Philipp Hauer's Blog (https://blog.philipphauer.de/)

Web Architecture, Java Ecosystem, Software Craftsmanship

Idiomatic Kotlin. Best Practices.

POSTED ON MAR 28, 2017

In order to take full advantage of Kotlin, we have to revisit some best practices we got used to in Java. Many of them can be replaced with better alternatives that are provided by Kotlin. Let's see how we can write idiomatic Kotlin code and do things the Kotlin way.

A word of **warning**: The following list is not exhaustive and does only express my humble opinion. Moreover, some Kotlin features should be used with sound judgment. If overused, they can make our code even harder to read. For instance, when you create a "train wreck" by trying to squeeze everything into a single unreadable expression.

Kotlin's Built-in Support for Common Java Idioms and Patterns

In Java, we have to write quite some boilerplate code to implemented certain idioms and patterns. Fortunately, many patterns are built-in right into Kotlin (/kotlin-java-ecosystem-language/)'s language or its standard library.

Java Idiom or Pattern	Idiomatic Solution in Kotlin
Optional	Nullable Types

Java Idiom or Pattern	Idiomatic Solution in Kotlin
Getter, Setter, Backing Field	Properties
Static Utility Class	Top-Level (extension) functions
Immutability, Value Objects	`data class` with immutable properties, `copy()`
Fluent Setter (Wither)	Named and default arguments, `apply()`
Method Chaining	Default arguments
Singleton	`object`
Delegation	Delegated properties `by`
Lazy Initialization (thread-safe)	Delegated properties `by`: `lazy()`
Observer	Delegated properties `by`: `Delegates.observable()`

Functional Programming

Among other advantages, functional programming allows us to reduce side-effects, which in turn makes our code…

- less error-prone,
- easier to understand,
- easier to test and
- thread-safe.

In contrast to Java 8, Kotlin has way better support for functional programming:

- Immutability: `val` for variables and properties, immutable data classes, `copy()`
- Expressions: Single expression functions. `if`, `when` and `try-catch` are expressions. We can combine these control structures with other expressions concisely.
- Function Types
- Concise Lambda Expressions
- Kotlin's Collection API

These features allow writing functional code in a safe, concise and expressive way. Consequently, we can create pure functions (functions without side-effects) more easily.

Use Expressions

```
// Don't
fun getDefaultLocale(deliveryArea: String): Locale {
    val deliverAreaLower = deliveryArea.toLowerCase()
    if (deliverAreaLower == "germany" || deliverAreaLower == "austria") {
        return Locale.GERMAN
    }
    if (deliverAreaLower == "usa" || deliverAreaLower == "great britain") {
        return Locale.ENGLISH
    }
    if (deliverAreaLower == "france") {
        return Locale.FRENCH
    }
    return Locale.ENGLISH
}
```

```
// Do
fun getDefaultLocale2(deliveryArea: String) = when (deliveryArea.toLowerCase()) {
    "germany", "austria" -> Locale.GERMAN
    "usa", "great britain" -> Locale.ENGLISH
    "france" -> Locale.FRENCH
    else -> Locale.ENGLISH
}
```

Rule of thumb: Every time you write an `if` consider if it can be replaced with a more concise `when` expression.

`try-catch` is also a useful expression:

```
val json = """{"message":"HELLO"}"""
val message = try {
    JSONObject(json).getString("message")
} catch (ex: JSONException) {
    json
}
```

Top-Level (Extension) Functions for Utility Functions

In Java, we often create static util methods in util classes. A direct translation of this pattern to Kotlin would look like this:

```
//Don't
object StringUtil {
    fun countAmountOfX(string: String): Int{
        return string.length - string.replace("x", "").length
    }
}
StringUtil.countAmountOfX("xFunxWithxKotlinx")
```

Kotlin allows removing the unnecessary wrapping util class and use top-level functions instead. Often, we can additionally leverage extension functions, which increases readability. This way, our code feels more like “telling a story”.

```
//Do
fun String.countAmountOfX(): Int {
    return length - replace("x", "").length
}
"xFunxWithxKotlinx".countAmountOfX()
```

Named Arguments instead of Fluent Setter

Back in Java, fluent setters (also called “Wither”) where used to simulate named and default arguments and to make huge parameter lists more readable and less error-prone:

```
//Don't
val config = SearchConfig()
       .setRoot("~/folder")
       .setTerm("kotlin")
       .setRecursive(true)
       .setFollowSymlinks(true)
```

In Kotlin, named and default arguments fulfil the same propose but are built directly into the language:

```
//Do
val config2 = SearchConfig2(
       root = "~/folder",
       term = "kotlin",
       recursive = true,
       followSymlinks = true
)
```

`apply()` for Grouping Object Initialization

```
//Don't
val dataSource = BasicDataSource()
dataSource.driverClassName = "com.mysql.jdbc.Driver"
dataSource.url = "jdbc:mysql://domain:3309/db"
dataSource.username = "username"
dataSource.password = "password"
dataSource.maxTotal = 40
dataSource.maxIdle = 40
dataSource.minIdle = 4
```

The extension function `apply()` helps to group and centralize initialization code for an object. Besides, we don’t have to repeat the variable name over and over again.

```
//Do
val dataSource = BasicDataSource().apply {
    driverClassName = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://domain:3309/db"
    username = "username"
    password = "password"
    maxTotal = 40
    maxIdle = 40
    minIdle = 4
}
```

`apply()` is often useful when dealing with Java libraries in Kotlin.

Don’t Overload for Default Arguments

Don’t overload methods and constructors to realize default arguments (so called “method chaining” or “constructor chaining”).

```
//Don't
fun find(name: String){
    find(name, true)
}
fun find(name: String, recursive: Boolean){
}
```

That is a crutch. For this propose, Kotlin has named arguments:

```
//Do
fun find(name: String, recursive: Boolean = true){
}
```

In fact, default arguments remove nearly all use cases for method and constructor *overloading* in general, because overloading is mainly used to create default arguments.

Concisely Deal with Nullability

Avoid `if-null` Checks

The Java way of dealing with nullability is cumbersome and easy to forget.

```
//Don't
if (order == null || order.customer == null || order.customer.address == null){
    throw IllegalArgumentException("Invalid Order")
}
val city = order.customer.address.city
```

Every time you write an `if-null` check, hold on. Kotlin provides much better ways to handle nulls. Often, you can use a null-safe call `?.` or the elvis operator `?:` instead.

```
//Do
val city = order?.customer?.address?.city ?: throw IllegalArgumentException("Invalid O
rder")
```

Avoid `if-type` Checks

The same is true for `if-type` -check.

```
//Don't
if (service !is CustomerService) {
    throw IllegalArgumentException("No CustomerService")
}
service.getCustomer()
```

Using `as?` and `?:` we can check the type, (smart-)cast it and throw an exception if the type is not the expected one. All in one expression!

```
//Do
service as? CustomerService ?: throw IllegalArgumentException("No CustomerService")
service.getCustomer()
```

Avoid not-null Assertions `!!`

```
//Don't
order!!.customer!!.address!!.city
```

> *"You may notice that the double exclamation mark looks a bit rude: it's almost like you're yelling at the compiler. This is intentional. The designers of Kotlin are trying to nudge you toward a better solution that doesn't involve making assertions that can't be verified by the compiler."* 'Kotlin in Action' by Dmitry Jemerov and Svetlana Isakova

Consider `let()`

Sometimes, using `let()` can be a concise alternative for `if`. But you have to use it with sound judgment in order to avoid unreadable "train wrecks". Nevertheless, I really want you to consider using `let()`.

```
val order: Order? = findOrder()
if (order != null){
    dun(order.customer)
}
```

With `let()`, there is no need for an extra variable. So we get along with one expression.

```
findOrder()?.let { dun(it.customer) }
//or
findOrder()?.customer?.let(::dun)
```

Leverage Value Objects

With data classes, writing immutable value objects is so easy. Even for value objects containing only a single property. So there is no excuse for not using value objects anymore!

```
//Don't
fun send(target: String){}

//Do
fun send(target: EmailAddress){}
// expressive, readable, type-safe

data class EmailAddress(val value: String)
```

Concise Mapping with Single Expression Functions

```
// Don't
fun mapToDTO(entity: SnippetEntity): SnippetDTO {
    val dto = SnippetDTO(
            code = entity.code,
            date = entity.date,
            author = "${entity.author.firstName} ${entity.author.lastName}"
    )
    return dto
}
```

With single expression functions and named arguments we can write easy, concise and readable mappings between objects.

```
// Do
fun mapToDTO(entity: SnippetEntity) = SnippetDTO(
        code = entity.code,
        date = entity.date,
        author = "${entity.author.firstName} ${entity.author.lastName}"
)
val dto = mapToDTO(entity)
```

If you prefer extension functions, you can use them here to make both the function definition and the usage even shorter and more readable. At the same time, we don't pollute our value object with the mapping logic.

```
// Do
fun SnippetEntity.toDTO() = SnippetDTO(
        code = code,
        date = date,
        author = "${author.firstName} ${author.lastName}"
)
val dto = entity.toDTO()
```

Refer to Constructor Parameters in Property Initializers

Think twice before you define a constructor body (`init` block) only to initialize properties.

```
// Don't
class UsersClient(baseUrl: String, appName: String) {
    private val usersUrl: String
    private val httpClient: HttpClient
    init {
        usersUrl = "$baseUrl/users"
        val builder = HttpClientBuilder.create()
        builder.setUserAgent(appName)
        builder.setConnectionTimeToLive(10, TimeUnit.SECONDS)
        httpClient = builder.build()
    }
    fun getUsers(){
        //call service using httpClient and usersUrl
    }
}
```

Note that we can refer to the primary constructor parameters in property initializers (and not only in the `init` block). `apply()` can help to group initialization code and get along with a single expression.

```
// Do
class UsersClient(baseUrl: String, appName: String) {
    private val usersUrl = "$baseUrl/users"
    private val httpClient = HttpClientBuilder.create().apply {
        setUserAgent(appName)
        setConnectionTimeToLive(10, TimeUnit.SECONDS)
    }.build()
    fun getUsers(){
        //call service using httpClient and usersUrl
    }
}
```

`object` for Stateless Interface Implementations

Kotlin's `object` comes in handy when we need to implement a framework interface that doesn't have any state. For instance, Vaadin 8's `Converter` interface.

```
//Do
object StringToInstantConverter : Converter<String, Instant> {
    private val DATE_FORMATTER = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm:ss Z")
            .withLocale(Locale.UK)
            .withZone(ZoneOffset.UTC)

    override fun convertToModel(value: String?, context: ValueContext?) = try {
        Result.ok(Instant.from(DATE_FORMATTER.parse(value)))
    } catch (ex: DateTimeParseException) {
        Result.error<Instant>(ex.message)
    }

    override fun convertToPresentation(value: Instant?, context: ValueContext?) =
            DATE_FORMATTER.format(value)
}
```

For further information about the synergies between Kotlin, Spring Boot and Vaadin, check out this blog post (/kotlin-practice-spring-boot-vaadin/).

Destructuring

On the one hand, destructuring is useful for returning multiple values from a function. We can either define an own data class (which is the preferred way) or use `Pair` (which is less expressive, because `Pair` doesn't contain semantics).

```
//Do
data class ServiceConfig(val host: String, val port: Int)
fun createServiceConfig(): ServiceConfig {
    return ServiceConfig("api.domain.io", 9389)
}
//destructuring in action:
val (host, port) = createServiceConfig()
```

On the other hand, destructuring can be used to concisely iterate over a map:

```
//Do
val map = mapOf("api.domain.io" to 9389, "localhost" to 8080)
for ((host, port) in map){
    //...
}
```

Ad-Hoc Creation of Structs

`listOf`, `mapOf` and the infix function `to` can be used to create structs (like JSON) quite concisely. Well, it’s still not as compact as in Python or JavaScript, but way better than in Java.

```
//Do
val customer = mapOf(
        "name" to "Clair Grube",
        "age" to 30,
        "languages" to listOf("german", "english"),
        "address" to mapOf(
                "city" to "Leipzig",
                "street" to "Karl-Liebknecht-Straße 1",
                "zipCode" to "04107"
        )
)
```

But usually, we should use data classes and object mapping to create JSON. But sometimes (e.g. in tests) this is very useful.

Source Code

You can find the source code in my GitHub project idiomatic kotlin (https://github.com/phauer/blog-related/tree/master/kotlin-idiomatic).

Related Posts

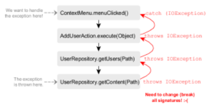
(https://blog.philipphauer.de/clean-code-kotlin/)

Clean Code with Kotlin

(https://blog.philipphauer.de/kotlin-practice-spring-boot-vaadin/)

Kotlin in Practice with Spring Boot and Vaadin

(https://blog.philipphauer.de/kotlin-java-ecosystem-language/)
Kotlin. The Java Ecosystem deserves this Language.


We want to handle the exception here!
ContextMenu.menuClicked()
catch (IOException)
AddUserAction.execute(Object)
throws IOException
UserRepository.getUsers(Path)
throws IOException
The exception is thrown here.
UserRepository.getContent(Path)
throws IOException
Need to change (break) all signatures! :-(

(https://blog.philipphauer.de/checked-exceptions-are-evil/)

Checked Exceptions are Evil

This entry was posted in Software Craftsmanship (/categories/software-craftsmanship/), and tagged with Kotlin (/tags/kotlin/), Clean Code (/tags/clean-code/),

Comments

32 Comments blog-philipphauer 1 Login

Recommend 23 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

jesse hodges • 6 months ago

I would add a section on using `apply` when working with java bean-style classes -

eg instead of

val person = Person()
person.firstName ="foo"
person.lastName="bar"

do

val person = Person().apply {
firstName="foo"
lastName="bar"
}

3 • Reply • Share ›

Philipp Hauer Mod → jesse hodges • 6 months ago

Hi jesse. Good point. I also really like to use apply(). I already recommend using apply() in my other blog posts. In this post I point to apply() in the section "Refer to Constructor Parameters in Property Initializers" and in the introduction table as a alternative for fluent setters. However, it should have its own section. Thanks!

• Reply • Share ›

Mykola Polonskyi → Philipp Hauer • 4 months ago

But apply won't be applicable for using with lateinit vars(e.g. in case with Spring Autiwired)? what do u think?

• Reply • Share ›

Philipp Hauer Mod ➦ Mykola Polonskyi • 4 months ago

Nope, because Spring initializes the object for you. Nobody else should do it. So there is no need for additional initialization logic with apply().

• Reply • Share ›

takahirom • 3 months ago

Thank you for a great article. If you do not mind, can I translate the article into Japanese?

1 • Reply • Share ›

Andreas Volkmann ➦ takahirom • 3 months ago

Please translate :) 翻訳を読んでみたです。

• Reply • Share ›

takahirom ➦ Andreas Volkmann • 3 months ago

Thanks :)

• Reply • Share ›

kenyee • 6 months ago

nicely done!
Something like this should be on the official Kotlin site...most of it is obvious after you've looked at enough Kotlin code, but it's nice to have it spelled out :-)

1 • Reply • Share ›

David Felix • 6 months ago

I think the idea that nullable types should be used as a replacement for Java's optional type risks missing the fact that Kotlin has sealed classes, which are even better than optionals. Sure, nullable types are checked like optional and act as a slightly safer equivalent to them so you can safely interact with something that could be null, that's true. If you use sealed classes, you can actually represent an alternate return type (or a "none", like optional) which provides a caller with more information than simply that the function returned nothing. The main difference for the caller would be needing to use `when` to determine the type of the sealed class. While nullable types are more frequently useful, you have a lot more options for how to best represent null in Kotlin.

1 • Reply • Share ›

Kiran Rao • 6 months ago

Great post! These are the kind of best practices that you only gain with experience!

One tiny nit-pick though: Destruction has an overloaded meaning (finalizers in Java or destruction in C++). You probably want to re-word it as de-structuring.

1 • Reply • Share ›

Philipp Hauer Mod ➦ Kiran Rao • 6 months ago

Hi Kiran, thank you for the hint! You are right, it's "destructuring". I just fixed this typo.

• Reply • Share ›

Kirill Rakhman • 6 months ago

Great post! Small suggestion regarding `let`: it looks even better when you can use it with function references. Instead of `findOrder()?.let { dun(it.customer) }`, you can write `findOrder()?.customer?.let(::dun)`.

1 • Reply • Share ›

Philipp Hauer Mod ➦ Kirill Rakhman • 6 months ago

Hi Kirill, thanks for your reply! You are right, using a method reference in this case is even better. I updated the post accordingly.

• Reply • Share ›

billy bibbit • 3 months ago

Can you give a quick example of this:

"But usually, we should use data classes and object mapping to create JSON."

Thanks

• Reply • Share ›

Philipp Hauer Mod ➦ billy bibbit • 3 months ago

Check out this out: https://github.com/FasterXM...

Reply • Share ›

billy bibbit ➦ Philipp Hauer • 3 months ago

Thanks!

Reply • Share ›

Tomoaki Imai • 3 months ago

Hi, thanks for the great post! I was curious how you differentiate the usage of Top-Level (Extension) Functions and `object`. A utility class is usually stateless.

Reply • Share ›

Philipp Hauer Mod ➦ Tomoaki Imai • 3 months ago

Sometimes you need an instance of a class (not only utility functions). A framework may require this. For instance, in Vaadin you need to implement a certain Converter interface (with multiple methods) and assign it to GUI fields. Often, converter implementations have no state. In this case, object is a perfect fit. You can't do this with top-level functions.
Another small advantage of object is that you can group methods, fields and constants that belong together into one unit. This arrangement states clearly: this fields belong together and are only used within this scope. This leads to high cohesion.
Check this out: https://blog.philipphauer.d...

Reply • Share ›

Tomoaki Imai ➦ Philipp Hauer • 3 months ago

> Sometimes you need an instance of a class

I see, that makes sense. It all cleared now! And also thanks for the reference, it's really helpful :)

Reply • Share ›

Leon • 4 months ago

Great article!

Reply • Share ›

Trần Đức Tâm (Rikimaru) • 4 months ago

//Do
fun (name: String, recursive: Boolean = true){
}
=> Function declaration must have a name

Reply • Share ›

Lukas Lechner • 5 months ago

Very nice and concise post! Bookmarked for reference!

Reply • Share ›

Stepan Goncharov • 6 months ago

NotNull assertions - also handy tool for implementing fail-fast technique, you shouldn't avoid it in any cost.
Most common use-case of `object` is Singleton implementation.
Type aliases could be also used as a replacement for Value objects in some cases.

Reply • Share ›

Stepan Goncharov • 6 months ago

Nullable types could be used in some cases instead of Optional, however nullable types just more expressive way to annotate your type as @Nullable/@NonNull. You should remember that in reality there is no guaranty that there would be no NPE in runtime.
Optional is different pattern that give you more guaranties about it's internal state. Try to use nullable types with rxJava2 for example;)

Reply • Share ›

Philipp Hauer Mod ➦ Stepan Goncharov • 6 months ago

With Kotlin's nullable types, a) the compiler forces me to handle the case of null values and b) we get a nice API and language features to handle this cases. For me, this makes nullable types much more like Optionals than @Nullable.

What guaranties do you mean?

• Reply • Share ›

Daniil Vodopian • 6 months ago

Hmmm...
Don't use `if` in favor of `let`? Why?

I find `let` very unreadable in this situation. When I deal with single properties (not big expressions), I prefer `if` since it communicates my intent better.

• Reply • Share ›

Philipp Hauer Mod Daniil Vodopian • 6 months ago

Hi Daniil, thanks for your feedback. I absolutely agree with you. let() may encourage you to squeeze everything into a single expression which makes things even worse. However, it's sometimes really handy. I updated my post to be less dogmatic at this point.

1 • Reply • Share ›

Daniil Vodopian Philipp Hauer • 6 months ago

Very nice, thank you!

`let` has an unfortunate problem that it is associated with Swift's `if-let` and "optional unwrapping". Kotlin's smart casting allows for many interesting patterns that unwrapping doesn't allow, but developers, coming from "unwrapping" languages, do not always see them.

• Reply • Share ›

Rafał Kotusiewicz • 6 months ago

//Do
val city = order?.customer?.address?.city ?: throw IllegalArgumentException("Invalid Order")

This piece

order?.customer?.address?.city

is very ugly anti-pattern...

• Reply • Share ›

Philipp Hauer Mod Rafał Kotusiewicz • 6 months ago

No, it isn't an anti-pattern. For me, it's a powerful idiom. Why do you think so?

• Reply • Share ›

Rafał Kotusiewicz Philipp Hauer • 6 months ago

Please search for "Inappropriate Intimacy code smell" in google. In your example object which calling

order?.customer? etc

know too much about structure of object order. Classes should know as little as possible about each other.
For more info:
- "Design patterns" GoF
- "Refactoring" M. Fowler
- "Refactoring to Patterns" J. Jerievsky

• Reply • Share ›

Philipp Hauer Mod Rafał Kotusiewicz • 6 months ago

Right, that's the law of demeter which is a good principle. But it says that you should encapsulate this knowledge at a single point (e.g. behind a method getOrderCity()). And all clients should not have this knowledge - they only have to call this method. Right. But _somewhere_ (in getOrderCity()) you _have to_ access the order's city. And in this case, the mentioned Kotlin idiom is extremely powerful. Nobody says that just because there is these smart notation in Kotlin, you should dig deeply into a model from everywhere.

Please don't mix up language features and good design. Design and discipline is still important. Even with Kotlin you have to think about good design.

• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

Η πιο δημοφιλής θεραπεία για την κακή μυρωδιά των ποδιών...

Αν θέλετε να απαλλαγείτε από την άσχημη μυρωδιά τότε ανακατέψτε νερό με...

Learn More

Sponsored by **Onycosolve**

Report ad

Philipp Hauer

I am Philipp Hauer (M.Sc.) and I work as a software engineer for Spreadshirt (https://www.spreadshirt.com/) in Leipzig, Germany. I focus on developing JVM-based web applications and I'm enthusiastic about Kotlin, clean code, web architectures and microservices. I'm tweeting under @philipp_hauer (https://twitter.com/philipp_hauer).

(mailto:xxblxxogxx@pxxhilippxxhauer.dxxe) (https://github.com/phauer) (https://www.linkedin.com/in/philipp-hauer-677738135) (https://twitter.com/philipp_hauer) (https://www.xing.com/profile/Philipp_Hauer3)

‹ RESTful API Documentation with Swagger and AsciiDoc (https://blog.philipphauer.de/rest-api-documentation-swagger-asciidoc/)
Talk 'Kotlin in Practice' at the JUG Saxony Camp 2017 in Leipzig › (https://blog.philipphauer.de/talk-jug-saxony-camp-2017/)

Recent Posts

Talk 'Kotlin in Practice' at the JUG Saxony Day 2017 in Dresden (/talk-jug-saxony-day-2017/)
Don't use In-Memory Databases (like H2) for Tests (/dont-use-in-memory-databases-tests-h2/)
Debugging within a PHP Docker Container using IDEA/PhpStorm and Xdebug (/debug-php-docker-container-idea-phpstorm/)
A Test Mail Server for a PHP Docker Container (/test-mail-server-php-docker-container/)
Clean Code with Kotlin (/clean-code-kotlin/)

Categories

Web Development (/Categories/Web-Development/) (11)
Build And Development Infrastructure (/Categories/Build-And-Development-Infrastructure/) (10)
Software Architecture (/Categories/Software-Architecture/) (8)
Software Craftsmanship (/Categories/Software-Craftsmanship/) (8)

Database (/Categories/Database/) (6)
Publications And Talks (/Categories/Publications-And-Talks/) (5)
Tools And Environment (/Categories/Tools-And-Environment/) (2)

Top Tags

Docker (/Tags/Docker/) (8)
Build (/Tags/Build/) (6)
Continuous Integration And Continuous Delivery (/Tags/Continuous-Integration-And-Continuous-Delivery/) (5)
Kotlin (/Tags/Kotlin/) (4)
Maven (/Tags/Maven/) (4)
Microservices (/Tags/Microservices/) (4)
Rest (/Tags/Rest/) (4)
Best Practices (/Tags/Best-Practices/) (3)
Clean Code (/Tags/Clean-Code/) (3)
Mongodb (/Tags/Mongodb/) (3)
Nosql (/Tags/Nosql/) (3)
Relational Databases (/Tags/Relational-Databases/) (3)