Marcin Moskala

Effective Kotlin

BEST PRACTICES

Kt. Academy

BETA

Effective Kotlin

Best practices

Marcin Moskala

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

This version was published on 2019-08-06

Contents

Item 30: Consider factory functions instead of constructors

The most common way for a class to allow a client to obtain an instance in Kotlin is to provide a primary constructor:

```kotlin
class MyLinkedList<T>(val head: T, val tail: MyLinkedList<T>?)
val list = MyLinkedList(1, MyLinkedList(2, null))
```

Though constructors are not the only way to create objects. There are many creational design patterns for object instantiation. Most of them revolve around the idea that instead of directly creating an object, a function can create the object for us. For instance, this top-level function creates an instance of `MyLinkedList`:

```kotlin
fun <T> myLinkedListOf(vararg elements: T): MyLinkedList<T>? {
   if(elements.isEmpty()) return null
   val head = elements.first()
   val elementsTail = elements.copyOfRange(1, elements.size)
   val tail = myLinkedListOf(*elementsTail)
   return MyLinkedList(head, tail)
}

val list = myLinkedListOf(1, 2)
```

Functions used as an alternative to constructors are called factory functions because they produce an object. Using factory functions instead of a constructor has many advantages, including:

- **Unlike constructors, functions have names**. Names explain how an object is created and what the arguments are. For example, let's say that you see the following code: `ArrayList(3)`. Can you guess what the argument means? Is it supposed to be the first element on this list, or is it the size of the list? It is definitely not self-explanatory. This is a situation where a name, like `ArrayList.withSize(3)`, would clear up any confusion. This is one case where a name is really useful: it explains arguments or characteristic ways of object creation. Another reason to have a name is that it solves a conflict between constructors with the same parameter types.

- **Unlike constructors, functions can return an object of any subtype of their return type**. This can be used to provide a better object for different cases. This is especially important when we want to hide actual object implementations behind an interface. Think of `listOf` from stdlib (standard library). It's declared return type is `List` which is an interface. What does it really return? The answer depends on the platform we use. It is different for Kotlin/JVM, Kotlin/JS, and Kotlin/Native because for every single one of them, built-in collections are used. This is an important optimization made by the Kotlin team. This gives them much more freedom because the actual type of list might change over time and as long as new objects still implement interface `List` and act the same way, everything will be fine.
- **Unlike constructors, functions are not required to create a new object each time they're invoked**. It can be helpful because when we create objects using functions, we can include a caching mechanism to optimize object creation or to ensure object reuse for some cases (like in the Singleton pattern). We can also define a static factory function that returns `null` if the object cannot be created. Like `Connections.createOrNull()` which returns `null` when `Connection` cannot be created for some reason.
- **Factory functions can provide objects that might not yet exist**. This fact is intensively used by creators of libraries that are based on annotation processing. Thanks to this, the project does not need to be built, for a programmer to operate on objects that are going to be generated or that are used via a proxy.
- **When we define a factory function outside of an object, we can control its visibility**. For instance, we can make a top-level factory function accessible only in the same file or in the same module.
- **Factory functions can be inline and so their type parameters can be reified**.
- **Factory functions can construct objects which might otherwise be complicated to construct**.
- **A constructor needs to immediately call a constructor of a superclass or a primary constructor**. When we use factory functions, we can postpone constructor usage:

```kotlin
fun makeListView(config: Config) : ListView {
   val items = … // Here we read items from config
   return ListView(items) // We call actual constructor
}
```

There is a limitation on factory functions usage: it cannot be used in subclass construction. This is because in subclass construction, we need to call the superclass constructor.

```kotlin
class IntLinkedList: MyLinkedList<Int>() {
// Supposing that MyLinkedList is open

   constructor(vararg ints: Int): myLinkedListOf(*ints) // Error
}
```

This is generally not a problem, since if we have decided that we want to create a superclass using a factory function, why would we use a constructor for its subclass? We should rather consider implementing a factory function for this class as well.

```kotlin
class MyLinkedIntList(head: Int, tail: MyLinkedIntList?):
   MyLinkedList<Int>(head, tail)

fun myLinkedIntListOf(vararg elements: Int): MyLinkedIntList? {
   if(elements.isEmpty()) return null
   val head = elements.first()
   val elementsTail = elements.copyOfRange(1, elements.size)
   val tail = myLinkedIntListOf(*elementsTail)
   return MyLinkedIntList(head, tail)
}
```

This function is longer than the previous constructor, but it has better characteristics - flexibility, independence of class, and the ability to declare a nullable return type.

There are strong reasons standing behind factory functions, though what needs to be understood is that **they are not a competition to the primary constructor**[1]. Factory

[1]See section about primary/secondary constructor in the dictionary

functions still needs to use a constructor in their body, so constructor must exist. It can be private if we really want to force creation using factory functions, but we rarely do (*Item 31: Consider primary constructor with named optional arguments*). **Factory functions are mainly a competition to secondary constructors**, and looking at Kotlin projects they generally win as secondary constructors are used rather rarely. **They are also a competition to themselves as there are variety of different kinds of factory functions**. Let's discuss different Kotlin factory functions:

1. Companion object factory function
2. Extension factory function
3. Top-level factory functions
4. Fake constructors
5. Methods on a factory class

Companion Object Factory Function

The most popular way to define a factory function is to define it in a companion object:

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

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

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

This approach should be very familiar to Java developers because it is a direct equivalent to a static factory method. Though developers of other languages might be familiar with it as well. In some languages, like C++, it is called a *Named Constructor Idiom* as its usage is similar to a constructor, but with a name.

In Kotlin, this approach works with interfaces too:

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

interface MyList<T> {
    // ...

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

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

Notice that the name of this function is not really descriptive, and yet it should be understandable for most developers. This is because there are some conventions that come from Java and thanks to them, a short word like of is enough to understand what the arguments mean. Here are some common names with their descriptions:

- `from`—A type-conversion function that takes a single parameter and returns a corresponding instance of this type, for example:

```
val date: Date = Date.from(instant)
```

- `of`—An aggregation function that takes multiple parameters and returns an instance of this type that incorporates them, for example:

```
val faceCards: Set<Rank> = EnumSet.of(JACK, QUEEN, KING)
```

- `valueOf`—A more verbose alternative to `from` and `of`, for example:

```
val prime: BigInteger = BigInteger.valueOf(Integer.MAX_VALUE)
```

- `instance` or `getInstance`—Used in singletons to get the only instance. When parameterized, will return an instance parameterized by arguments. Often we can expect that returned instance to always be the same when arguments are the same. For example:

```
val luke: StackWalker = StackWalker.getInstance(options)
```

- `createInstance` or `newInstance`—Like `getInstance`, but this function guarantees that each call returns a new instance. For example:

```
val newArray = Array.newInstance(classObject, arrayLen)
```

- `getType`—Like `getInstance`, but used if the factory function is in a different class. Type is the type of object returned by the factory function, for example:

```
val fs: FileStore = Files.getFileStore(path)
```

- `newType`—Like `newInstance`, but used if the factory function is in a different class. Type is the type of object returned by the factory function, for example:

```
val br: BufferedReader = Files.newBufferedReader(path)
```

Many less-experienced Kotlin developers treat companion object members like static members which need to be grouped in a single block. However, companion objects are actually much more powerful: for example, companion objects can implement interfaces and extend classes. So, we can implement general companion object factory functions like the one below:

```kotlin
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 products, you will see that companion objects are strongly used. For instance in the Kotlin Coroutines library, nearly every companion object of coroutine context implements an interface `CoroutineContext.Key` as they all serve as a key we use to identify this context.

Extension factory functions

Sometimes we want to create a factory function that acts like an existing companion object function, and we either cannot modify this companion object or we just want to specify a new function in a separate file. In such a case we can use another advantage of companion objects: we can define extension functions for them.

Suppose that we cannot change the `Tool` interface:

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

Nevertheless, we can define an extension function on its companion object (when this class or interface has any companion object declared, at least an empty one):

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

At the call site we can then write:

```
Tool.createBigTool()
```

This is a powerful possibility that lets us extend external libraries with our own factory methods. One catch is that to make an extension on companion object, there must be some (even empty) companion object:

```
interface Tool {
    companion object {}
}
```

Top-level functions

One popular way to create an object is by using top-level factory functions. Some common examples are `listOf`, `setOf`, and `mapOf`. Similarly, library designers specify top-level functions that are used to create objects. Top-level factory functions are used widely. For example, in Android we have the tradition of defining a function to create an `Intent` to start an Activity. In Kotlin, the `getIntent()` can be written as a companion object function:

```
class MainActivity: Activity {

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

In Kotlin Anko library, we can use the top-level function `intentFor` with reified type instead:

```
intentFor<MainActivity>()
```

This function can be also used to pass arguments:

```
intentFor<MainActivity>("page" to 2, "row" to 10)
```

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

Fake constructors

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

```
class A
val a = A()
```

They are also referenced the same as top-level functions (and constructor reference implements function interface):

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

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

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

This is why following functions are included (since Kotlin 1.1) in Collections.kt:

```kotlin
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*.

Fake constructors have all the advantages of factory functions over normal constructors. The ones that are the most useful in this case are:

- Being a constructor for an interface or an abstract function

- They can be inline, and they can have reified type arguments

Except for that, we prefer to use them as constructors. They look like constructors and they should behave this way. If you want to include caching, returning a nullable type or returning a subclass of a class that can be created, consider using a factory function with a name.

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:

```kotlin
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 very rarely used and I do not recommend it. First of all, because it breaks *Item 12: Use operator methods according to their names*. What does it mean to invoke a companion object? Remember that the name can be used instead of the operator:

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

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

Constructor:

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

Fake constructor:

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

Invoke in companion object:

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

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

Methods on a factory class

There are many creational patterns associated with factory classes. For instance, abstract factory or prototype. Every one of them has some advantages.

We will see that some of these approaches are not reasonable in Kotlin. In the next item, we will see that the telescoping constructor and builder pattern rarely makes sense in Kotlin.

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

```kotlin
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("Maja", "Markiewicz")
println(s2) // Student(id=1, name=Maja, Surname=Markiewicz)
```

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

Summary

As you can see, Kotlin offers variety of ways to specify factory functions and they all have their own use. We should have them in mind when we design object creation. Each of them is reasonable for different cases. Some of them should preferably be used with caution: Fake Constructors, Top-Level Factory Method and Extension Factory Function. The most universal way to define a factory function, is by using a Companion Object. It is safe and very intuitive for most developers since usage is very similar to Java Static Factory Methods, and Kotlin mainly inherits its style and practices from Java.

Item 33: Prefer composition over inheritance

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

Simple behavior reuse

Let’s start with a simple problem: we have two classes with partially similar behavior - progress bar display before and hide after logic.

```
class ProfileLoader {

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

class ImageLoader {

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

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

```kotlin
abstract class LoaderWithProgress {

    fun load() {
        // show progress
        innerLoad()
        // hide progress
    }

    abstract fun innerLoad()
}

class ProfileLoader: LoaderWithProgress() {

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

class ImageLoader: LoaderWithProgress() {

    override fun innerLoad() {
        // 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 huge BaseXXX classes that accumulate many functionalities or too deep and complex hierarchies of types.
- **When we extend, we take everything from a class**, which leads to classes that have functionality and methods they don't need (a violation of the Interface Segregation Principle).
- **Using superclass functionality is much less explicit**. In general, it is a bad sign when a developer reads a method and needs to jump into superclasses

many times to understand how the method works.

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

```kotlin
class Progress {
    fun showProgress() { /* show progress */ }
    fun hideProgress() { /* hide progress */ }
}

class ProfileLoader {
    val progress = Progress()

    fun load() {
        progress.showProgress()
        // load profile
        progress.hideProgress()
    }
}

class ImageLoader {
    val progress = Progress()

    fun load() {
        progress.showProgress()
        // load image
        progress.hideProgress()
    }
}
```

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

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

```kotlin
class ImageLoader {
    private val progress = Progress()
    private val finishedAlert = FinishedAlert()

    fun load() {
        progress.showProgress()
        // load image
        progress.hideProgress()
        finishedAlert.show()
    }
}
```

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

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

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

    abstract fun innerLoad()
}

```

```kotlin
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. It takes functionality a subclass doesn't need and blocks it. The problem is compounded when the subclass cannot block other unneeded functionality. When we use inheritance we take everything from the superclass, not only what we need.

Taking the whole package

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

```kotlin
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() {
       throw Error("Operation not supported")
       // Do you really want that?
   }
}
```

Notice that such solution violates *interface-segregation principle* - `RobotDog` has a method it doesn't need. It also violates Liskov Substitution Principle by breaking superclass behavior. On the other hand, what if your `RobotDog` needs to be a `Robot` class as well because `Robot` can calculate (have `calculate` method)? *Multiple inheritance* is not supported in Kotlin.

```
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 do we want to reuse. To represent type hierarchy it is safer to use interfaces, and we can implement multiple interfaces. What was not yet shown is that inheritance can lead to unexpected behavior.

Inheritance breaks encapsulation

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

```kotlin
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:

```kotlin
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`. The problem can be naively solved by removing custom `addAll` function:

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

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

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

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

```
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 interface delegation presented in action:

```
class CounterSet<T>(
        private val innerSet: MutableSet<T> = setOf()
) : 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 where delegation is a good choice: We need polymorphic behavior and inheritance would be dangerous. In most cases, polymorphic behavior is not needed or we use it in a different way. In such a case composition without delegation is more suitable. It is easier to understand and more flexible.

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

Restricting overriding

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

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

class Child: Parent() {
   override fun a() {} // Error: final a cannot be overridden
   override fun b() {}
}
```

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

```kotlin
open class ProfileLoader: InternetLoader() {

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

This way you can limit the number of methods that can be overridden in subclasses.

Summary

There are a few important differences between composition and inheritance:

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

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

When is composition more reasonable then? The rule of thumb: **we should use inheritance when there is a definite "is a" relationship**. Not only linguistically, but meaning that every class that inherits from a superclass needs to “be” its superclass. All unit tests for superclasses should always pass for their subclasses (Liskov substitution principle). Object-oriented frameworks for displaying views are good examples: `Application` in JavaFX, `Activity` in Android, `UIViewController` in iOS, and `React.Component` in React. The same is true when we define our own special kind of view element that always has the same set of functionalities and characteristics. Just remember to design these classes with inheritance in mind, and specify how inheritance should be used. Also, keep methods that are not designed for inheritance final.

Item 42: Avoid unnecessary object creation

Object creation can sometimes be expensive and always costs something. This is why avoiding unnecessary object creation can be an important optimization. It can be done on many levels. For instance, in JVM it is guaranteed that a string object will be reused by any other code running in the same virtual machine that happens to contain the same string literal[2]:

```kotlin
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 Integer Cache holds numbers in a range from -128 to 127).

```kotlin
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. If we use number that is either smaller than -128 or bigger than 127 though, different objects will be produced:

```kotlin
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 when we make it nullable or when we use it as a type argument, `Integer` is used instead. It is because primitive cannot be `null` and cannot be used as a type argument. Knowing that such mechanisms were introduced in the language, you might wonder how significant they are. Is object creation expensive?

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

Is object creation expensive?

Wrapping something into an object has 3 parts of cost:

- **Objects take additional space**. In a modern 64-bit JDK, an object has a 12-byte header, padded to a multiple of 8 bytes, so the minimum object size is 16 bytes. For 32-bit JVMs, the overhead is 8 bytes. Additionally, object references take space as well. Typically, references are 4 bytes on 32bit platforms or on 64bit platforms up to -Xmx32G, and 8 bytes above 32Gb (-Xmx32G). Those are relatively small numbers, but they can add up to a significant cost. When we think about such small elements like integers, they make a difference. Int as a primitive fits in 4 bytes, when as a wrapped type on 64-bit JDK we mainly use today, it requires 16 bytes (it fits in the 4 bytes after the header) + its reference required 8 bytes. In the end it takes 5 times more space[3].
- **Access requires an additional function call when elements are encapsulated**. That is again a small cost as function use is really fast, but it can add-up when we need to operate on a huge pool of objects. Do not limit encapsulation, avoid creating unnecessary objects especially in performance critical parts of your code.
- **Objects need to be created**. An object needs to be created, allocated in the memory, a reference needs to be created, etc. It is also a really small cost, but it can add up.

```
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) {
```

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

```kotlin
    blackhole.consume(A())
}

// 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 costs. By reusing objects, we can eliminate the first and the third one. Knowing that, you might start thinking about limiting the number of unnecessary objects in your code. Let's see some ways we can do that.

Object declaration

A very simple way to reuse an object instead of creating it every time is using object declaration (singleton). To see an example, let's imagine that you need to implement a linked list. The linked list can be either empty, or it can be a node containing an element and pointing to the rest. This is how it can be implemented:

```kotlin
sealed class LinkedList<T>

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

class Empty<T>: LinkedList<T>()

// Usage
```

```
val list: LinkedList<Int> =
    Node(1, Node(2, Node(3, Empty())))
val list2: LinkedList<String> =
    Node("A", Node("B", Empty()))
```

One problem with this implementation is that we need to create an instance of `Empty` every time we create a list. Instead, we should just have one instance and use it universally. The only problem is the generic type. What generic type should we use? We want this empty list to be subtype of all lists. We cannot use all types, but we also don't need to. A solution is that we can make it a list of `Nothing`. `Nothing` is a subtype of every type, and so `LinkedList<Nothing>` will be a subtype of every `LinkedList` once this list is covariant (`out` modifier). Making type arguments covariant truly makes sense here since the list is immutable and this type is used only in out positions. So this is the improved code:

```
sealed class LinkedList<out T>

class Node<out T>(
        val head: T,
        val tail: LinkedList<T>
) : LinkedList<T>()

object Empty : LinkedList<Nothing>()

// Usage
val list: LinkedList<Int> =
    Node(1, Node(2, Node(3, Empty)))

val list2: LinkedList<String> =
    Node("A", Node("B", Empty))
```

This is a useful trick that is often used, especially when we define immutable sealed classes. Immutable, because using it for mutable objects can lead to subtle and hard to detect bugs connected to shared state management. The general rule is that mutable objects should not be cached (*Item 1: Limit mutability*). Apart from object declaration, there are more ways to reuse objects. Another one is a factory function with a cache.

Factory function with a cache

Every time we use a constructor, we have a new object. Though it is not necessarily true when you use a factory method. Factory functions can have cache. The simplest case is when a factory function always returns the same object. This is, for instance, how `emptyList` from stdlib is implemented:

```kotlin
fun <T> List<T> emptyList() {
   return EMPTY_LIST;
}
```

Sometimes we have a set of objects, and we return one of them. For instance, when we use the default dispatcher in the Kotlin coroutines library `Dispatchers.Default`, it has a pool of threads, and whenever we start anything using it, it will start it in one that is not in use. Similarly, we might have a pool of connections with a database. Having a pool of objects is a good solution when object creation is heavy, and we might need to use a few mutable objects at the same time.

Caching can also be done for parameterized factory methods. In such a case, we might keep our objects in a map:

```kotlin
private val connections =
    mutableMapOf<String, Connection>()

fun getConnection(host: String) =
    connections.getOrPut(host) { createConnection(host) }
```

Caching can be used for all pure functions. In such a case, we call it memorization. Here is, for instance, a function that calculates the Fibonacci number at a position based on the definition:

```
private val FIB_CACHE = mutableMapOf<Int, BigInteger>()

fun fib(n: Int): BigInteger = FIB_CACHE.getOrPut(n) {
   if (n <= 1) BigInteger.ONE else fib(n - 1) + fib(n - 2)
}
```

Now our method during the first run is nearly as efficient as a linear solution, and later it gives result immediately if it was already calculated. Comparison between this and classic linear fibonacci implementation is presented in the below graph and table. Also the iterative implementation we compare it to is presented below.

	n = 100	**n = 200**	**n = 300**	**n = 400**
fibIter	1997 ns	5234 ns	7008 ns	9727 ns
fib (first)	4413 ns	9815 ns	15484 ns	22205 ns
fib (later)	8 ns	8 ns	8 ns	8 ns

```
fun fibIter(n: Int): BigInteger {
   if(n <= 1) return BigInteger.ONE
   var p = BigInteger.ONE
   var pp = BigInteger.ONE
   for (i in 2..n) {
       val temp = p + pp
       pp = p
       p = temp
   }
   return p
}
```

You can see that using this function for the first time is slower than using the classic approach as there is additional overhead on checking if the value is in the cache and adding it there. Once values are added, the retrieval is nearly instantaneous.

It has a significant drawback though: We are reserving and using more memory since the `Map` needs to be stored somewhere. Everything would be fine if this was cleared at some point. But take into account that for the Garbage Collector (GC), there is no difference between a cache and any other static field that might be necessary in the future. It will hold this data as long as possible, even if we never use the `fib` function again. One thing that helps is using a soft reference that can be removed by the GC when memory is needed. It should not be confused with weak reference. In simple words, the difference is:

- Weak references do not prevent Garbage Collector from cleaning-up the value. So once no other reference (variable) is using it, the value will be cleaned.
- Soft references are not guaranteeing that the value won't be cleaned up by the GC either, but in most JVM implementations, this value won't be cleaned unless memory is needed. Soft references are perfect when we implement a cache.

Designing a cache well is not easy, and in the end, caching is always a tradeoff: performance for memory. Remember this, and use caches wisely. No-one wants to move from performance issues to lack of memory issues.

Heavy object lifting

A very useful trick for performance is lifting a heavy object to an outer scope. Surely, we should lift if possible all heavy operations from collection processing functions to a general processing scope. For instance, in this function we can see that we will need to find the maximal element for every element in the `Iterable`:

```kotlin
fun <T: Comparable<T>> Iterable<T>.countMax(): Int =
    count { it == this.max() }
```

A better solution is to extract the maximal element to the level of `countMax` function:

```kotlin
fun <T: Comparable<T>> Iterable<T>.countMax(): Int {
   val max = this.max()
   return count { it == max }
}
```

This solution is better for performance because we will not need to find the maximal element on the receiver on every iteration. Notice that it also improves readability by making it visible that `max` is called on the extension receiver and so it is the same through all iterations.

Extracting a value calculation to an outer scope to not calculate it is an important practice. It might sound obvious, but it is not always so clear. Just take a look at this function where we use a regex to validate if a string contains a valid IP address:

```kotlin
fun String.isValidIpAddress(): Boolean {
   return this.matches("\\A(?:(?:25[0-5]|2[0-4][0-9]|[01]
?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?
)\\z".toRegex())
}

// Usage
print("5.173.80.254".isValidIpAddress()) // true
```

The problem with this function is that `Regex` object needs to be created every time we use it. It is a serious disadvantage since regex pattern compilation is a complex operation. This is why this function is not suitable to be used repeatedly in performance-constrained parts of our code. Though we can improve it by lifting regex up to the top-level:

```kotlin
private val IS_VALID_EMAIL_REGEX = "\\A(?:(?:25[0-5]|2[0-
4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\
)\\z".toRegex()

fun String.isValidIpAddress(): Boolean =
    matches(IS_VALID_EMAIL_REGEX)
```

If this function is in a file together with some other functions and we don't want to create this object if it is not used, we can even initialize the regex lazily:

```
private val IS_VALID_EMAIL_REGEX by lazy { "\\A(?:(?:25[0-5]|2[0-4][0-9\
]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\z".\
toRegex()
}
```

Making properties lazy is also useful when we are dealing with classes.

Lazy initialization

Often when we need to create a heavy class, it is better to do that lazily. For instance, imagine that class `A` needs instances of `B`, `C`, and `D` that are heavy. If we just create them during class creation, `A` creation will be very heavy, because it will need to create `B`, `C` and `D` and then the rest of its body. The heaviness of objects creation will just accumulates.

```
class A {
    val b = B()
    val c = D()
    val d = D()

    //...
}
```

There is a cure though. We can just initialize those heavy objects lazily:

```
class A {
    val b by lazy { B() }
    val c by lazy { C() }
    val d by lazy { D() }

    //...
}
```

Each object will then be initialized just before its first usage. The cost of those objects creation will be spread instead of accumulated.

Keep in mind that this sword is double-edged. You might have a case where object creation can be heavy, but you need methods to be as fast as possible. Imagine that `A` is a controller in a backend application that responds to HTTP calls. It starts quickly, but the first call requires all heavy objects initialization. So the first call needs to wait significantly longer for the response, and it doesn't matter for how long our application runs. This is not the desired behavior. This is also something that might clutter our performance tests.

Using primitives

In JVM we have a special built-in type to represent basic elements like number or character. They are called primitives, and they are used by Kotlin/JVM compiler under the hood wherever possible. Although there are some cases where a wrapped class needs to be used instead. The two main cases are:

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

Knowing that, you can optimize your code to have primitives under the hood instead of wrapped types. Such optimization makes sense mainly on Kotlin/JVM and on some flavours of Kotlin/Native. Not at all on Kotlin/JS. It also needs to be remembered that it makes sense only when operations on a number are repeated many times. Access of both primitive and wrapped types are relatively really fast compared to other operations. The difference manifests itself when we deal with a really big collections (we will discuss it in the *Item 48: Consider Arrays with primitives for performance critical processing*) or when we operate on an object intensively. Also remember that forced changes might lead to less readable code. **This is why I suggest this optimization only for performance critical parts of our code and in libraries**. You can find out what is performance critical using a profiler.

To see an example, imagine that you implement a standard library for Kotlin, and you want to introduce a function that will return the maximal element or `null` if this iterable is empty. You don't want to iterate over the iterable more than once. This is not a trivial problem, but it can be solved with the following function:

```kotlin
fun Iterable<Int>.maxOrNull(): Int? {
    var max: Int? = null
    for (i in this) {
        max = if(i > (max ?: Int.MIN_VALUE)) i else max
    }
    return max
}
```

This implementation has serious disadvantages:

1. We need to use an Elvis operator in every step
2. We use a nullable value, so under the hood in JVM, there will be an `Integer` instead of an `int`.

Resolving these two problems requires us to implement the iteration using while loop:

```kotlin
fun Iterable<Int>.maxOrNull(): Int? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var max: Int = iterator.next()
    while (iterator.hasNext()) {
        val e = iterator.next()
        if (max < e) max = e
    }
    return max
}
```

For a collection of elements from 1 to 10 million, in my computer, the optimized implementation took 289 ms, while the previous one took 518 ms. This is nearly 2 times faster, but remember that this is an extreme case designed to show the difference. Such optimization is rarely reasonable in a code that is not performance-critical. Though if you implement a Kotlin standard library, everything is performance critical. This is why the second approach is chosen here:

```
/**
* Returns the largest element or `null` if there are no elements.
*/
public fun <T : Comparable<T>> Iterable<T>.max(): T? {
   val iterator = iterator()
   if (!iterator.hasNext()) return null
   var max = iterator.next()
   while (iterator.hasNext()) {
       val e = iterator.next()
       if (max < e) max = e
   }
   return max
}
```

Summary

In this item, you've seen different ways to avoid object creation. Some of them are cheap in terms of readability: they should be used freely. For instance, heavy object extraction is a good idea. It is not only a good practice for performance, but also thanks to this extraction, we can name this object so we make our function easier to read. Those optimizations that are tougher or require bigger changes should possibly be skipped. We should avoid premature optimization unless we have such guidelines in our project or we develop a library that might be used in who-knows-what-way. We've also learned some optimizations that might be used to optimize the performance critical parts of our code.