Marcin Moskała

Effective Kotlin

BEST PRACTICES

Second Edition

Effective Kotlin

Best practices

Marcin Moskała

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

This version was published on 2024-12-03

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.

Tweet This Book!

Please help Marcin Moskała by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought @EffectiveKotlin!

The suggested hashtag for this book is #effectivekotlin.

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

#effectivekotlin

For my parents

Contents

Introduction: Be pragmatic

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The philosophy of Kotlin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The purpose of this book

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Who is this book for?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Book design

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Chapter organization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

How should this book be read?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Labels

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Code Sources

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Suggestions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Acknowledgments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Part 1: Good code

Chapter 1: Safety

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 1: Limit mutability

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

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

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

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

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

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

class InsufficientFunds : Exception()

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

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

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

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

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

Limiting mutability in Kotlin

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

- Read-only properties `val`,

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

Let's discuss these one by one.

Read-only properties

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

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

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

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

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

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

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

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

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

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

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

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

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

```
interface Element {
    val active: Boolean
}

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

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

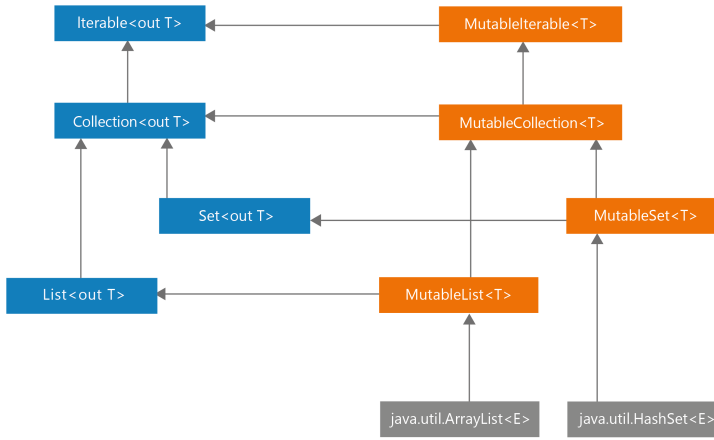
Remember that `val` doesn't mean immutable. It can be defined by a getter or a delegate. This fact gives us more freedom to change a final property into a property represented by a getter. However, when we don't need to use anything more complicated, we should define final properties, which are easier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

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

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

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

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

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

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

Separation between mutable and read-only collections

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

Copy in data classes

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

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

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

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

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

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

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

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

```
class User(
    val name: String,
    val surname: String
) {
    fun withSurname(surname: String) = User(name, surname)
}

var user = User("Maja", "Markiewicz")
user = user.withSurname("Moskała")
print(user) // User(name=Maja, surname=Moskała)
```

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

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

var user = User("Maja", "Markiewicz")
user = user.copy(surname = "Moskała")
print(user) // User(name=Maja, surname=Moskała)
```

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

Different kinds of mutation points

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

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

Both properties can be modified, but in different ways:

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

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

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

Both these ways are correct, and both have their pros and cons. They both have a single mutating point, but each is located in a different place. In the first one, the mutation takes place on the concrete list implementation. We might depend on the fact that the collection has proper synchronization in the case of multithreading, if we used a collection with support for concurrency[1]. In the second one, we need to implement the synchronization ourselves, but the overall safety is better because the mutating point is only a single property. However, in the case of a lack of synchronization, remember that we might still lose some elements:

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

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

[1]We will discuss such collections in the next item.

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

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

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

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

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

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

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

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

Summary

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

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

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

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

Item 2: Eliminate critical sections

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

The problem with threads and shared state

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

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

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

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

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

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

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

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

As I explained in the previous chapter, we don't encounter all these problems if we don't use mutability. However, in real-life applications we often cannot avoid mutability, so we need to learn how to deal with shared state[2]. **Whenever you have a shared state that might be modified by multiple threads, you need to ensure that all the operations on this state are executed correctly.** Each platform offers different tools for this, so let's learn about the most important tools for Kotlin/JVM[3].

[2] By *shared state*, I mean a state used by multiple threads.

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

Synchronization in Kotlin/JVM

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

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

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

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

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

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

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

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

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

Atomic objects

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

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

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

Concurrent collections

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

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

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

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

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

Of course, there are also libraries that offer other tools that support code synchronization. There are even Kotlin multiplatform libraries, such as AtomicFU, which provides multiplatform atomic objects[4].

[4] At the time of writing these words, AtomicFU is still in beta version, but it is already well-developed and seems rather stable.

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

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

Do not leak mutation points

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

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

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

    fun loadAll(): MutableList<User> = users

    //...
}
```

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

```
val userRepository = UserRepository()

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

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

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

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

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

    fun loadAll(): List<User> = users

    //...
}
```

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

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

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

    fun loadAll(): List<User> = users

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

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

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

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

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

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

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

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

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

```
class UserRepository {
    private var users: List<User> = listOf()

    fun loadAll(): List<User> = users

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

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

```
class UserRepository {
    private var users: List<User> = listOf()
    private val lock = Any()

    fun loadAll(): List<User> = users

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

Summary

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

Item 3: Eliminate platform types as soon as possible

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 4: Minimize the scope of variables

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Capturing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 5: Specify your expectations for arguments and state

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Arguments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

State

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Nullability and smart casting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Using Elvis operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

`error` function

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 6: Prefer standard errors to custom ones

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Using `Result` result type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Using `null` result type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Null is our friend, not an enemy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Defensive and offensive programming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 8: Close resources with `use`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 9: Write unit tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Chapter 2: Readability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 10: Design for readability

It is a known observation in programming that developers read code much more than they write it. A common estimate is that for every minute spent writing code, ten minutes are spent reading it[5]. If this seems unbelievable, just think about how much time you spend reading code when you are trying to find an error. I believe that everyone has been in this situation at least once in their career when they've been searching for an error for days or weeks, just to fix it by changing a single line. When we learn how to use a new API, it's often from reading code. We usually read code to understand the logic or how the implementation works. **Programming is mostly about reading, not writing.** Knowing this, it should be clear that we should code with readability in mind.

Reducing cognitive load

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

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

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

Which one is better, A or B? Using the naive reasoning that the one with fewer lines is better is not a good answer. We could remove the line breaks from the first implementation, but this wouldn't make it more readable. Counting the number of characters would not be very useful neither. Especially since the difference is not big. The first implementation has 79 characters, and the second has 68. The second implementation is only a bit shorter, but it is much less readable.

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

[5]This ratio was popularized by Robert C. Martin in the book *Clean Code*.

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

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

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

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

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

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

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

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

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

Do not get extreme

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

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

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

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

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

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

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

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

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

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

Conventions

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

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

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

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

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

infix fun String.and(s: String) = this + s
```

This code violates many rules that we will describe later:

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

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

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Unclear cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

When is it fine to break this rule?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 12: Use operators to increase readability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 13: Consider making types explicit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Explicit API mode for library authors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 14: Consider referencing receivers explicitly

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Many receivers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

DSL marker

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

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

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 17: Consider naming arguments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

When should we use named arguments?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Parameters with default arguments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Many parameters with the same type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Parameters with function types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

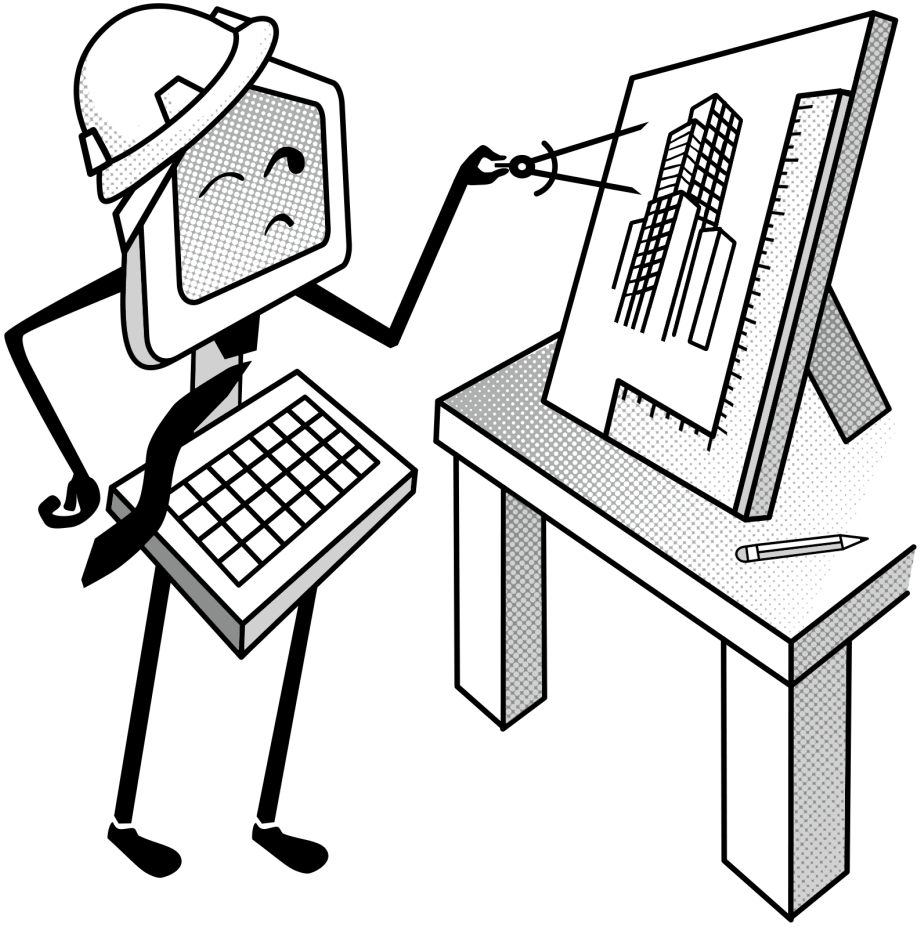
Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 18: Respect coding conventions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Part 2: Code design



Chapter 3: Reusability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 19: Do not repeat knowledge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Knowledge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Everything can change

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

When should we allow code repetition?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The single responsibility principle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 20: Do not repeat common algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Learn the standard library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Implementing your own utils

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 21: Use generics when implementing common algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Generic constraints

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 22: Avoid shadowing type parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 23: Consider using variance modifiers for generic types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 24: Reuse between different platforms by extracting common modules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Full-stack development

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Mobile development

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

All together

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Chapter 4: Abstraction design

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Abstraction in programming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Car metaphor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 25: Each function should be written in terms of a single level of abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Level of abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The Single Level of Abstraction principle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Abstraction levels in program architecture

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 26: Use abstraction to protect code against changes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Constant

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Next ID

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Abstractions give freedom

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Problems with abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

How to find a balance?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 27: Specify API stability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 28: Consider wrapping external APIs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 29: Minimize elements' visibility

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Using visibility modifiers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 30: Define contracts with documentation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Contracts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Defining a contract

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Do we need comments?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The KDoc format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The type system and expectations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Leaking implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 31: Respect abstraction contracts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Contracts are inherited

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Chapter 5: Object creation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 32: Consider factory functions instead of constructors

To create an object from a class, you need to use a constructor. In Kotlin, it is typically the primary constructor[6]:

```
class LinkedList<T>(
    val head: T,
    val tail: LinkedList<T>?
)

val list = LinkedList(1, LinkedList(2, null))
```

It is typical of Kotlin classes that their primary constructor defines properties that are essential part of this object state; as a result, primary constructor parameters are strongly bound to object structure. Using primary constructor for object creation is enough for simple classes, but more complex cases require different ways of constructing them. Think of the `LinkedList` from the above snippet. We might want to create it:

- Based on a set of items passed with the `vararg` parameter.
- From a collection of a different type, like `List` or `Set`.
- From another instance of the same type.

It is poor practice to define such functions as constructors. Don't do that. It is better to define them as functions (like `linkedListOf`, `toLinkedList` or `copy`), and here are a few reasons why:

- **Unlike constructors, functions have names.** Names explain how an object is created and what the arguments are. For example, let's say that you see the following code: `ArrayList(3)`. Can you guess what the argument means? Is it supposed to be the first element in the newly created list, or is it the initial capacity of the list? It is definitely not self-explanatory. In such a situation, a name like `ArrayList.withCapacity(3)` would clear up any confusion. Names are really useful: they explain arguments or characteristic ways of object creation. Another reason to have a name is that it solves potential conflicts between constructors with the same parameter types.
- **Unlike constructors, functions can return an object of any subtype of their return type.** This is especially important when we want to hide actual object implementations behind an interface. Think of `listOf` from

[6]See the section about primary/secondary constructors in the dictionary.

stdlib. Its declared return type is `List`, which is an interface. But what does this really return? The answer depends on the platform we use. It is different for Kotlin/JVM, Kotlin/JS, and Kotlin/Native because they each use different built-in collections. This is an important optimization that was implemented by the Kotlin team. It also gives Kotlin creators much more freedom. The actual type of a list might change over time, but as long as new objects still implement the `List` interface and act the same way, everything will be fine. Another example is `lazy` that declares `Lazy` interface as its result type, and depending on thead safety mode, in JVM it returns either `SynchronizedLazyImpl`, `SafePublicationLazyImpl` or `UnsafeLazyImpl`. Each of those classes is private, so their implementations are protected.

- **Unlike constructors, functions are not required to create a new object each time they're invoked.** This can be helpful because when we create objects using functions, we can include a caching mechanism to optimize object creation or to ensure object reuse for some cases (like in the Singleton pattern). We can also define a static factory function that returns `null` if the object cannot be created, like `Connections.createOrNull()`, which returns `null` when `Connection` cannot be created for some reason.
- **Factory functions can provide objects that might not yet exist.** This is intensively used by creators of libraries that are based on annotation processing. In this way, programmers can operate on objects that will be generated or used via a proxy without building the project.
- **When we define a factory function outside an object, we can control its visibility.** For instance, we can make a top-level factory function accessible only in the same file (`private` modifier) or in the same module (`internal` modifier).
- **Factory functions can be inlined, so their type parameters can be reified[7].** Libraries use this to provide a more convenient API.
- **A constructor needs to immediately call a constructor of a superclass or a primary constructor. When we use factory functions, we can postpone constructor usage.** That allows us to include a more complex algorithm in object creation.

Functions used to create an object are called **factory functions**. They are very important in Kotlin. When you search through Kotlin's official libraries, including the standard library, you will have trouble finding a non-private constructor, not to mention a secondary constructor. Most libraries expose only factory functions, most applications expose only primary constructors.

There are many kinds of factory functions we can use. We can create a list with `listOf`, `toList`, `List`, etc. Those are all factory functions. Let's learn about the most important kinds of factory functions and their conventions:

[7]Reified type parameters are explained in *Item 51: Use the inline modifier for functions with parameters of functional types*.

1. Companion object factory functions
2. Top-level factory functions
3. Builders
4. Conversion methods
5. Copying methods
6. Fake constructors
7. Methods in factory classes

Companion Object Factory Functions

In Java, every function has to be placed in a class. This is why most factory functions in Java are static functions that are placed either in the class they are producing or in some accumulator of static functions (like `Files`). Since the majority of the Kotlin community originated in Java, it has become popular to mimic this practice by defining factory functions in companion objects:

```
class LinkedList<T>(
    val head: T,
    val tail: LinkedList<T>?
) {

    companion object {
        fun <T> of(vararg elements: T): LinkedList<T> {
            /*...*/
        }
    }
}

// Usage
val list = LinkedList.of(1, 2)
```

The same can also be done with interfaces:

```
class LinkedList<T>(
    val head: T,
    val tail: LinkedList<T>?
) : MyList<T> {
    // ...
}

interface MyList<T> {
```

```
    // ...

    companion object {
        fun <T> of(vararg elements: T): MyList<T> {
            // ...
        }
    }
}

// Usage
val list = MyList.of(1, 2)
```

The advantage of this practice is that it is widely recognized among different programming languages. In some languages, like C++, it is called a *Named Constructor Idiom* as its usage is similar to a constructor, but with a name. It is also highly interoperable with other languages. From my personal experience, we used **companion object factory functions** most often when we were writing tests in Groovy. You just need to use `JvmStatic` annotation before the function, and you can easily use such a function in Groovy or Java in the same way as you use it in Kotlin.

The disadvantage of this practice is its complexity. Writing `List.of` is longer than `listOf` because it requires applying a suggestion two times instead of one. A companion object factory function needs to be defined in a companion object, while a top-level function can be defined anywhere.

It is worth mentioning that a companion object factory function can be defined as an extension to a companion object. It is possible to define an extension function to a companion object as long as such an object (even an empty one) exists.

```
interface Tool {
    companion object { /*...*/ }
}

fun Tool.Companion.createBigTool(/*...*/): Tool {
    //...
}

val tool = Tool.createBigTool()
```

There are some naming conventions for companion object factory functions. They are generally a Java legacy, but they still seem to be alive in our community:

- `from` - A type-conversion function that expects a single argument and returns a corresponding instance of the same type, for example:
  `val date: Date = Date.from(instant)`
- `of` - An aggregation function that takes multiple arguments and returns an instance of the same type that incorporates them, for example:
  `val faceCards: Set<Rank> = EnumSet.of(JACK, QUEEN, KING)`
- `valueOf` - A more verbose alternative to `from` and `of`, for example:
  `val prime: BigInteger = BigInteger.valueOf(Integer.MAX_VALUE)`
- `instance` or `getInstance` - Used in singletons to get the object instance. When parameterized, it will return an instance parameterized by arguments. Often, we can expect the returned instance to always be the same when the arguments are the same, for example:
  `val luke: StackWalker = StackWalker.getInstance(options)`
- `createInstance` or `newInstance` - Like `getInstance`, but this function guarantees that each call returns a new instance, for example:
  `val newArray = Array.newInstance(classObject, arrayLen)`
- `get{Type}` - Like `getInstance`, but used if the factory function is in a different class. Type is the type of the object returned by the factory function, for example:
  `val fs: FileStore = Files.getFileStore(path)`
- `new{Type}` - Like `newInstance`, but used if the factory function is in a different class. Type is the type of object returned by the factory function, for example:
  `val br: BufferedReader = Files.newBufferedReader(path)`

Those conventions map to other kind of factory functions. For example, `listOf` suggests that is creates a list from a set of elements, so it is an aggregation function. `createViewModel` suggests that it creates a new instance of a ViewModel, so it is a `new{Type}` function.

Companion objects are often treated as an alternative to static elements, but they are much more than that. Companion objects can implement interfaces and extend classes. This is a response to a popular request to allow inheritance for "static" elements. You can create abstract builders that are extended by concrete companion objects:

```
abstract class ActivityFactory {
    abstract fun getIntent(context: Context): Intent

    fun start(context: Context) {
        val intent = getIntent(context)
        context.startActivity(intent)
    }

    fun startForResult(
        activity: Activity,
        requestCode: Int
    ) {
        val intent = getIntent(activity)
        activity.startActivityForResult(
            intent,
            requestCode
        )
    }
}

class MainActivity : AppCompatActivity() {
    //...

    companion object : ActivityFactory() {
        override fun getIntent(context: Context): Intent =
            Intent(context, MainActivity::class.java)
    }
}

// Usage
val intent = MainActivity.getIntent(context)
MainActivity.start(context)
MainActivity.startForResult(activity, requestCode)
```

Notice that such abstract companion object factories can hold values, and so they can implement caching or support fake creation for testing. The advantages of companion objects are not as well used as they could be in the Kotlin programming community. Still, if you look at the implementations of the Kotlin team's libraries, you will see that companion objects are used extensively. For instance, in the Kotlin Coroutines library, nearly every companion object of a coroutine context implements a `CoroutineContext.Key` interface, which serves as a key we

use to identify this context[8].

Top-level factory functions

A popular way to create an object is by using top-level factory functions. Some common examples are `listOf`, `setOf`, `mapOf`, `lazy`, `sequence`, `flow`, etc.

```
fun <T> lazy(mode: LazyThreadSafetyMode, init:() -> T):Lazy<T\
>=
    when (mode) {
        SYNCHRONIZED -> SynchronizedLazyImpl(init)
        PUBLICATION -> SafePublicationLazyImpl(init)
        NONE -> UnsafeLazyImpl(init)
    }
```

Top-level factory functions are also used in projects to create objects in the way specific to the project. For instance, this is how one project could define Retrofit service creation:

```
fun createRetrofitService(baseUrl: String): Retrofit {
    return Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
}

fun <T> createService(clazz: Class<T>, baseUrl: String): T {
    val retrofit = createRetrofitService(baseUrl)
    return retrofit.create(clazz)
}
```

Object creation using top-level functions is a perfect choice for small and commonly created objects like `List` or `Map` because `listOf(1,2,3)` is simpler and more readable than `List.of(1,2,3)`. However, public top-level functions need to be used judiciously. Public top-level functions have a disadvantage: they are available everywhere, therefore it is easy to clutter up the developer's IDE tips. This problem becomes more serious when top-level functions have the same names as class methods and therefore get confused with them. This is why top-level functions should be named wisely.

[8]This mechanism is better explained in my *Kotlin Coroutines* book.

Builders

A very important kind of top-level factory function is builders. A good example is a list or a sequence builder:

```
val list = buildList {
    add(1)
    add(2)
    add(3)
}
println(list) // [1, 2, 3]

val s = sequence {
    yield("A")
    yield("B")
    yield("C")
}
println(s.toList()) // [A, B, C]
```

The typical way to implement a builder in Kotlin is using a top-level function and a DSL pattern[9]. In Kotlin Coroutines, builders are the standard way to start a coroutine or define a flow:

```
// Starting a coroutine
scope.launch {
    val processes = repo.getActiveProcesses()
    for (process in processes) {
        launch {
            process.start()
            repo.markProcessAsDone(process.id)
        }
    }
}

// Defining a flow
val flow = flow {
    var lastId: String = null
    do {
        val page = fetchPage(lastId)
```

[9]This will be explained soon in *Item 34: Consider defining a DSL for complex object creation*.

```
        emit(page.data)
        lastId = page.lastId
    } while (!page.isLast)
}
```

We will discuss DSLs in detail in *Item 34: Consider defining a DSL for complex object creation*. Of course, you can also meet builders defined using Java Builder Pattern, that look like this:

```
val user = UserBuilder()
    .withName("Marcin")
    .withSurname("Moskala")
    .withAge(30)
    .build()
```

In Kotlin we consider them less idiomatic than DSL builders. However, they are still used in some libraries.

Conversion methods

We often convert from one type to another. You might convert from `List` to `Sequence`, from `Int` to `Double`, from RxJava `Observable` to `Flow`, etc. For all these, the standard way is to use **conversion methods**. Conversion methods are methods used to convert from one type to another. They are typically named `to{Type}` or `as{Type}`. For example:

```
val sequence: Sequence = list.asSequence()

val double: Double = i.toDouble()

val flow: Flow = observable.asFlow()
```

The `to` prefix means that we are actually creating a new object of another type. For instance, if you call `toList` on a `Sequence`, you will get a new `List` object, which means that all elements of the new list are calculated and accumulated into a newly created list when this function is called. The `as` prefix means that the newly created object is a wrapper or an extracted part of the original object. For example, if you call `asSequence` on a `List`, the result object will be a wrapper around the original list. Using `as` conversion functions is more efficient but can lead to synchronization problems or unexpected behavior. For example, if you call `asSequence` on a `MutableList`, you will get a `Sequence` that references the original list.

```
fun main() {
    val seq1 = sequence<Int> {
        repeat(10) {
            print(it)
            yield(10)
        }
    }
    seq1.asSequence() // Nothing printed
    seq1.toList() // Prints 0123456789

    val l1 = mutableListOf(1, 2, 3, 4)
    val l2 = l1.toList()
    val seq2 = l1.asSequence()
    l1.add(5)
    println(l2) // Prints [1, 2, 3, 4]
    println(seq2.toList()) // Prints [1, 2, 3, 4, 5]
}
```

We often define our own conversion functions to convert between our own types. For example, when we need to convert between `UserJson` and `User` in an example application. Such methods are often defined as extension functions.

```
class User(
    val id: UserId,
    val name: String,
    val surname: String,
    val age: Int,
    val tokens: List<Token>
)

class UserJson(
    val id: UserId,
    val name: String,
    val surname: String,
    val age: Int,
    val tokens: List<Token>
)

fun User.toUserJson() = UserJson(
    id = this.id,
    name = this.name,
    surname = this.surname,
```

```
    age = this.age,
    tokens = this.tokens
)

fun UserJson.toUser() = User(
    id = this.id,
    name = this.name,
    surname = this.surname,
    age = this.age,
    tokens = this.tokens
)
```

Copying methods

When you need to make a copy of an object, define a copying method instead of defining a **copying constructor**. When you just want to make a direct copy, a good name is `copy`. When you need to apply a change to this object, a good name starts with `with` and the name of the property that should be changed (like `withSurname`).

```
val user2 = user.copy()
val user3 = user.withSurname(newSurname)
```

Data classes support the `copy` method, which can modify any primary constructor property, as we will see in *Item 37: Use the data modifier to represent a bundle of data*.

Fake constructors

Constructors in Kotlin are used the same way as top-level functions:

```
class A

fun b() = A()

val a1 = A()
val a2 = b()
```

They are also referenced in the same way as top-level functions (and constructor references implement a function type):

```
val reference: () -> A = ::A
```

From a usage point of view, capitalization is the only distinction between constructors and functions. By convention, classes begin with an uppercase letter, and functions begin with a lowercase letter. However, technically, functions can begin with an uppercase letter. This is used in different places, for example, in the Kotlin standard library. `List` and `MutableList` are interfaces. They cannot have constructors, but Kotlin developers wanted to allow the following `List` construction:

```
List(4) { "User$it" } // [User0, User1, User2, User3]
```

This is why the following functions are included in the Kotlin stdlib:

```
public inline fun <T> List(
    size: Int,
    init: (index: Int) -> T
): List<T> = MutableList(size, init)

public inline fun <T> MutableList(
    size: Int,
    init: (index: Int) -> T
): MutableList<T> {
    val list = ArrayList<T>(size)
    repeat(size) { index -> list.add(init(index)) }
    return list
}
```

These top-level functions look and act like constructors, but they have all the advantages of factory functions. Lots of developers are unaware of the fact that they are top-level functions under the hood. This is why they are often called *fake constructors*. They are a specific kind of top-level factory functions.

It is a very popular pattern in Kotlin libraries to expose only an interface, and produce its instance using a fake constructor. This way the actual implementation can be hidden. That has all advantages of factory functions, like:

- Hiding the actual implementation behind an interface (see `Job`, `CoroutineScope`, `Mutex` from kotlinx.coroutines).
- Depending on arguments, a different implementation can be returned, optimized for the given case (see `Channel` from kotlinx.coroutines).
- An algorithm can be used to create an object, which is not possible with a constructor (see `List` and `MutableList`).

Here are examples of fake constructors from the Kotlin Coroutines library:

```
fun Job(parent: Job? = null): CompletableJob = JobImpl(parent)

fun CoroutineScope(context: CoroutineContext): CoroutineScope=
   ContextScope(
       if (context[Job] != null) context else context + Job()
   )
```

Fake constructors should conceptually behave like regular constructors, otherwise you should prefer a different factory function kind.

There is one more way to declare a fake constructor. A similar result can be achieved using a companion object with the `invoke` operator. Take a look at the following example:

```
class Tree<T> {

    companion object {
        operator fun <T> invoke(
            size: Int,
            generator: (Int) -> T
        ): Tree<T> {
            //...
        }
    }
}

// Usage
Tree(10) { "$it" }
```

However, implementing *invoke* in a companion object to make a fake constructor is considered less idiomatic. I do not recommend it, primarily because it violates *Item 11: An operator's meaning should be consistent with its function name*. What does it mean to invoke a companion object? Remember that the name can be used instead of the operator:

```
Tree.invoke(10) { "$it" }
```

Invocation is a different operation from object construction. Using the `invoke` operator in this way is inconsistent with its name. More importantly, this approach is more complicated than just a top-level function. Just compare what reflection looks like when we reference a constructor, a fake constructor, and the `invoke` function in a companion object:

Constructor:

```
val f: ()->Tree = ::Tree
```

Fake constructor:

```
val f: ()->Tree = ::Tree
```

Invoke in a companion object:

```
val f: ()->Tree = Tree.Companion::invoke
```

I recommend using standard top-level functions when you need a fake constructor. However, these should be used sparingly to suggest typical constructor-like usage when we cannot define a constructor in the class itself, or when we need a capability that constructors do not offer (like a reified type parameter).

Methods on factory classes

There are many creational patterns associated with factory classes. For instance, an abstract factory or a prototype. Every creational pattern has some advantages.

Factory classes hold advantages over factory functions because classes can have a state. For instance, this is a very simple factory class that produces students with sequential id numbers:

```
data class Student(
    val id: Int,
    val name: String,
    val surname: String
)

class StudentsFactory {
    var nextId = 0
    fun next(name: String, surname: String) =
        Student(nextId++, name, surname)
}

val factory = StudentsFactory()
val s1 = factory.next("Marcin", "Moskala")
println(s1) // Student(id=0, name=Marcin, Surname=Moskala)
val s2 = factory.next("Igor", "Wojda")
println(s2) // Student(id=1, name=Igor, Surname=Wojda)
```

Factory classes can have properties that can be used to optimize object creation. When we can hold a state, we can introduce different kinds of optimizations or capabilities. We can, for instance, use caching or speed up object creation by duplicating previously created objects.

In practice, we most often extract factory classes when object creation requires multiple services or repositories. Extracting object creation logic helps us better organize our code.

```
class UserFactory(
    private val uuidProvider: UuidProvider,
    private val timeProvider: TimeProvider,
    private val tokenService: TokenService,
) {
    fun create(newUserData: NewUserData): User {
        val id = uuidProvider.next()
        return User(
            id = id,
            creationTime = timeProvider.now(),
            token = tokenService.generateToken(id),
            name = newUserData.name,
            surname = newUserData.surname,
            // ...
        )
    }
}
```

Summary

As you can see, Kotlin offers a variety of ways to specify factory functions, and they all have their own use. We should have them in mind when we design object creation. Each of them is reasonable for different cases. The most important thing is to be aware of the differences between them and to use them appropriately. The most popular factory function types are:

- Companion object factory functions
- Top-level factory functions (including fake constructors and builders)
- Conversion functions
- Methods on factory classes

Item 33: Consider a primary constructor with named optional arguments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Telescoping constructor pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Builder pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 34: Consider defining a DSL for complex object creation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Defining your own DSL

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

When should we use DSLs?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 35: Consider using dependency injection

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Chapter 6: Class design

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 36: Prefer composition over inheritance

Not Kotlin-specific

Inheritance is a powerful feature, but it is designed to create a hierarchy of objects with an "is a" relationship. When such a relationship is not clear, inheritance might be problematic and dangerous. When all we need is a simple code extraction or reuse, inheritance should be used with caution; instead, we should prefer a lighter alternative: class composition.

Simple behavior reuse

Let's start with a simple problem: we have two classes with partially similar behavior. They should both display a progress bar before some action and hide it afterwards.

```
class ProfileLoader {

    fun load() {
        // show progress bar
        // load profile
        // hide progress bar
    }
}

class ImageLoader {

    fun load() {
        // show progress bar
        // load image
        // hide progress bar
    }
}
```

In my experience, many developers would extract this common behavior by extracting a common superclass:

```
abstract class LoaderWithProgressBar {

    fun load() {
        // show progress bar
        action()
        // hide progress bar
    }

    abstract fun action()
}

class ProfileLoader : LoaderWithProgressBar() {

    override fun action() {
        // load profile
    }
}

class ImageLoader : LoaderWithProgressBar() {

    override fun action() {
        // load image
    }
}
```

This approach works for such a simple case, but it has important downsides we should be aware of:

- **We can only extend one class.** Extracting functionalities using inheritance often leads to either excessively complex hierarchies of types or to huge BaseXXX classes that accumulate many functionalities.
- **When we extend, we take everything from a class**, which leads to classes that have functionalities and methods they don't need (a violation of the Interface Segregation Principle).
- **Using superclass functionality is much less explicit.** In general, it is a bad sign when a developer reads a method and needs to jump into superclasses many times to understand how this method works.

These are strong reasons that should make us think about an alternative, and a very good one is composition. By composition, we mean holding an object as a property (we compose it) and using its functionalities. This is an example of how we can use composition instead of inheritance to solve our problem:

```
class ProgressBar {
    fun show() {
        /* show progress bar */
    }
    fun hide() {
        /* hide progress bar */
    }
}

class ProfileLoader {
    val progressBar = ProgressBar()

    fun load() {
        progressBar.show()
        // load profile
        progressBar.hide()
    }
}

class ImageLoader {
    val progressBar = ProgressBar()

    fun load() {
        progressBar.show()
        // load image
        progressBar.hide()
    }
}
```

Notice that composition is harder. We need to include the composed object and use it in every single class. This is the key reason why many prefer inheritance. However, this additional code is not useless: it informs the reader **that** a progress bar is used and **how** it is used. It also gives the developer more power over how this progress bar works.

Another thing to note is that composition is better when we want to extract multiple pieces of functionality. For instance, information that loading has finished:

```
class ImageLoader {
    private val progressBar = ProgressBar()
    private val finishedAlert = FinishedAlert()

    fun load() {
        progressBar.show()
        // load image
        progressBar.hide()
        finishedAlert.show()
    }
}
```

We cannot extend more than a single class. Therefore, if we wanted to use inheritance instead, we would be forced to place both functionalities in a single superclass. This often leads to a complex hierarchy of the types that are used to add these functionalities. Such hierarchies are very hard to read and often also to modify. Just think about what happens if we need an alert in two subclasses but not in a third one? One way to deal with this problem is to use a parameterized constructor:

```
abstract class InternetLoader(val showAlert: Boolean) {

    fun load() {
        // show progress bar
        innerLoad()
        // hide progress bar
        if (showAlert) {
            // show alert
        }
    }

    abstract fun innerLoad()
}

class ProfileLoader : InternetLoader(showAlert = true) {

    override fun innerLoad() {
        // load profile
    }
}

class ImageLoader : InternetLoader(showAlert = false) {
```

```
    override fun innerLoad() {
        // load image
    }
}
```

This is a bad solution. Having functionality blocked by a flag (`showAlert` in this case) is a bad sign. This problem is compounded when the subclass cannot block other unneeded functionality. This is a trait of inheritance: it takes everything from the superclass, not only what is needed.

Taking the whole package

When we use inheritance, we take everything from the superclass: methods, expectations (contract), and behavior. Therefore, it is a great tool for representing a hierarchy of objects, but it's not so great when we just want to reuse some common parts. For such cases, composition is better because we can choose the behavior we need. As an example, let's say that in our system we have decided to represent a `Dog` that can `bark` and `sniff`:

```
abstract class Dog {
    open fun bark() {
        /*...*/
    }
    open fun sniff() {
        /*...*/
    }
}
```

What if then we need to create a robot dog that can bark but can't sniff?

```
class Labrador : Dog()

class RobotDog : Dog() {
    override fun sniff() {
        error("Operation not supported")
        // Do you really want that?
    }
}
```

Notice that such a solution violates the *interface-segregation principle* as `RobotDog` has a method it doesn't need. It also violates the Liskov Substitution Principle as

it breaks superclass behavior. On the other hand, what if your `RobotDog` also needs to be a `Robot` class because `Robot` can calculate (i.e., it has the `calculate` method)? *Multiple inheritance* is not supported in Kotlin.

```
abstract class Robot {
    open fun calculate() {
        /*...*/
    }
}

class RobotDog : Dog(), Robot() // Error
```

These are serious design problems and limitations that do not occur when you use composition instead. When we use composition, we choose what we want to reuse. To represent type hierarchy, it is safer to use interfaces, and we can implement multiple interfaces. What has not yet been shown is that inheritance can lead to unexpected behavior.

Inheritance breaks encapsulation

To some degree, when we extend a class, we depend not only on how it works from the outside but also on how it is implemented inside. This is why we say that inheritance breaks encapsulation. Let's look at an example inspired by the book *Effective Java* by Joshua Bloch. Let's say that we need a set that will know how many elements have been added to it during its lifetime. Such a set can be created using inheritance from `HashSet`:

```
class CounterSet<T> : HashSet<T>() {
    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return super.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        elementsAdded += elements.size
        return super.addAll(elements)
    }
}
```

This implementation might look good, but it doesn't work correctly:

```
val counterList = CounterSet<String>()
counterList.addAll(listOf("A", "B", "C"))
print(counterList.elementsAdded) // 6
```

Why is that? The reason is that `HashSet` uses the `add` method under the hood of `addAll`. The counter is then incremented twice for each element added using `addAll`. This problem can be naively solved by removing the custom `addAll` function:

```
class CounterSet<T> : HashSet<T>() {
    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return super.add(element)
    }
}
```

However, this solution is dangerous. What if the creators of Java decided to optimize `HashSet.addAll` and implement it in a way that doesn't depend on the `add` method? If they did that, this implementation would break with a Java update. Together with this implementation, any other libraries which depend on our current implementation would break as well. The Java creators know this, so they are cautious of making changes to these types of implementations. The same problem affects any library creator or even developers of large projects. So, how can we solve this problem? We should use composition instead of inheritance:

```
class CounterSet<T> {
    private val innerSet = HashSet<T>()
    var elementsAdded: Int = 0
        private set

    fun add(element: T) {
        elementsAdded++
        innerSet.add(element)
    }

    fun addAll(elements: Collection<T>) {
        elementsAdded += elements.size
        innerSet.addAll(elements)
    }
```

```
}

val counterList = CounterSet<String>()
counterList.addAll(listOf("A", "B", "C"))
print(counterList.elementsAdded) // 3
```

One problem is that in this case we lose polymorphic behavior because `CounterSet` is not a `Set` anymore. To keep this behavior, we can use the delegation pattern. The delegation pattern means our class implements an interface, composes an object that implements the same interface, and forwards methods defined in the interface to this composed object. Such methods are called *forwarding methods*. Take a look at the following example:

```
class CounterSet<T> : MutableSet<T> {
    private val innerSet = HashSet<T>()
    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        elementsAdded += elements.size
        return innerSet.addAll(elements)
    }

    override val size: Int
        get() = innerSet.size

    override fun contains(element: T): Boolean =
        innerSet.contains(element)

    override fun containsAll(elements: Collection<T>):
    Boolean = innerSet.containsAll(elements)

    override fun isEmpty(): Boolean = innerSet.isEmpty()

    override fun iterator() =
        innerSet.iterator()
```

```
    override fun clear() =
        innerSet.clear()

    override fun remove(element: T): Boolean =
        innerSet.remove(element)

    override fun removeAll(elements: Collection<T>):
    Boolean = innerSet.removeAll(elements)

    override fun retainAll(elements: Collection<T>):
    Boolean = innerSet.retainAll(elements)
}
```

The problem now is that we need to implement a lot of forwarding methods (nine, in this case). Thankfully, Kotlin introduced interface delegation support that is designed to help in this kind of scenario. When we delegate an interface to an object, Kotlin will generate all the required forwarding methods during compilation. Here is Kotlin's interface delegation in action:

```
class CounterSet<T>(
    private val innerSet: MutableSet<T> = mutableSetOf()
) : MutableSet<T> by innerSet {

    var elementsAdded: Int = 0
        private set

    override fun add(element: T): Boolean {
        elementsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean {
        elementsAdded += elements.size
        return innerSet.addAll(elements)
    }
}
```

This is a case in which delegation is a good choice: we need polymorphic behavior and inheritance would be dangerous. However, delegation is not common. In most cases, polymorphic behavior is not needed or we use it in a different way, so composition without delegation is more suitable.

The fact that inheritance breaks encapsulation is a security concern, but in many cases the behavior is specified in a contract or we don't depend on it in subclasses (this is generally true when methods are designed for inheritance). There are other reasons to choose the composition pattern, one of which is that it is easier to reuse and gives us more flexibility.

Restricting overriding

To prevent developers from extending classes that are not designed for inheritance, we can just keep them final. However, if for some reason we need to allow inheritance, all methods are still final by default. To let developers override them, they must be set to open:

```
open class Parent {
    fun a() {}
    open fun b() {}
}

class Child : Parent() {
    override fun a() {} // Error
    override fun b() {}
}
```

Use this mechanism wisely and open only those methods that are designed for inheritance. Also, remember that when you override a method, you can make it final for all subclasses:

```
open class ProfileLoader : InternetLoader() {

    final override fun load() {
        // load profile
    }
}
```

In this way, you can limit the number of methods that can be overridden in subclasses.

Summary

There are a few important differences between composition and inheritance:

- **Composition is more secure** - We depend not on how a class is implemented but only on its externally observable behavior.
- **Composition is more flexible** - We can extend only a single class but we can compose many. When we inherit, we take everything; but when we compose, we can choose what we need. When we change the behavior of a superclass, we change the behavior of all subclasses. It is hard to change the behavior of only some subclasses. When a class we have composed changes, it will only change our behavior if it has changed its contract with the outside world.
- **Composition is more explicit** - This is both an advantage and a disadvantage. When we use a method from a superclass, we can do so implicitly, like methods from the same class. This requires less work, but it can be confusing and is more dangerous as it is easy to confuse where a method comes from (is it from the same class, a superclass, the top level, or is it an extension?). When we call a method on a composed object, we know where it comes from.
- **Composition is more demanding** - We need to use a composed object explicitly. When we add some functionalities to a superclass, we often do not need to modify the subclasses. When we use composition, we more often need to adjust usages.
- **Inheritance gives us strong polymorphic behavior** - This is also a double-edged sword. On one hand, it is convenient that a dog can be treated like an animal. On the other hand, it is very constraining: it must be an animal. Every subclass of an animal should be consistent with animal behavior. The superclass defines the contract, and the subclasses should respect it.

It is a general OOP rule to prefer composition over inheritance, but Kotlin encourages composition even more by making all classes and methods final by default and by making interface delegation a first-class citizen. This makes this rule even more important in Kotlin projects.

So, when is inheritance more reasonable? The rule of thumb is that **we should use inheritance when there is a definite "is a" relationship**. In other words, every class that uses inheritance needs to **be** its superclass. All unit tests written for superclasses should also pass for their subclasses, and every usage in production code should be exchangeable (Liskov substitution principle). Object-oriented frameworks for displaying views are good examples: `Application` in JavaFX, `Activity` in Android, `UIViewController` in iOS, and `React.Component` in React. The same is true when we define our own special kind of view element that always has the same set of functionalities and characteristics. Just remember to design these classes with inheritance in mind, and specify how inheritance should be used. Also, all the methods that are not designed for inheritance should be kept final.

Item 37: Use the data modifier to represent a bundle of data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The methods that `data` modifier overrides

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

When and how should we use destructuring?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Prefer data classes instead of tuples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 38: Use function types or functional interfaces to pass operations and actions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Using function types with type aliases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Reasons to use functional interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Avoid expressing actions using interfaces with multiple abstract methods

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 39: Use sealed classes and interfaces to express restricted hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Sealed classes and `when` expressions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 40: Prefer class hierarchies instead of tagged classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Tagged classes are not the same as classes using the state pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 41: Use enum to represent a list of values

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Enum or a sealed class?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 42: Respect the contract of `equals`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Equality

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Why do we need equals?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The contract of equals

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The problem with equals in java.net.URL

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Implementing `equals`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 43: Respect the contract of `hashCode`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Hash table

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The problem with mutability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

The contract of hashCode

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Implementing hashCode

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 44: Respect the contract of `compareTo`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Do we need a compareTo?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Implementing `compareTo`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 45: Consider extracting non-essential parts of your API into extensions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 46: Avoid member extensions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Why to avoid extension functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Avoid, not prohibit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Part 3: Efficiency

Chapter 7: Make it cheap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 47: Avoid unnecessary object creation

Object creation always costs something and can sometimes be expensive. This is why avoiding unnecessary object creation can be an important optimization. It can be done on many levels. For instance, in JVM it is guaranteed that a string object will be reused by other code running in the same virtual machine that happens to contain the same string literal[10]:

```
val str1 = "Lorem ipsum dolor sit amet"
val str2 = "Lorem ipsum dolor sit amet"
print(str1 == str2) // true
print(str1 === str2) // true
```

Boxed primitives (`Integer`, `Long`) are also reused in JVM when they are small (by default, the Integer Cache holds numbers in the range from -128 to 127).

```
val i1: Int? = 1
val i2: Int? = 1
print(i1 == i2) // true
print(i1 === i2) // true, because i2 was taken from cache
```

Reference equality (===) shows that this is the same object. However, if we use a number that is either smaller than -128 or bigger than 127, different objects will be created:

```
val j1: Int? = 1234
val j2: Int? = 1234
print(j1 == j2) // true
print(j1 === j2) // false
```

> Notice that a nullable type is used to force `Integer` instead of `int` under the hood. When we use `Int`, it is generally compiled to the primitive `int`, but if we make it nullable or when we use it as a type argument, `Integer` is used instead. This is because a primitive cannot be `null` and cannot be used as a type argument.

Knowing that such mechanisms are available in Kotlin, you might wonder how significant they are. Is object creation expensive?

[10] Java Language Specification, Java SE 8 edition, 3.10.5

Is object creation expensive?

Wrapping something into an object has 3 costs:

- **Objects take additional space.** In a modern 64-bit JDK, an object has a 12-byte header that is padded to a multiple of 8 bytes, so the minimum object size is 16 bytes. For 32-bit JVMs, the overhead is 8 bytes. Additionally, object references also take space. Typically, references are 4 bytes on 32-bit or 64-bit platforms up to -Xmx32G, and they are 8 bytes for memory allocation pool set above 32Gb (-Xmx32G). These are relatively small numbers, but they can add up to a significant cost. When we think about small elements like integers, they make a difference. `Int` as a primitive fit in 4 bytes, but when it is a wrapped type on the 64-bit JDK we mainly use today, it requires 16 bytes (it fits in the 4 bytes after the header), and its reference requires 4 or 8 bytes. In the end, it takes 5 or 6 times more space[11]. This is why an array of primitive integers (`IntArray`) takes 5 times less space than an array of wrapped integers (`Array<Int>`), as explained in the *Item 58: Consider Arrays with primitives for performance-critical processing*.
- **Access requires an additional function call when elements are encapsulated.** Again, this is a small cost as function use is very fast, but it can add up when we need to operate on a huge pool of objects. We will see how this cost can be eliminated in *Item 51: Use the inline modifier for functions with parameters of functional types* and *Item 49: Consider using inline classes*.
- **Objects need to be created and allocated in memory, references need to be created, etc.** These are small numbers, but they can rapidly accumulate when there are many objects. In the snippet below, you can see the cost of object creation.

```
class A

private val a = A()

// Benchmark result: 2.698 ns/op
fun accessA(blackhole: Blackhole) {
    blackhole.consume(a)
}

// Benchmark result: 3.814 ns/op
fun createA(blackhole: Blackhole) {
    blackhole.consume(A())
}
```

[11] To measure the size of concrete fields in JVM objects, use Java Object Layout.

```
// Benchmark result: 3828.540 ns/op
fun createListAccessA(blackhole: Blackhole) {
    blackhole.consume(List(1000) { a })
}

// Benchmark result: 5322.857 ns/op
fun createListCreateA(blackhole: Blackhole) {
    blackhole.consume(List(1000) { A() })
}
```

By eliminating objects, we can avoid all three of these costs. By reusing objects, we can eliminate the first and the third ones. If we know the costs of objects, we can start considering how we can minimize these costs in our applications by limiting the number of unnecessary objects. In the next few items, we will see different ways to eliminate or reduce the number of objects. In this item, I will only present one technique, that is designing classes to use primitives instead of wrapped types.

Using primitives

In JVM, we have a special built-in type to represent basic elements like numbers or characters. These are called primitives and are used by the Kotlin/JVM compiler under the hood wherever possible. However, there are some cases where a wrapped class (an object instance containing a primitive) needs to be used instead. The two main cases are:

1. When we operate on a nullable type (primitives cannot be `null`).
2. When we use a type as a generic type argument.

So, in short:

Kotlin type	Java type
Int	int
Int?	Integer
List<Int>	List<Integer>

Now you know that you can optimize your code to have primitives under the hood instead of wrapped types. Such optimization makes sense mainly on Kotlin/JVM and on some flavors of Kotlin/Native, but it doesn't make any sense on Kotlin/JS. Access to both primitive and wrapped types is relatively fast compared to other operations. The difference manifests itself when we deal with bigger collections

(we will discuss this in *Item 58: Consider Arrays with primitives for performance-critical processing*) or when we operate on an object intensively. Also, remember that forced changes might lead to less-readable code. **This is why I suggest this optimization only for performance-critical parts of code and in libraries.** You can identify the performance-critical parts of your code using a profiler.

To consider a concrete example, let's imagine that you implement a financial application in which you need to represent a stock snapshot. A snapshot is a set of values that are updated twice a second. It contains the following information:

```
class Snapshot(
    val afterHours: SessionDetails,
    val preMarket: SessionDetails,
    val regularHours: SessionDetails,
)

data class SessionDetails(
    val open: Double? = null,
    val high: Double? = null,
    val low: Double? = null,
    val close: Double? = null,
    val volume: Long? = null,
    val dollarVolume: Double? = null,
    val trades: Int? = null,
    val last: Double? = null,
    val time: Int? = null,
)
```

Since you are tracking tens of thousands of stocks, and the snapshot for each of them is updated twice a second, your application will create instances of `SessionDetails` many times per second, which will require a lot of effort from the garbage collector. To avoid this, you can change the `SessionDetails` class to use primitives instead of wrapped types by eliminating nullability.

```
data class SessionDetails(
    val open: Double = Double.NaN,
    val high: Double = Double.NaN,
    val low: Double = Double.NaN,
    val close: Double = Double.NaN,
    val volume: Long = -1L,
    val dollarVolume: Double = Double.NaN,
    val trades: Int = -1,
    val last: Double = Double.NaN,
```

```
    val time: Int = -1,
)
```

Note that **this change harms readability and makes this class harder to use** because `null` is a better way to represent the lack of a value than a special value like `NAN` or `-1`. However, in this case we decided to make this change because we are dealing with a performance-critical part of the application. By eliminating nullability, we've made our object allocate far fewer objects and much less memory. On a typical machine, the first version of `SessionDetails` allocates 192 bytes and needs to create 10 objects; in contrast, the second version allocates only 80 bytes and needs to create only one object. This is a significant difference that might be worth the trouble when we are dealing with tens of thousands of objects.

> If such interventions are not enough in your application, you can consider using a very powerful but also very dangerous pattern *object pool*. Its core idea is to make objects mutable and to store and reuse unused objects. This pattern is hard to implement correctly, and it is easy to introduce synchronization issues, which is why I don't recommend using it unless you're sure that you need it.

Summary

In this chapter, we learned about the costs of object creation and allocation. We also learned that we can reduce these costs by eliminating objects or reusing them, or by designing our objects to use primitives. The next items present other ways to reduce the number of unnecessary objects in our applications.

Item 48: Consider using object declarations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 49: Use caching when possible

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 50: Extract objects that can be reused

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Lazy initialization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 51: Use the inline modifier for functions with parameters of functional types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

A type argument can be reified

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Functions with functional parameters are faster when they are inlined

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Non-local return is allowed

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Costs of inline modifiers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Crossinline and noinline

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 52: Consider using inline value classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Indicate unit of measure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Protect us from value misuse

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Optimize for memory usage

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Inline value classes and interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Typealias

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 53: Eliminate obsolete object references

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Chapter 8: Efficient collection processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 54: Prefer Sequences for big collections with more than one processing step

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Order is important

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Sequences do the minimal number of operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Sequences can be infinite

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Sequences do not create collections at every processing step

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

When aren't sequences faster?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

What about Java streams?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Kotlin Sequence debugging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 55: Consider associating elements to a map

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Using Maps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Associating elements with keys

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 56: Consider using groupingBy instead of groupBy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

groupBy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

groupingBy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 57: Limit the number of operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 58: Consider Arrays with primitives for performance-critical processing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 59: Consider using mutable collections

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Item 60: Use appropriate collection types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Array-based list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Deque

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Linked lists

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Hash tables

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Sorted binary trees

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Dictionary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Function vs method

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Extension vs member

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Parameter vs argument

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.

Primary vs Secondary constructor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/effectivekotlin.