limited number of elements at some point. We cannot iterate over infinity.

```
1 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`, `any`, `all`, `none` or `indexOf`. Basically, those are the same operations for which sequences are more efficient because they do not need to process all elements. Although notice that for most of those operations, it is easy to fall into an infinite loop. `any` can only return `true` or run forever. Similarly `all` and `none` can only return `false` on an infinite collection. As a result, we generally either limit the number of elements by `take`, or we ask for just the first element using `first`.

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 - after every point, we have something ready to be used or stored. But it comes at a cost. Such collections need to be created and filled with data on every step.

```
1 numbers
2     .filter { it % 10 == 0 } // 1 collection here
3     .map { it * 2 } // 1 collection here
4     .sum()
5 // In total, 2 collections created under the hood
6
7 numbers
8     .asSequence()
9     .filter { it % 10 == 0 }
10    .map { it * 2 }
11    .sum()
12 // No collections created
```

This is a problem especially when we are dealing with big or heavy collections. Let's start from an extreme and yet common case: file reading. Files can weigh gigabytes. Allocating all the data in a collection at every processing step would be a huge waste of memory. This is why by default we use sequences to process files.

As an example, let's analyze crimes in the Chicago city. This city, like many others, shared on the internet the whole database of crimes that took place there since 2001⁶¹. This dataset at the moment weights 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>`):

⁶¹You can find this database at www.data.cityofchicago.org

```
1 // BAD SOLUTION, DO NOT USE COLLECTIONS FOR
2 // POSSIBLY BIG FILES
3 File("ChicagoCrimes.csv").readLines()
4     .drop(1) // Drop descriptions of the columns
5     .mapNotNull { it.split(",") .getOrNull(6) }
6         // Find description
7     .filter { "CANNABIS" in it }
8     .count()
9     .let(::println)
```

The result on my computer is `OutOfMemoryError`.



Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space

No wonder why. We create a collection and then we have 3 intermediate processing steps, which add up to 4 collections. 3 out of them contain the majority of this data file, which takes 1.53 GB, so they all need to consume more than 4.59 GB. This is a huge waste of memory. The correct implementation should involve using a sequence, and we do that using the function `useLines` that always operates on a single line:

```
1 File("ChicagoCrimes.csv").useLines { lines ->
2     // The type of `lines` is Sequence<String>
3     lines
4         .drop(1) // Drop descriptions of the columns
5         .mapNotNull { it.split(",") .getOrNull(6) }
6             // Find description
7         .filter { "CANNABIS" in it }
8         .count()
9         .let { println(it) } // 318185
10 }
```

On my computer it took 8.3s. To compare the efficiency of both ways, I made another experiment and I reduced this dataset size by dropping columns I don't need. This way I achieved `CrimeData.csv` file with the same crimes but with a size of only 728 MB. Then I did the same processing. In the first implementation, using collection processing, it takes around 13s; while the second one, using sequences, around 4.5s. As you can see, using sequences for bigger files is not only for memory but also for performance.

Although a collection does not need to be heavy. The fact that in every step we are creating a new collection is also a cost in itself, and this cost manifests itself when we are dealing with collections with a larger number of elements. The difference is not huge - mainly because created collections after many steps are initialized with the expected size and so when we add elements we just place them in the next position. This difference is still substantial, and it is the main reason why we should **prefer to use Sequence for big collections with more than one processing step**.

By “big collections” I mean both many elements and really heavy collection. It might be a list of integers with tens of thousands of elements. It might also be a list with just a few strings, but so each long that they all take many megabytes of data. Those 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:

```
1 fun singleStepListProcessing(): List<Product> {
2     return productsList.filter { it.bought }
3 }
4
5 fun singleStepSequenceProcessing(): List<Product> {
6     return productsList.asSequence()
7         .filter { it.bought }
8         .toList()
9 }
```

You will notice that there is nearly no difference in performance (actually simple list processing is faster because its `filter` function is inline). Although when you compare functions with more than one processing step, like the functions below, which use the `filter` and then `map`, the difference will be visible for bigger collections. To see a difference, let's compare typical processing with two and three processing steps for 5000 products:

```
1 fun twoStepListProcessing(): List<Double> {
2     return productsList
3         .filter { it.bought }
4         .map { it.price }
5 }
6
7 fun twoStepSequenceProcessing(): List<Double> {
8     return productsList.asSequence()
9         .filter { it.bought }
10        .map { it.price }
11        .toList()
12 }
13
14 fun threeStepListProcessing(): Double {
15     return productsList
16         .filter { it.bought }
17         .map { it.price }
```

```
18         .average()
19     }
20
21     fun threeStepSequenceProcessing(): Double {
22         return productsList.asSequence()
23             .filter { it.bought }
24             .map { it.price }
25             .average()
26     }
```

Below you can see the average results on a MacBook Pro (Retina, 15-inch, Late 2013)⁶² for 5000 products in the `productsList`:

1	twoStepListProcessing	81 095 ns
2	twoStepSequenceProcessing	55 685 ns
3	twoStepListProcessingAndAccumulate	83 307 ns
4	twoStepSequenceProcessingAndAccumulate	6 928 ns

It is hard to predict what performance improvement we should expect. From my observations, in a typical collection processing with more than one step, for at least a couple of thousands of elements, we can expect around a 20-40% performance improvement.

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 either way. `sorted` is an example from Kotlin stdlib (currently it is the only example). `sorted` uses optimal implementation: It accumulates the Sequence into `List` and then uses `sort` from Java stdlib. The disadvantage is that this accumulation process takes some additional time if we compare it to the same processing on a `Collection` (although if `Iterable` is not a `Collection` or array, then

⁶²Processor 2.6 GHz Intel Core i7, Memory 16 GB 1600 MHz DDR3

the difference is not significant because it needs to be accumulated as well).

It is controversial if Sequence should have methods like sorted because sequences with a method requiring all elements to calculate the next one are only partially lazy (evaluated when we need to get the first element) and don't work on infinite sequences. It was added because it is a popular function and it is much easier to use it in this way. Although Kotlin developers should remember its flaws, and especially that it cannot be used on infinite sequences.

```
1 generateSequence(0) { it + 1 }.take(10).sorted().toList()
2 // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 generateSequence(0) { it + 1 }.sorted().take(10).toList()
4 // Infinite time. Does not return.
```

sorted is a rare example of processing step which is faster on Collection than on Sequence. Still, when we do a few processing steps and single sorted function (or other function that needs to work on the whole collection), we can expect a performance improvement using sequence processing.

```
1 // Benchmarking measurement result: 150 482 ns
2 fun productsSortAndProcessingList(): Double {
3     return productList
4         .sortedBy { it.price }
5         .filter { it.bought }
6         .map { it.price }
7         .average()
8 }
9
10 // Benchmarking measurement result: 96 811 ns
11 fun productsSortAndProcessingSequence(): Double {
12     return productList.asSequence()
13         .sortedBy { it.price }
14         .filter { it.bought }
```

```
15      .map { it.price }
16      .average()
17 }
```

What about Java stream?

Java 8 introduced streams to allow collection processing. They act and look similar to Kotlin sequences.

```
1 productsList.asSequence()
2     .filter { it.bought }
3     .map { it.price }
4     .average()
5
6 productsList.stream()
7     .filter { it.bought }
8     .mapToDouble { it.price }
9     .average()
10    .orElse(0.0)
```

Java 8 streams are lazy and collected in the last (terminal) processing step. 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 a result of the fact that Kotlin sequences were designed when Java streams was already used—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 when we have a machine with multiple cores that are often unused (which is common

nowadays). Although use this with caution as this feature has known pitfalls⁶³.

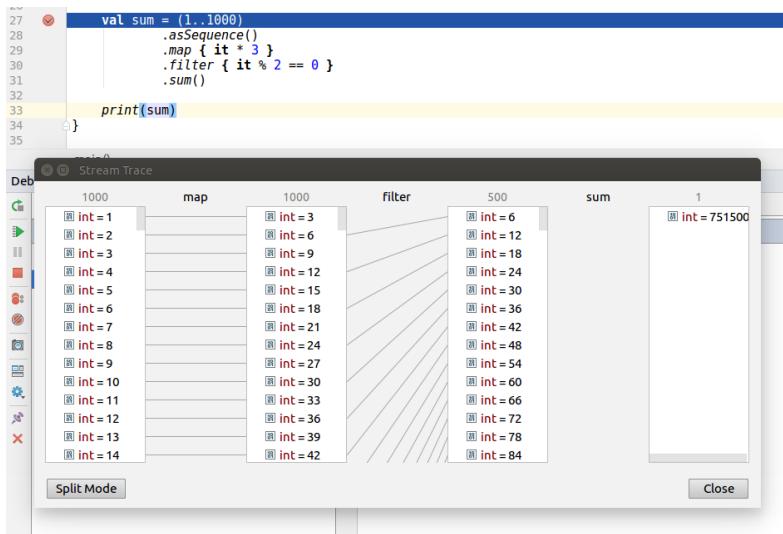
- Kotlin sequences can be used in common modules, Kotlin/JVM, Kotlin/JS, and Kotlin/Native modules. Java streams only in Kotlin/JVM, and only when the JVM version is at least 8.

In general, when we don't use parallel mode, it is hard to give a simple answer which of 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 the parallel mode. Otherwise, use Kotlin stdlib functions to have a homogeneous and clean code that can be used in different platforms or on common modules.

Kotlin Sequence debugging

Both Kotlin Sequence and Java Stream have the support that helps us debug elements flow at every step. For Java Stream, it requires a plugin named "Java Stream Debugger". Kotlin Sequences also required plugin named "Kotlin Sequence Debugger", though now this functionality is integrated into Kotlin plugin. Here is a screen showing sequence processing at every step:

⁶³The problems come from the common join-fork thread pool they use. Because of that, one processing can block another. There's also a problem with single element processing blocking other elements. Read more about it here: <https://dzone.com/articles/think-twice-using-java-8>



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 and so changing to collections requires a transformation to 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 to process heavy objects or for bigger collections with more than one processing step. Sequences are also supported by Kotlin Sequence Debugger that can help us by

visualizing how elements are processed. Sequences are not to replace classic collection processing. You should use them only when there's a good reason to, and you'll be rewarded with significant performance optimization.

Item 50: Limit the number of operations

Every collection processing method is a cost. For standard collection processing, it is most often another iteration over elements and additional collection created under the hood. For sequence processing, it is another object wrapping the whole sequence, and another object to keep operation⁶⁴. Both are costly especially when the number of processed elements is big. Therefore we limit the number of collection processing steps and we mainly do that by 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 do `mapNotNull`.

```
1 class Student(val name: String?)  
2  
3 // Works  
4 fun List<Student>.getNames(): List<String> = this  
5     .map { it.name }  
6     .filter { it != null }  
7     .map { it!! }  
8  
9 // Better  
10 fun List<Student>.getNames(): List<String> = this  
11     .map { it.name }  
12     .filterNotNull()  
13  
14 // Best  
15 fun List<Student>.getNames(): List<String> = this  
16     .mapNotNull { it.name }
```

⁶⁴We need to create lambda expression as an object because the operation is passed to the sequence object and so it cannot be inlined.

The biggest problem is not the misunderstanding of the importance of such changes, but rather the lack of knowledge about what collection processing functions we should use. This is another reason why it is good to learn them. Also, help comes from warnings which often suggest us to introduce a better alternative.

```

19
20
21
22
23
24
25
26
27
    fun makePassingStudentsListText(): String = studentsRepository
        .getStudents()
        .filter { it.pointsInSemester > 15 && it.result >= 50 }
        .sortedWith(compareBy { it.surname, it.name })
        .map { "${it.name} ${it.surname}, ${it.result}" }
        joinToString(separator = "\n")

```

Call chain on collection type may be simplified more... (⌘F1)

Still, it is good to know alternative composite operations. Here is a list of a few common function calls and alternative ways that limit the number of operations:

Instead of:	Use:
<code>.filter { it != null }</code> <code>.map { it!! }</code>	<code>.filterNotNull()</code>
<code>.map { <Transformation> }</code> <code>.filterNotNull()</code>	<code>.mapNotNull { <Transformation> }</code>
<code>.map { <Transformation> }</code> <code>.joinToString()</code>	<code>.joinToString { <Transformation> }</code>
<code>.filter { <Predicate 1> }</code> <code>.filter { <Predicate 2> }</code>	<code>.filter { <Predicate 1> && <Predicate 2> }</code>
<code>.filter { it is Type }</code> <code>.map { it as Type }</code>	<code>.filterIsInstance<Type>()</code>
<code>.sortedBy { <Key 2> }</code> <code>.sortedBy { <Key 1> }</code>	<code>.sortedWith(</code> <code>compareBy { <Key 1>, <Key 2> })</code>
<code>listOf(...)</code> <code>.filterNotNull()</code>	<code>listOfNotNull(...)</code>
<code>.withIndex()</code> <code>.filter { (index, elem) -></code> <code><Predicate using index> }</code> <code>.map { it.value }</code>	<code>.filterIndexed { index, elem -></code> <code><Predicate using index> }</code> (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 51: Consider Arrays with primitives for performance-critical processing

We cannot declare primitives in Kotlin, but as an optimization, they are used under the hood. This is a significant optimization, described already in *Item 45: Avoid unnecessary object creation*. Primitives are:

- Lighter, as every object adds additional weight,
- Faster, as accessing the value through accessors is an additional cost.

Therefore using primitives for a huge amount of data might be a significant optimization. One problem is that in Kotlin typical collections, like `List` or `Set`, are generic types. Primitives cannot be used as generic types, and 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 the 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<Int></code>	<code>List<Integer></code>
<code>Array<Int></code>	<code>Integer[]</code>
<code>IntArray</code>	<code>int[]</code>

How much lighter arrays with primitives are? Let's say that in Kotlin/JVM we need to hold 1 000 000 integers, and we can either choose to keep them in `IntArray` or in `List<Int>`. When you make measurements, you will find out that the `IntArray` allocates

4 000 016 bytes, while `List<Int>` allocates 20 000 040 bytes. It is 5 times more. If you optimize for memory use and you keep collections of types with primitive analogs, like `Int`, choose arrays with primitives.

```
1 import jdk.nashorn.internal.ir.debug.  
2 ObjectSizeCalculator.getObjectSize  
3  
4 fun main() {  
5     val ints = List(1_000_000) { it }  
6     val array: Array<Int> = ints.toTypedArray()  
7     val intArray: IntArray = ints.toIntArray()  
8     println(getObjectSize(ints))      // 20 000 040  
9     println(getObjectSize(array))    // 20 000 016  
10    println(getObjectSize(intArray)) // 4 000 016  
11 }
```

There is also a difference in performance. For the same collection of 1 000 000 numbers, when calculating an average, the processing over primitives is around 25% faster.

```
1 open class InlineFilterBenchmark {  
2  
3     lateinit var list: List<Int>  
4     lateinit var array: IntArray  
5  
6     @Setup  
7     fun init() {  
8         list = List(1_000_000) { it }  
9         array = IntArray(1_000_000) { it }  
10    }  
11  
12    @Benchmark  
13    // On average 1 260 593 ns  
14    fun averageOnIntList(): Double {
```

```
15         return list.average()
16     }
17
18     @Benchmark
19     // On average 868 509 ns
20     fun averageOnIntArray(): Double {
21         return array.average()
22     }
23 }
```

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. Although improvement in most cases is not significant enough to use arrays with primitives by default instead of lists. Lists are more intuitive and much more often in use, so in most cases we should use them instead. Just keep in mind this optimization in case you need to optimize some performance-critical parts.

Summary

In a typical case, `List` or `Set` should be preferred over `Array`. Though if you hold big collections of primitives, using `Array` might significantly improve your performance and memory use. This item is especially for library creators or developers writing games or advanced graphic processing.

Item 52: Consider using mutable collections

The biggest advantage of using mutable collections instead of immutable is that they are faster in terms of performance. 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 it is currently implemented in the Kotlin stdlib (Kotlin 1.2):

```
1 operator fun <T> Iterable<T>.plus(element: T): List<T> {
2     if (this is Collection) return this.plus(element)
3     val result = ArrayList<T>()
4     result.addAll(this)
5     result.add(element)
6     return result
7 }
```

Adding all elements from a previous collection is a costly process when we deal with bigger collections. This is why using mutable collections, especially if we 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. Although notice that those arguments rarely apply to local variables where synchronization or encapsulation is rarely needed. This is why for local processing, it generally makes more sense to use mutable collections. This fact can be reflected in the standard library where all collection processing functions are internally implemented using mutable collections:

```
1 inline fun <T, R> Iterable<T>.map(  
2     transform: (T) -> R  
3 ): List<R> {  
4     val size = if (this is Collection<*>) this.size else 10  
5     val destination = ArrayList<R>(size)  
6     for (item in this)  
7         destination.add(transform(item))  
8     return destination  
9 }
```

Instead of using immutable collections:

```
1 // This is not how map is implemented  
2 inline fun <T, R> Iterable<T>.map(  
3     transform: (T) -> R  
4 ): List<R> {  
5     var destination = listOf<R>()  
6     for (item in this)  
7         destination += transform(item)  
8     return destination  
9 }
```

Summary

Adding to mutable collections is generally faster, but immutable collections give us more control over how they are changed. Though in local scope we generally do not need that control, so mutable collections should be preferred. Especially in utils, where element insertion might happen many times.

Dictionary

Some technical terms are not well-understood and they require explanation. It 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, in opposition to other 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 top-level (top-level functions)
- In a class (member functions)
- In a function (local functions)

Here are some examples:

```
1 fun double(i: Int) = i * 2 // Top-level function
2
3 class A {
4
5     fun triple(i: Int) = i * 3 // Member function
6
7     fun twelveTimes(i: Int): Int { // Member function
8         fun fourTimes() = // Local function
9             double(double(i))
10        return triple(fourTimes())
11    }
12 }
```

Additionally, we can define anonymous functions using function literals:

```
1 val double = fun(i: Int) = i * 2 // Anonymous function
2 val triple = { i: Int -> i * 3 } // Lambda expression,
3 // 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. It is required to have an instance of this class to call it, and we need to use this class name to reference it. In the below example `doubled` is a member and a method. It is also a function as every method is a function.

```
1 class IntWrapper(val i: Int) {
2     fun doubled(): IntWrapper = IntWrapper(i * 2)
3 }
4
5 // Usage
6 val wrapper = IntWrapper(10)
7 val doubledWrapper = wrapper.doubled()
8
9 val doubledReference = IntWrapper::doubled
```

It is debatable if extension functions are methods as well, but in this book, we will say that they are, based on the premise that they are assigned to the extension type. An instance of this type is required to use it and to reference it. Such nomenclature is consistent with C# documentation. In the example below, you can see `tripled` which is an extension function, and also a method.

```
1 fun IntWrapper.tripled() = IntWrapper(i * 3)
2
3 // Use
4 val wrapper = IntWrapper(10)
5 val tripledWrapper = wrapper.tripled()
6
7 val tripledReference = IntWrapper::tripled
```

Notice that to call both references we need to pass an instance of `IntWrapper`:

```
1 val doubledWrapper2 = doubledReference(wrapper)
2 val tripledWrapper2 = tripledReference(wrapper)
```

Extension vs member

Member is an element defined in a class. In the following example, there are 4 members declared: the `name`, `surname`, and `fullName` properties, and the `withSurname` method.

```
1 class User(
2     val name: String,
3     val surname: String
4 ) {
5
6     val fullName: String
7         get() = "$name $surname"
8
9     fun withSurname(surname: String) =
10         User(this.name, surname)
11 }
```

Extensions are like fake added members to an existing class: they are elements defined outside of a class, but they are called same

as members. In the below example there are 2 extensions: the `officialFullName` extension property, and the `withName` extension function.

```
1 val User.officialFullName: String
2     get() = "$surname, $name"
3
4 fun User.withName(name: String) =
5     User(name, this.surname)
```

Parameter vs argument

The **parameter** is a variable defined in a function declaration. The **argument** is the actual value of this variable that gets passed to the function. Take a look at the next example. There, `length` in the `randomString` declaration is a parameter, and `10` in this function call is an argument.

```
1 fun randomString(length: Int): String {
2     // ....
3 }
4
5 randomString(10)
```

Similarly with 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.

```
1 inline fun <reified T> printName() {
2     print(T::class.simpleName)
3 }
4
5
6 fun main() {
7     printName<String>() // String
8 }
```

Primary vs Secondary constructor

A constructor is a special type of function⁶⁵ called to create an object. In Kotlin we treat constructors like a function producing an object. It can be declared in a class:

```
1 class SomeObject {
2     val text: String
3
4     constructor(text: String) {
5         this.text = text
6         print("Creating object")
7     }
8 }
```

Although as it is common that a constructor would be used to set up an object, there is a concept called **primary constructor** - a constructor defined just after the class name, with parameters that can be used to initialize properties:

⁶⁵Formally a subroutine, though in Kotlin constructors are treated just like functions, and constructor reference implements function type.

```
1 class SomeObject(text: String) {  
2     val text: String = text  
3  
4     init {  
5         print("Creating object")  
6     }  
7 }
```

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**:

```
1 class SomeObject(val text: String) {  
2  
3     init {  
4         print("Creating object")  
5     }  
6 }
```

When we need to create another constructor, we need to use a so-called **secondary constructor** - constructor that calls the primary constructor using `this` keyword:

```
1 class SomeObject(val text: String) {  
2  
3     constructor(date: Date): this(date.toString())  
4  
5     init {  
6         print("Creating object")  
7     }  
8 }
```

Although such a 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 34:

Consider primary constructor with named optional arguments) or a factory method when we need a different kind of object creation (described in detail in Item 33: Consider factory functions instead of constructors). If we need to support multiple subsets of the primary constructor arguments for Java use, we can use the primary constructor with the `JvmOverloads` annotation that is an instruction for the compiler to generate overloads for this function that substitute default parameter values.

```
1 class SomeObject @JvmOverloads constructor(
2     val text: String = ""
3 ) {
4     init {
5         print("Creating object")
6     }
7 }
```