A MAGAZINE FOR KOTLIN™ DEVELOPERS

fun code()

23 MAY 2024

LIMITED ONE-TIME EDITION

BRAND NEW

Build it Yourself

3D Kotlin Logo

Page 6

iOS

Spotlight

What iOS developers say about Kotlin Multiplatform

Page 8

Fun

Crossword Puzzle

Page 19

Page 10 Mario Bodemann

With Koonlander Typealong

EXCLUSIVE!

Martin Bonnin Page 16

What kind of Kotliner are You?

Kotlynne

The Future of Kotlin, with the History of English

Chet Haase Page 7

The *Why* of Code Sharing

Marc Reichelt Page 5

Identifiers aren't Services

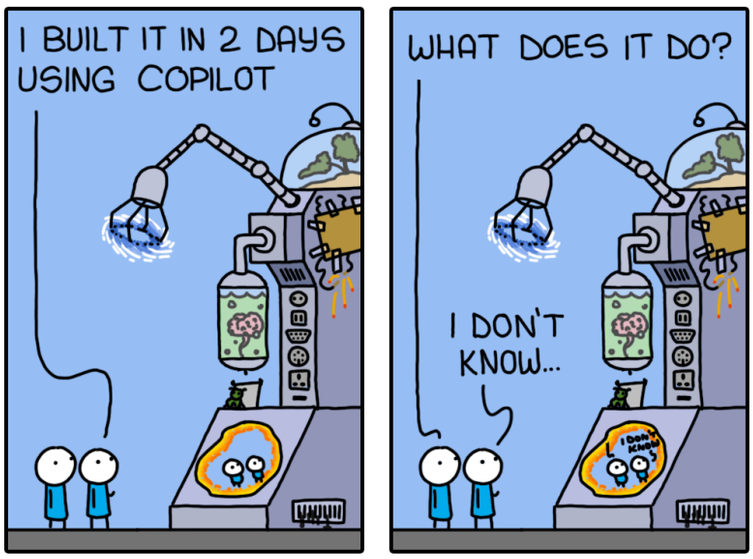
Jesse Wilson Page 4

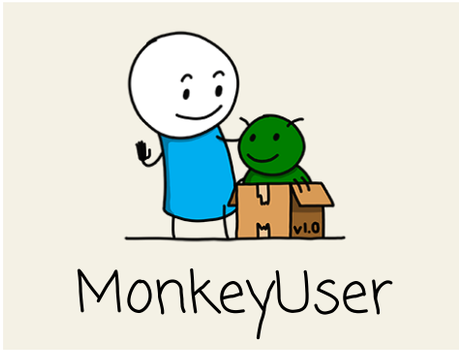
HTML to PDF with Kotlin/JVM

Mariusz Sołtysiak Page 12

AI ASSISTANT

MONKEYUSER.COM



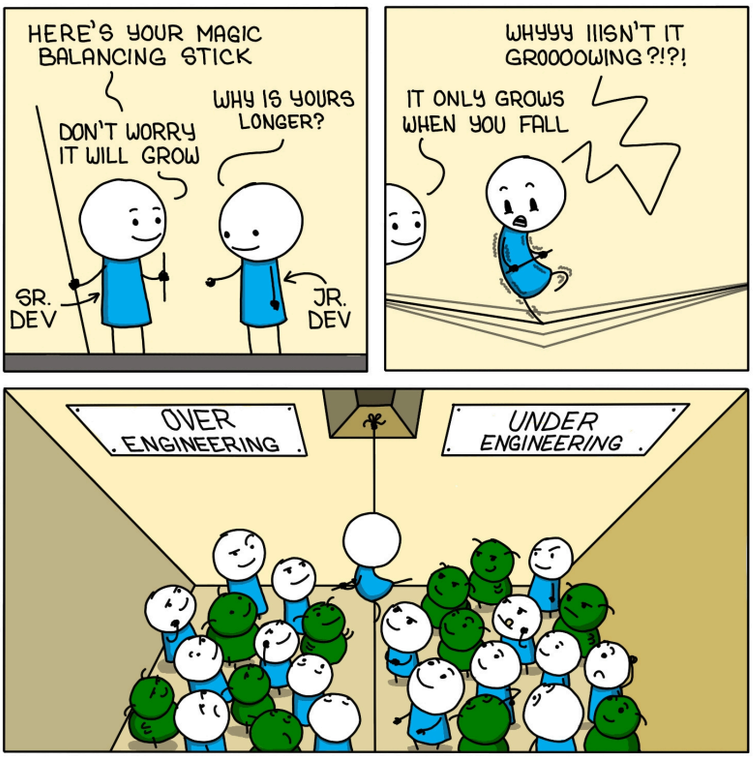


monkeyuser.com

store.monkeyuser.com

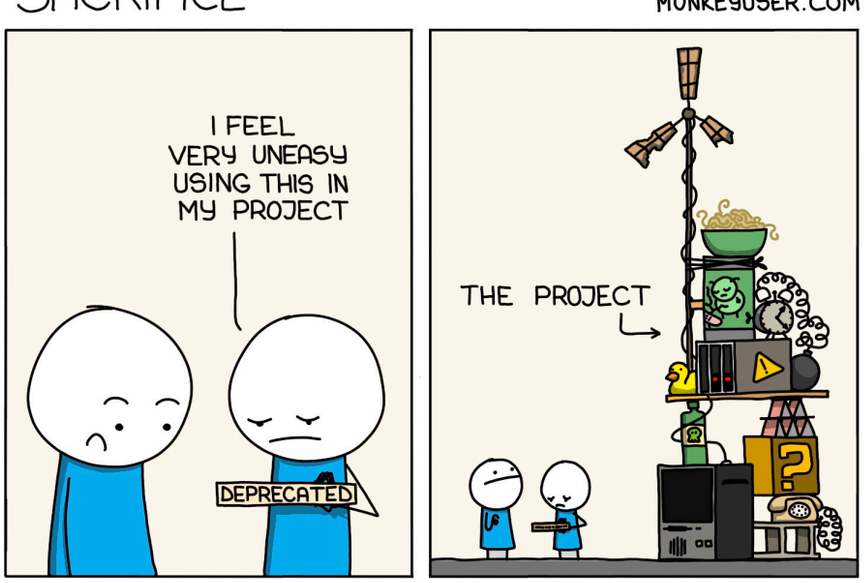
BALANCING STICK

MONKEYUSER.COM



SACRIFICE

MONKEYUSER.COM



Big thanks for the cartoons, MonkeyUser! ❤️

Support the author here:
patreon.com/monkeyuser

Nature's own tribute to Kotlin: the polar lights in soft, familiar hues as seen in Frankfurt on May 10th, 2024

It's a great time to be a Kotlin developer. The new K2 compiler is finally stable. K2 brings a significant performance upgrade, and it starts a new era: the development of new language features beyond Kotlin 2.0. With Google focusing on Kotlin first for Android and Compose, millions of developers have jumped on the Kotlin train. Meanwhile, Kotlin Multiplatform has evolved to stable, and we are seeing more foundations and even big UI frameworks like Compose being made multiplatform-ready. And the exceptionally great IDEs by JetBrains make writing Kotlin a breeze. All of these not just increase the worth of your Kotlin code, but it also increases your worth as a Kotlin developer!

To top it all off, Kotlin has one of the best communities I have ever experienced. It's a great mix of kindness, high competence, approachability, fun, will to improve, and just the right touch of craziness.

Take this magazine for example. It started out as a vague idea to create just a cover as an April's joke, and when I told the idea to people from JetBrains at KotlinConf 2023 and earned laughter I knew we had to make it happen (check kotlintoday.com if you haven't seen it yet). The idea grew. I thought: Wouldn't it be a nice surprise for people if we actually launched a real magazine just in time for KotlinConf 2024 - with actual content? Something tangible, which is not happening so often these days. We assembled a small team from the Kotlin community, with one mission: to build a magazine that we would enjoy reading ourselves. I can say I have never been more proud of the Kotlin community!

My gratitude goes to everyone who made this magazine possible: Thank you, you're amazing! ❤️

And to you, dear readers: Whether you're just starting out with Kotlin (oh boy, you're in for a ride!) or you're a Kotlin pro, we hope this magazine brings joy to your heart!

Enjoy the magazine, and have a nice Kotlin. And if you see one of us at KotlinConf: come say hi! 👋

Marc Reichelt

Big thanks to:
Mario Bodemann, Jesse Wilson, Chet Haase, Martin Bonnin, Mariusz Sołtysiak, Nicole Terc, Marc Reichelt, Sigurdur Gunnarsson, Andrey Breslav, Matthias Geisler, Romain Guy, Amanda Hichman, Louis CAD, Enrique López Mañas, Marton Braun, Kevin Galligan, Mohsen Mirhoseini, Erik Hellman, Ash Davies, Yiğit Boyar, Volodymyr Galandzij, Oksana Prokopiv, Martin Höller, Alexander von Below, Grace Ohanian, Ahmed Bouchfaa, Luke Davis, José Jeria, Stefan Herold

And special thanks to Miki Stefanoski from We Are Systematic for helping with design! 👏

Follow us:
github.com/kotlin-magazine/kotlin-magazine

x.com/funcode_mag
mastodon.social/@funcode_mag
kotlintoday.com

Identifiers aren't Services

Jesse Wilson, publicobject.com

Contributor to lots of open-source Kotlin stuff including OkHttp and Zipline

The programs I write frequently involve strings that identify things: email addresses, file paths, URLs, time zones... even credit card numbers and driver's license numbers.

For many years I followed the patterns of the Java standard library when creating my own identifiers.
I've grown to dislike these patterns!

java.net.URL

I'll start by complaining about Java's URL class, again:

```
val url = URL("https://publicobject.com/helloworld.txt")
val content = url.openStream().use { it.readBytes() }
println(content.decodeToString())
```

This does a lot in 3 lines of Kotlin! We identify a URL, fetch its contents, and print them to the console. But despite its compactness, this code is bad.
An HTTP client is hiding in the URL class. When I call *openStream()*, that client is prepared and put to work. I don't like being cut out of that setup! I can't dependency-inject my own configured instance for production or a fake in a test.
java.net.URL is serving two competing purposes: as an identifier and as a service.

Identifiers

Identifiers are values that we do value-like things with:

- Accept as input from a person
- Validate for structure
- Write to a database, file, or remote process
- Assert equality in a test case

Nothing is lost when we send an identifier from one program to another.

Services

While it's easy to pass a URL string from one computer to another, we can't pass the InputStream that reads the response body.
Symmetrically, we can send a timestamp from one computer to another, but we can't send the Clock that produced it. That's because it's a software abstraction over a specific quartz crystal that's bound to the physical world!

java.io.File

Java's original file class identifies both a location on the file system and also operates on that location.
I might write an Android app that sends its collection of cached images to the server as a List<File>. My server could call File.delete() to free up space on that Android device, but that's not what would happen!
Subclassing File is another thing you could do, but shouldn't:

```
class ImmortalFile(delegate: File) : File(delegate.path) {
 override fun delete() = false
}
```

If you want to write testable code that operates on files, consider Okio!

java.nio.Path

The new (2011) file system APIs are on the right track, but you have to be careful to use them in a testable way. Each Path has both a path string (the identifier) and a file system (the service).

This code is implicitly coupled to the default file system:

```
class HelloReader() {
  fun readHello(): String {
    val helloPath = Paths.get("hello.txt")
    return Files.readString(helloPath)
  }
}
```

By changing every call to Paths.get(...) with FileSystem.getPath(...), I can make this testable (such as with Jimfs):

```
class HelloReader(
  val fileSystem: FileSystem,
) {
  fun readHello(): String {
    val helloPath = fileSystem.getPath("hello.txt")
    return Files.readString(helloPath)
  }
}
```

java.net.InetSocketAddress

I get myself into trouble whenever I use Java's Internet address API:

```
// Wrong! This eagerly looks up an IP address for publicobject.com
val connectAddress = InetSocketAddress("publicobject.com", 443)
```

The method to use is *createUnresolved()*:

```
val connectAddress =
    InetSocketAddress.createUnresolved("publicobject.com", 443)
```

Even though I used the same host and port to create these two instances, they don't *.equals()* each other. Unless I'm offline, in which case they do.

kotlinx.datetime.TimeZone

I need to build a report that summarizes the emails that our service sends each day. The input is a set of SentEmail records:

```
data class SentEmail(
  val customerId: Id,
  val timeZone: TimeZone,
  val locale: Locale,
  val emailAddressId: Id,
  val templateId: TemplateId,
  val enqueuedAt: Instant,
  val deliveredAt: Instant,
)
```

The TimeZone class looks like an innocuous way to track a string like "America/New_York" in a type-safe way.
Unfortunately, Kotlin's time zone class throws when it's given a time zone that isn't in its host JVM's time zone database. My report will crash if any customer uses Europe/Kyiv (renamed from Europe/Kiev in 2022).

```
java.time.zone.ZoneRulesException: Unknown time-zone ID: Europe/Kyiv
  at java.time.zone.ZoneRulesProvider.getProvider(ZoneRulesProvider.java)
  at java.time.zone.ZoneRulesProvider.getRules(ZoneRulesProvider.java)
  at java.time.ZoneRegion.ofId(ZoneRegion.java)
  at java.time.ZoneId.of(ZoneId.java)
  at kotlinx.datetime.TimeZone$Companion.of(TimeZoneJvm.kt)
```

I can fix this crash by updating my JVM to one with more up-to-date time zone data. Even when I only need an identifier, TimeZone always loads the offset rules.

Advice

When using an identifier type like File, InetSocketAddress, or TimeZone, pay careful attention to what side effects your identifier is triggering.
When writing your identifier code, please use a data class for the value part and a separate interface for its related services.

The *Why* of Code Sharing

Marc Reichelt
Senior Android Developer at Snapp Mobile

As multiple technologies emerged for code sharing, the *Why* has changed over time. It's good to take a look back in time - to understand what were the primary reasons for developers to share code, what mistakes have been done in the past, and what we can learn from that for today.

Simplification
The primary reason for sharing code has always been to *reduce complexity*. Heck, why would we even want to write programs multiple times in different languages? And keep adding even more bugs while we're at it? That's insane! Thankfully, there were technologies available that ran on multiple platforms. Java™ with its slogan "Write once, run anywhere" made it possible to share programs for the desktop. Even with C/C++, much of the code was compiling on multiple platforms. And with Flash and JavaScript, there were technologies available that allowed developers to share programs for the web. Then the mobile revolution happened.

Saving Cost
Suddenly, developers could not share code with all interesting target platforms anymore. Apple disallowed dynamic runtimes like the JVM on iOS, so Java was not an option. And the web was locked down, too. For example, iOS users and developers had to wait for over 15 years to get push notifications in the browser. Of course, we developers are creative folks - new possibilities for code sharing appeared. PhoneGap/Cordova and other solutions made it possible to package web pages written with HTML, CSS and JavaScript into native apps, essentially getting access to the desired native platform APIs. Decision makers at companies faced the challenge of creating products for 2 platforms: Android and iOS. Developing software is costly, so some of the managers argued: let's *save costs* by using PhoneGap, so "we'll just write it once"!

Ironically, the decision to write web apps in native packages in order to save costs often backfired. Now developers would need to support 3 tech stacks instead of 2 - driving up complexity. In addition, they would need to learn some native skills anyway in order to fight fires, and debugging bugs was hard - all driving up costs. Plus, customers would often reject these kinds of apps. And rewriting apps costs even more!

Quality
What went wrong? It appears a very important ingredient was missing. Users were much more satisfied with native apps because they excelled in *quality*: they felt snappy, were consistent with the system UI, and shined with beautiful animations - all by staying close to the system and using the native UI toolkits. The *user experience* was so much better than any wrapped web app. Frankly, I have not met a wrapped web app that I enjoyed using. Newer code sharing approaches started tapping into this potential: by using the system UI components instead of rendering web pages like React Native, or by drawing pixels that looked as close to the system UI as possible. For some time, Flutter was the only framework on the market that made this possible.

Speed
While some approaches like React Native still had some performance issues when crossing the bridge between JavaScript and native, many modern code sharing approaches focus on *speed* from the beginning. Even though it's not a primary reason for sharing code it deserves its own mention, because it improves the perceived app *quality*. But it's not just the apps that get faster: modern approaches improve *time-to-market*, and they got fast tooling, too: No matter if it is Flutter's hot reload or Compose previews: faster iteration cycles allow us to ship faster - and it's more fun to code, too!

Target More Platforms
With big players like Apple and Google limiting what can be published in their stores, we remember that there are platforms with fewer rules. Platforms where we can publish what we like, with no strings attached, where we don't need to give away 30% of all digital sales we make, and - in the case of the web - where we can update our apps pretty much instantly. Developers and decision makers are discovering that being locked into one platform and relying on the gratitude of a platform owner can be bad for them. Today, we rediscover the desktop and the web. Code sharing technologies allow us to *target more platforms*. With Kotlin Multiplatform and Compose Multiplatform, we can write code and run it on multiple platforms - be it mobile, desktop, the web or on servers.

Tapping into Existing Potential
Development rarely happens in a vacuum. We are constrained by the skills we have, the codebases that already exist, budgets and more. Sure, it's easy to pick any code sharing technology when starting a greenfield project - but even then we would like to use technologies that we are familiar with. And most of the time, we already have existing codebases, and a complete rewrite is a luxury we can not afford. What if we could *tap into existing potential*? Often, our Android apps are written in Kotlin anyway! That's a big potential to tap into. And with libraries like Room and ViewModel becoming Multiplatform-ready, that potential grows even larger as the costs for migrating get lower.
But there is more: what if we could *keep great features* that already exist? It rarely makes sense to rewrite a beautiful UI that has been carefully crafted by iOS and web developers. By having a great interop, with Kotlin Multiplatform we can pick the best tool and the most fitting developer for a job - also for the future!

Conclusion
So why do we share code? Looking back, we developers always wanted one codebase in the first place in order to *reduce complexity*. But due to the rise of new platforms we had to build apps multiple times. So we started to use code sharing to *save costs*, only to find out that we had to keep up the *quality* and *speed* as well. We share code because it allows us to *get to the market faster*. As we discover that the new platforms are limiting us, we are finding more freedom by *targeting more platforms*. And with Kotlin Multiplatform, we can *tap into the potential* of our existing Kotlin codebase and rely on our amazing colleagues to do what they do best.

Ultimately, code sharing isn't just about convenience or trend-following. It's about using our resources wisely, empowering innovation, and staying ahead in an ever-evolving tech landscape. In essence, in today's world, not embracing code sharing might just be the real cost we can't afford.

Kotlin 3D

Marc Reichelt

Senior Android Developer at Snapp Mobile

Here's a Kotlin logo for you in 3D - to do-it-yourself! The perfect companion for your real desktop. A guaranteed eye-catcher for your coworkers! If you have kids this might be a great opportunity to build this together with them - caution with scissors advised.
You can also print this on bigger paper formats, print and build multiple of these and share with your friends - you name it. Enjoy!

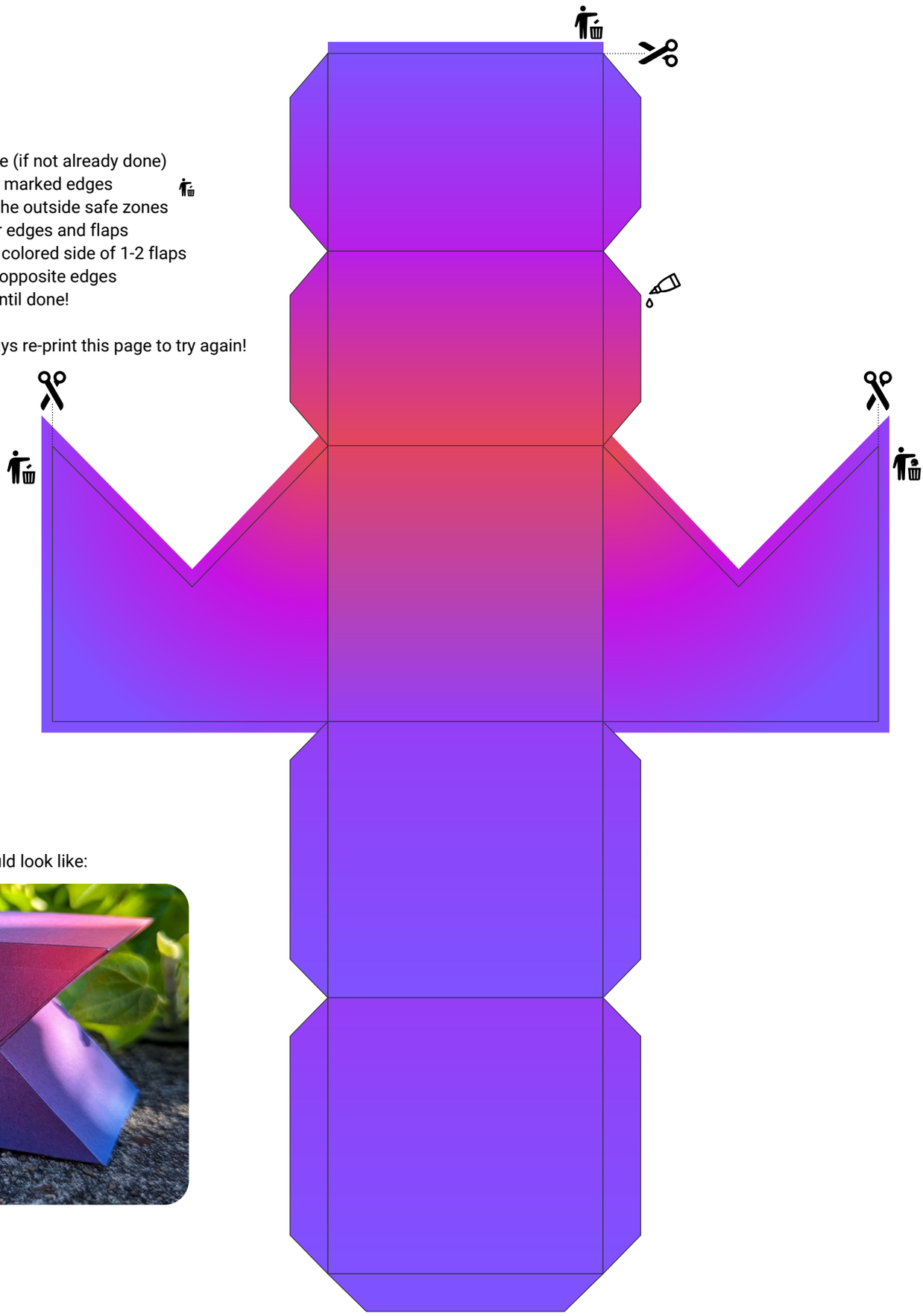
What you'll need

- A printer
- Scissors
- Glue

Instructions

1. Print this page (if not already done)
2. Cut along the marked edges
3. Throw away the outside safe zones
4. Fold the inner edges and flaps
5. Apply glue to colored side of 1-2 flaps
6. Glue flaps to opposite edges
7. Repeat 5+6 until done!

Tip: you can always re-print this page to try again!



This is how it could look like:

Kotlynne

The Future of Kotlin, with the History of English

Chet Haase

Past: tech, Current: school, Future: comedy writing

One of the first joyous experiences that everyone has when starting to use Kotlin is with the *for()* expression. Having spent years in the same old drudgery of other, similar languages, we all write something like:

```
for (i = 0; i < n; ++i)
```

And then see the syntax errors highlighted and remember that isn't quite right. We then remember there's something about ellipses and write:

```
for (i = 0 ... n)
```

but that doesn't work either. We add in more ellipses, take some away, and finally end up doing a web search to be reminded of the more eloquent, explanatory, and prose-like:

```
for (i in 0 .. n)
```

expression. Now that the syntax errors are gone, we build the code, ship the product, and discover to our joy that this Kotlin expression has enabled a beautiful new off-by-one error potential for us. So we quickly ship a new version of the product with the correct, and even more explanatory:

```
for (i in 0 until n)
```

Bugs can be unfortunate, as are apps crashing on devices in the field, angry customers, lost revenue, and potential hits to reputation, job, and career, but these risks pale in comparison to the wonder and joy of having re-learned, over and over, the beauty and eloquence of Kotlin's more natural way of expressing this logic. It's so easy and natural that we happily learn it again and again, to joyously re-train ourselves out of the rut that we've been in for all of these years prior to encountering Kotlin's beauty, since every other language we've ever used does it in that other, single, classic fashion. Thank goodness for Kotlin, shaking things up and opening our eyes to unique ways of doing mundane tasks!

It is in this wonderful tradition of the *for* loop's eloquence that I hereby propose a new Kotlin dialect which expands—nay, expounds—upon this type of expression to other situations throughout the language, in an attempt to create more opportunities for wondrous coding expressions. It does so by dipping into ye Olde Englishe language in a way that code has avoided thus far. By why should we not look to the past to inform the future? As the philosopher said surprisingly long ago, "Those who forget the past are doomed to repeat it until there's a segmentation fault."

Here is an introduction to Kotlynne. The full list of new operators and expressions is beyond the scope of this article, but I will whet your appetite with just a few:

whence

whence is a simple term which begs the question, "How did we get here?" This operator can replace more obtuse mechanisms for getting a stack track (*Thread.dumpStack()*? Please. How gauche!), resulting in a more elegant snippet such as:

```
println("Lo, upon a summer's day, didst I find myself ${whence()}")
```

beseech

beseech is not only beautiful to read and to hear (especially in dramatic code readings, which I expect to become a thing once Kotlynne is in common usage throughout the land), but it is also very flexible and functional. Kotlin has multiple different mechanisms for requesting values, or state, or even pausing until such things can be returned to the requester. All of these can be replaced by a simple, and indeed polite, call to *beseech*:

```
val foo = service.beseech(TARRY)
println("Yon value dost equal $foo")
```

for blocking calls, and:

```
val foo = service.beseech(ANON, (foo) ->
    { "Hither is thine value $foo" })
```

for unblocking calls, which will eventually call the supplied lambda… anon.

e'er

const is a useful (if inelegant) operator in Kotlin. But besides its raw harshness of language used (abbreviations? Really? Tarry we still in the dark days of Unix, limiting command length and readability to optimize storage space and typing speed?), *const* only works with primitive types. *e'er*, however, will not only work for all values; it will do so in a whimsical, poetic way:

```
e'er val foo = // any type of value whatsoe'er
```

betwixt

While the beauty of Kotlin's *for()* syntax began this journey, there is clearly more that can be done for that lonely loop. Why limit yourself to small words when larger ones can suffice? For example:

```
for (i betwixt 0 until n)
```

It means the same as the simpler *in* expression, but it's obviously far prettier and therefore better.

nought

How many bugs have you chased in your job—nay, in your entire career!—that ended up being related to 0? Zero-sized arrays that you mistakenly added items to, iterating down to 0 and then accidentally beyond, or mistaking yet again whether a given collection's first member is the 0th or first? All of this is due to 0 itself; it is an abomination in the number system. It is, indeed, the only "value" in the infinite set of numbers which is not actually a number (except for the loathsome case of imaginary numbers; do *not* get me started on that detestable mess). Zero is, by definition, nothing, where all other numbers are indeed something. So why do we treat zero like everything else, when it is clearly something else entirely?

It is time that we dealt with zero separately, to make it clear when we are dealing with it versus literally *any other number in the universe*. Kotlynne does this by changing the name of 0 to *nought*, making it clear when you are dealing with that specific value (or lack thereof), thus easily avoiding all zero-based problems of the past.

What bugs will come from the use of this new expression? None.

All's Well that Ends Swell

That's all I have time for now. Please tune in next time to learn about some of the other exciting operators and expressions, including *wherefore*, *perchance*, and of course *gallimaufry* in this new exciting dialect:

Kotlynne: A proper language.™

(continue reading on next page)

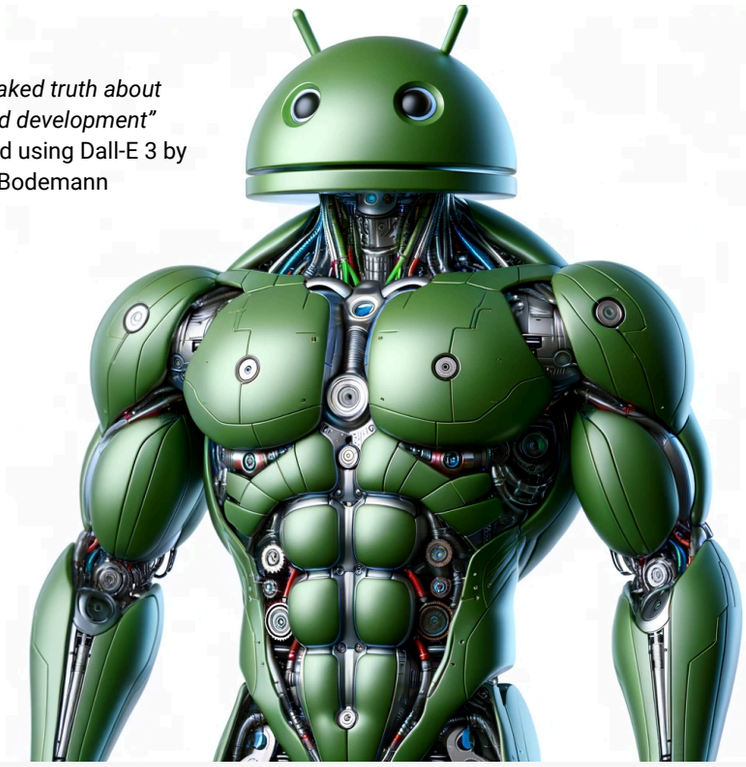
(continuation of "Kotlynne")

But of course, no treatise on proper Olde Englishe language would be complete without a sonnet.

An Ode to Code
With code that reads like books upon the shelf,
And logic that is pleasing to the eye,
I hope that you are pleased as I, myself,
And bid all other languages goodbye.

For Lo, there is more beauty in this world,
More things of wonder yearning to be seen,
More bits of flags that wait to be unfurled,
And splashed upon each massive 4k screen.

The language! That's the thing which we desire,
More crucial than the products that we ship.
'Tis joy we seek when syntax we acquire,
Until we have that language in our grip.
And then we feel life's passion, even truer
(Until there is a language that is newer).



"The naked truth about Android development"
Created using Dall-E 3 by Mario Bodemann

iOS Developers on Kotlin Multiplatform

We wanted to get an honest picture of what developers outside our Kotlin bubble think of Kotlin Multiplatform. So what's better than just asking multiple iOS developers? That's exactly what we did!

We've asked them: *"What's your take on Kotlin Multiplatform?"* Here are their responses:

"I am generally not a huge fan of cross-platform frameworks because they diminish the native experience both for users and developers. However, Kotlin Multiplatform is a great exception. It provides native performance with small overhead and as a developer you can choose how far to take it. It is great for shared business logic and models under the hood, combined with native UI."

— Martin Höller

"I was initially skeptical about using KMP, but I ended up liking using it for shared logic. Unlike other cross-platform options it allowed us to keep native performance while having a consistent behavior across platforms."

— Ahmed Bouchfaa

"KMP to me is the best cross-platform approach I have seen for a long time. It does not try to re-invent the wheel but builds on proven technology like LLVM, more importantly though: It never tries to work against the platforms, and allows the developers to concentrate on what they do best."

— Alexander von Below

"KMP might seem like a great solution on paper, but its promise falls short without a good strategy to get iOS developers to embrace and join the development. Not having iOS developers on board might lead to a slower development process. iOS devs should also be part of the planning for what KMP should include."

— José Jeria

"Appreciating cross-platform code, we faced hurdles with KMP SDK. It inflated our iOS SDK size, making distribution problematic. Runtime crashes were tough to debug. For adoption, it should compile without needing a support library. KMP's framework requirement, closed-source binary limits, and generated types like KotlinIterator and KotlinShort, which bloat the binary and complicate integration with other KMP apps, were significant concerns."

— Luke Davis

"My take on Kotlin Multiplatform is that it's a promising technology that addresses the challenges of code sharing across multiple platforms. It offers several advantages, including code reusability, improved maintainability, and potentially faster development cycles. However, like any technology, it also has its limitations and trade-offs, and its suitability depends on the specific requirements and constraints of each project. Overall, Kotlin Multiplatform can be a valuable tool for developers looking to build cross-platform applications efficiently."

— Grace Ohanian

Fun with Kotlin

By Microsoft Copilot **AI**, reviewed, improved, and organized by **Human!** 🎉

Mohsen Mirhoseini
Android Team/Tech Lead at Firework
YouTube.com/@androiddevelopertips
X.com/@mohsenoid

Disclaimer: This article is purely for laughs and giggles. Kotlin is an amazing language, and I love it (seriously). Also, don't do anything suggested in this article at work! 🎩🤖

Kotlin: The Language origin

Did you know Kotlin was named after a secret island near St. Petersburg? It's true! Just like Java was named after coffee (or so the legend goes), Kotlin draws its inspiration from exotic locales.

Kotlin Island Fun Facts

- Population: 42 programmers and a talking parrot.
- Official Language: Kotlin, obviously.
- Currency: Null coins (because null is priceless).
- National Sport: Swimming in cold flows and hot channels.

Top 10 Kotlin Pranks for Developers

1. **The Null Pointer Parade**: Replace all null values with random emojis. 🤪
2. **Extension Function Extravaganza**: Extend Int to have a dance() function. Usage: *42.dance()*.
3. **Operator Overload Overload**: Override the + operator to concatenate strings with cat GIFs.
4. **Infinite Loop Labyrinth**: Introduce a hidden loop that leads to a mysterious dimension.
5. **Typealias Tumult**: Rename all types to "*Potato*" using typealias. 🥔
6. **Coroutine Carousel**: Randomly suspend and resume functions during runtime.
7. **The Great when Swap**: Swap all when branches for maximum confusion.
8. **Classy Chaos**: Replace all class names with famous movie quotes.
9. **The Elusive it**: Rename all lambda parameters to it and watch the chaos unfold.
10. **The "Hello, World!" Remix**: Print "*Goodbye, Universe!*" instead. 🌌

KotlinConf 2024 Highlights

At KotlinConf 2024, JetBrains dropped some bombshells:

- **Kotlin 3.0.0**: Now with built-in teleportation (beta feature).
- **Google's Love Affair with Kotlin**: Android devs, rejoice! Kotlin is now Google's favorite language. Java, who?
- **Extra coffee breaks**: Kotlin can reduce the lines of code by up to 40% compared to Java!

Kotlin vs. Java: The Epic Rap Battle 🎤

Kotlin (spitting bars):

I'm Kotlin, the new kid on the block,
Null safety, coroutines, I rock!
Java, you're old, verbose, and bland,
I'll outshine you like a supernova grand!

Java (mic drop):

I've been here since '95, don't you forget,
Your syntax might be cute, but mine's legit.
Kotlin, you're cool, but I paved the way,
So step aside, kid, and let the OG play!

Koonlander

Mario Bodemann
DevRel @ Deutsche Telekom

The operating system of your space ship has just crashed! As the result of using outdated languages something called a ‘use after free’ exception occurred and wiped out the complete software system! You need to rebuild the operating system of the **KSS Pegasus** (Kotlin Software Ship).

Luckily you found the following old drawings of the source code, so you should be able to reconstruct the system, and finally land the lander safely!

During your exploration of the code drawings, you'll also need to look at the hints to see how they apply to your flying saucer.

HINT 1: Code Editor
Since the software system you are rebuilding will use some graphical powers of ANSI terminals, you need to run the result in an ANSI compatible terminal. Sadly IntelliJ does not interpret the needed bits, yet. Recommendation is to either write it there and execute it on the command line, or write it completely in the command line using tools like vim, emacs, nano, pico.

```
fun interact(
  move: Int,
  height: Int,
  velocity: Int,
  fuel: Int,
  update: (height: Int, state: State) -> Unit
): State {
  print(move(23, 7) + "You have ${color(f = FG.GREEN, s = "${fuel}l")} fuel left.")
  print(move(23, 8) + "The ship is ${color(f = FG.YELLOW, s = "${height}m")} above landing and")
  print(move(23, 9) + "it accelerates with a speed of ${color(f = FG.MAGENTA, s = "${velocity}m²")}.")

  var input: Int?
  do {
    print(
      move(23, 12) +
        color(FG.BRIGHT_YELLOW, BG.BLUE, "How long do you want to fire the engines?") +
        " "
    )

    input = readln().toIntOrNull()
    if (input == null) {
      print(
        move(23, 13) +
          color(f = FG.BRIGHT_RED, s = "Not a valid number. Try again.")
      )
    }
  } while (input == null)

  val duration = when {
    input < 0 -> 0
    input > 30 -> 30
    input > fuel -> fuel
    else -> input
  }

  val v = velocity - duration + 5
  val f = fuel - duration
  val h = height - (v + velocity) / 2

  return if ((v + velocity) / 2 >= height) {
    val v1 = velocity + (5 - duration) * height / velocity
    when {
      v1 > 5 -> CRASHED(move + 1)
      v > 1 -> LANDED_WITH_CASUALTIES(move + 1)
      else -> LANDED_SAFELY(move + 1)
    }
  } else {
    val state = when (duration) {
      in 3..15 -> BREAKING_MIN
      in 15..25 -> BREAKING
      in 25..30 -> BREAKING_MAX
      else -> LANDING
    }

    update(h, state)
    interact(move + 1, h, v, f, update)
  }
}
```

```
@file:JvmName("Koonlander")

import State.*
import kotlin.math.roundToInt

const val ESC = "\u001b"
const val CLS = "$ESC[2J"
const val BLINK = "$ESC[5m"

enum class FG(val c: Int) {
  RED(31),
  GREEN(32),
  YELLOW(33),
  BLUE(34),
  MAGENTA(35),
  CYAN(36),
  WHITE(37),
  BRIGHT_RED(91),
  BRIGHT_GREEN(92),
  BRIGHT_YELLOW(93),
  BRIGHT_BLUE(94),
  BRIGHT_MAGENTA(95),
  BRIGHT_CYAN(96),
  BRIGHT_WHITE(97),
}
```

```
enum class BG(val c: Int) {
  RED(41),
  GREEN(42),
  YELLOW(43),
  BLUE(44),
  MAGENTA(45),
  CYAN(46),
  WHITE(47),
  BRIGHT_RED(101),
  BRIGHT_GREEN(102),
  BRIGHT_YELLOW(103),
  BRIGHT_BLUE(104),
  BRIGHT_MAGENTA(105),
  BRIGHT_CYAN(106),
  BRIGHT_WHITE(107),
}

sealed interface State {
  data object LANDING : State
  data object BREAKING_MIN : State
  data object BREAKING : State
  data object BREAKING_MAX : State
  data class CRASHED(val moves: Int) : State
  data class LANDED_WITH_CASUALTIES(val moves: Int) : State
  data class LANDED_SAFELY(val moves: Int) : State
}
```

```
fun State.toShip(x: Int, y: Int): String = move(x, y) + when (this) {
  is LANDING -> 128760.utf8
  is BREAKING_MIN -> 128760.utf8 + move(x, y + 1) + "::"
  is BREAKING -> 128760.utf8 + move(x, y + 1) + "^^"
  is BREAKING_MAX -> 128760.utf8 + move(x - 1, y + 1) + color(f = FG.RED, s = "/|\\")
  is CRASHED -> 128165.utf8
  is LANDED_WITH_CASUALTIES -> 128760.utf8 + 128173.utf8
  is LANDED_SAFELY ->
    move(x - 1, y) + 128170.utf8 + 128760.utf8 + 128077.utf8
}
```

```
val Int.utf8: String
  get() = Character.toString(this)

fun color(f: FG? = null, b: BG? = null, s: String) =
  "$ESC[${f?.c ?: ""}${if (b != null) ";${b.c}" else ""}m$s$ESC[m"

fun move(x: Int, y: Int) = "$ESC[$y;${x}H"

data class Star(
  val x: Int, val y: Int,
  val c: FG,
  val s: String
)
```

```
fun rect(
  x: Int, y: Int,
  w: Int, h: Int,
  border: String, fill: String = ""
) {
  if (fill.isNotEmpty()) {
    for (j in 1 until h - 1) {
      for (i in 1 until w - 1) {
        print(move(x + i, y + j) + fill)
      }
    }
  }

  for (i in 0 until w) {
    print(move(x + i, y + 0) + border)
    print(move(x + i, y + h - 1) + border)
  }

  for (j in 0 until h) {
    print(move(x + 0, y + j) + border)
    print(move(x + w - 1, y + j) + border)
  }
}
```

```
fun display(stars: List<Star>, height: Int, state: State) {
  print(CLS)

  stars.forEach { s ->
    print(move(s.x, s.y) + color(f = s.c, s = s.s))
  }

  rect(
    1, 14, 20, 3,
    border = color(b = BG.BRIGHT_WHITE, s = " "),
    fill = color(b = BG.WHITE, s = " ")
  )
  rect(
    1, 1, 20, 16,
    border = "+"
  )

  val y = (height / 50.0).roundToInt()
  print(color(s = state.toShip(10, 13 - y)))

  print(move(23, 1) + color(f = FG.RED, s = "Koonlander: A Space Landing Odyssey"))

  print(move(23, 3) + "Your engines just failed. Break your fall by accelerating")
  print(move(23, 4) + "your ship to not crash. Keep an eye on your fuel and")
  print(move(23, 5) + "your speed. The ground is closer then you might think.")
}
```

```
fun main() {
  val stars = (0..24).map {
    Star(
      (2..18).random(),
      (2..12).random(),
      FG.entries.random(),
      listOf(10024,/*11088,*/8902).random().utf8,
    )
  }

  display(stars = stars, height = 500, LANDING)

  val state = interact(
    move = 0, height = 500,
    velocity = 50, fuel = 1200
  ) { height, state ->
    display(stars, height, state)
  }

  display(stars, 0, state)

  var moves = 0
  when (state) {
    is CRASHED -> {
      rect(
        23, 7, 19, 3,
        color(f = FG.RED, b = BG.BRIGHT_RED, s = "$BLINK!")
      )
      print(
        move(24, 8) +
          BLINK +
          color(f = FG.RED, b = BG.BRIGHT_RED, s = " ! You crashed ! ")
      )
      moves = state.moves
    }

    is LANDED_WITH_CASUALTIES -> {
      rect(
        23, 7, 19, 3,
        color(f = FG.YELLOW, b = BG.BRIGHT_YELLOW, s = "+")
      )
      print(
        move(24, 8) +
          BLINK +
          color(f = FG.RED, b = BG.BRIGHT_YELLOW, s = "  Injured but ok ")
      )
      moves = state.moves
    }

    is LANDED_SAFELY -> {
      rect(
        23, 7, 19, 3,
        color(f = FG.GREEN, b = BG.BRIGHT_GREEN, s = "~")
      )
      print(
        move(24, 8) +
          BLINK +
          color(f = FG.GREEN, b = BG.BRIGHT_GREEN, s = "      LANDED     ")
      )
      moves = state.moves
    }

    else -> {}
  }

  println(
    move(23, 16) + if (state is CRASHED)
      color(f = FG.BRIGHT_RED, s = "Better luck next time!")
    else
      color(
        f = FG.YELLOW,
        s = "$BLINK You used $moves steps. Can you use less? "
      ) + 127808.utf8
  )
}
```

HINT 2: Installing Kotlin

Best to also reinstall the *kotlin* compiler. On Mac systems, you'd need to install it by running **brew install kotlin**, after brew is installed. On Android, you'd need to install and fire up **termux** and install it using **pkg install kotlin**. On Linux, you'd use your favorite package manager / snap / flatpak and issue this line in a terminal: **sudo dnf install kotlin**, replacing dnf with your manager of choice. On Windows, you download a ZIP from github.com/JetBrains/kotlin/releases.

HINT 3: Building and Running

Following Hint 2, you can simply open a terminal to where you saved your recovered source code and call the Kotlin compiler named *kotlinc* with the kotlin file you created. I.e. **kotlinc Koonlander.kt**. If you typed everything correctly, there should be no output from the compiler and you can continue with the running part. If the compiler complained about something, take a look at your source code and the source code presented here, maybe something is different?

Running is similarly simple: Assuming you are still in the same folder, you type **kotlin Koonlander** and let the OS run for the first time. Can you save the space ship from crashing?

Disclaimer

Originally written by Daniel Isaaman and Jenny Tyler in Computer Spacegames-magazine from Usborne Publishing Ltd. in 1982, downloaded from usborns public archive:
usborne.com/gb/books/computer-and-coding-books
Adapted to Kotlin and slight story changes by Mario Bodemann ❤️

Background image thanks to Ryan Hutton via Unsplash

Convert HTML to PDF with Kotlin/JVM

Mariusz Sołtysiak

Staff Engineer at SumUp

Recently, my team encountered the challenge of converting HTML files to PDF without resorting to hacks or outdated CSS versions. Our goal was to find a high-performance, open-source solution that would not keep customers waiting for their generated PDFs. After evaluating several options, we found that none of the popular projects met our requirements.

After two rounds of searching for possible solutions, we found that Gotenberg, Flying Saucer, and PDFBox did not fully satisfy our needs.

However, we discovered Playwright, which, although not primarily designed for generating PDFs, provides this functionality. Playwright was originally designed for use with Node.js, but there is also a Java version called Playwright for Java, which works well with Kotlin.

What about scalability?

After conducting tests, we found that Playwright is highly performant. However, we also required a scalable solution that could handle up to 100 conversions per second. Generating PDFs is a CPU and memory-intensive process, so running 100 instances continuously, even during low-traffic hours, was not feasible.

To address this, I proposed creating a pool of Playwright instances, similar to a pool of database connections. We decided to use Apache Commons Pool for that purpose, which was the right choice.

Time to code!

Now that we have selected our tools, we can begin writing code. For this article, I will be using Spring Boot 3.2.x, but the example should work with any other framework. Project configuration details will not be covered here, but Gradle imports and tasks can be found in the example repository.

The pool works like the following:

1. A client requests an object from the pool using the borrowObject() method.
2. The pool checks if there are any idle objects available.
3. If there are idle objects available, the pool returns an idle object to the client.
4. If there are no idle objects available, the pool creates a new object using the factory object and returns it to the client.
5. The client uses the object.
6. The client returns the object to the pool using the returnObject(T obj) method.
7. The pool checks if the object is still valid using the evictionPolicy object.
8. If the object is valid, the pool adds the object to the set of idle objects.
9. If the object is not valid, the pool destroys the object using the factory object and removes it from the pool.
10. The client can request another object from the pool using the borrowObject() method.
11. The client can close the pool using the close() method.
12. The pool releases all resources associated with it.

```
class BrowserContextPooledObjectFactory :
  PooledObjectFactory<BrowserContext>, AutoCloseable {
  // pool storage
  private val playwrightMap =
    ConcurrentHashMap<BrowserContext, Playwright>()

  // executed after "borrowObject"
  //  method call on the pool
  override fun activateObject(p: PooledObject<BrowserContext>) {
    p.getObject()?.clearCookies()
  }

  // executed when the pool evicts idle object
  override fun destroyObject(p: PooledObject<BrowserContext?>) {
    p.getObject()?.run {
      playwrightMap.remove(this)?.close()
    }
  }

  private fun cleanupBrowserContext(context: BrowserContext) {
    context.clearCookies()
    context.clearPermissions()
    val pages = context.pages()
    if (pages.isNotEmpty()) {
      for (page in pages) {
        if (page.isClosed) {
          continue
        }
      }
    }
  }

  // creates a new instance of the object,
  //  which is added to the pool
  override fun makeObject(): PooledObject<BrowserContext> {
    val playwright = Playwright.create()
    val browserContext =
        playwright.chromium().launch().newContext()
    playwrightMap[browserContext] = playwright
    return DefaultPooledObject(browserContext)
  }

  // called after "returnObject" method call on the pool
  override fun passivateObject(p: PooledObject<BrowserContext>) {
    p.getObject()?.run {
      clearCookies()
      pages().forEach { page ->
        page?.takeIf { !it.isClosed }?.close()
      }
    }
  }

  // checks if the object can be safely borrowed
  override fun validateObject(p: PooledObject<BrowserContext>) =
    p.getObject() != null

  // executed to close the whole pool of objects,
  //  usually during the app shutdown
  override fun close() =
    playwrightMap.forEach { (browserContext, playwright) ->
      cleanupBrowserContext(browserContext)
      playwright.close()
    }
}
```

(continue reading on next page)

(continuation of "Convert HTML to PDF with Kotlin/JVM")

The above class is the foundation of our solution. This straightforward implementation is sufficient for use with the pool configuration. Let's now create a bean with the pool:

```
@Configuration
@ConditionalOnClass(Playwright::class,
PooledObjectFactory::class)
class PlaywrightConfig{
  @Bean
  fun browserContextPool() =
    BrowserContextPool(
      BrowserContextPooledObjectFactory(),
      GenericObjectPoolConfig<BrowserContext>().apply {
        jmxEnabled = false
        minIdle = 5
        maxIdle = 10
        maxTotal = 15
        softMinEvictableIdleDuration = Duration.ofMinutes(3)
        timeBetweenEvictionRuns = Duration.ofSeconds(30)
      },
    ).also {
      it.addObjects(it.minIdle)
    }

  class BrowserContextPool(
    factory: BrowserContextPooledObjectFactory,
    config: GenericObjectPoolConfig<BrowserContext>,
  ) : GenericObjectPool<BrowserContext>(factory, config)
}
```

This configuration defines a pool of 5 objects that will be created during startup. These 5 objects will be the minimum amount of instances available to borrow. If necessary, the amount can be increased up to 15 instances, but only a maximum of 10 instances can be in the idle state. Any idle instances will be removed after 3 minutes of not being borrowed again.

Play(wright) time!

To test the solution, use this simple example:

```
@RestController
class TestController(
  private val browserContextPool: BrowserContextPool,
) {
  @GetMapping("/test")
  suspend fun test(): ResponseEntity<Resource> = ResponseEntity
    .ok()
    .contentType(MediaType.APPLICATION_PDF)
    .header("Content-Disposition", "attachment; filename=test.pdf")
    .body(InputStreamResource(getPdf()))

  private suspend fun getPdf(): InputStream {
    val browserContext = browserContextPool.borrowObject()
    val page = browserContext.newPage()
    page.setContent(
      "<html><body><h1>Hello, World!</h1></body></html>"
    )
    return ByteArrayInputStream(page.pdf())
      .also { browserContextPool.returnObject(browserContext) }
  }
}
```

The function takes a BrowserContext instance, adds HTML code to the first page of the browser window, prints it to PDF, and returns a ByteArray. This ByteArray should be converted to a ByteArrayInputStream and returned from the controller method with proper headers.

What's next?

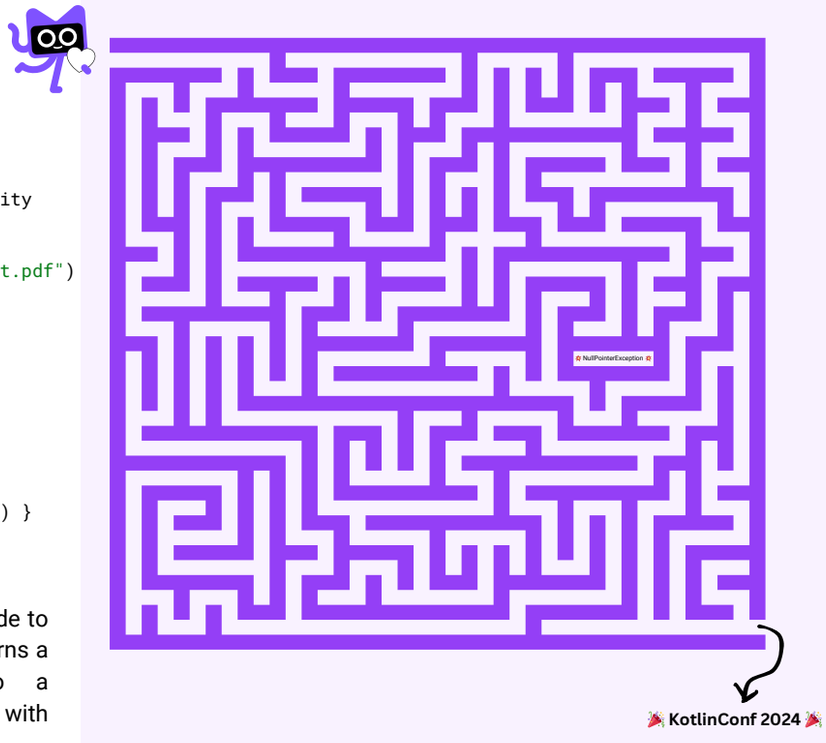
The provided implementation and configuration are sufficient for services that do not require generating a high volume of PDFs. However, for optimal concurrent performance, both horizontal and vertical scaling may be necessary. During testing on an M1 Pro processor, 20 browser instances were found to effectively process 20 concurrent requests in 550-650ms, which is a favorable result. My team assumes the same number of instances per Kubernetes pod and autoscales based on CPU usage.

Useful links:

- The example implementation you can find on my GitHub: github.com/marrek13/kotlin-html-to-pdf-demo
- Playwright For Java: playwright.dev/java/docs/intro
- Apache Commons Pool: commons.apache.org/proper/commons-pool

Kodee at KotlinConf

Oh no! The KotlinConf keynote is about to start, and Kodee still needs to find a way through the streets of Copenhagen to get to the conference entrance! Can you help Kodee find a way and bring some love to the KotlinConf team?



Declarative Gradle? Amper? What's going on?!

Siggi Gunnarsson

Mobile Developer @ Bosch eBike

We Kotlin developers have it pretty easy as it comes to configuring and executing our builds. We usually work with nice IDEs that help setting projects up for us, and a robust build system in Gradle. But there are some issues with Gradle. New projects are attempting to remedy these issues by changing how we work.

Gradle gives us a nice way to import dependencies, compile and package the project. It even helps with running tests. Not all languages have such a robust and "standardized" system. In Python, dependencies are declared by writing them into a *requirements.txt* file. Then you use *pip* to install everything into your (hopefully virtual) Python environment and pray that there are no dependency conflicts. Since there will be conflicts anyway, you can write explicit library versions into this *.txt* file based on the complaints from *pip* for a while before deciding maybe AI development isn't for you.

Back to Kotlin! For a simple project it's pretty easy to read the Groovy based *build.gradle* build files. Since you can now write *build.gradle.kts* in Kotlin, build files are also easy to write!

Kotlin application `build.gradle.kts`

```
plugins {
    application
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(libs.someDependency)
}

tasks.register("funTask") {
    moveFiles()
}

tasks.getByName("build").dependsOn("funTask")

application {
    mainClass.set("MainClassFile")
}
```

This all looks straightforward. The project type is application. We declare some dependencies, and where they should be fetched from.

It's nice to write build configurations in our favorite programming language, but that comes at a cost. Gradle is not able to statically analyze our build files when they contain arbitrary build logic. This limits how much our IDEs can help us, and can lead to hard-to-debug build issues.

We can mitigate this by following best practices:
- Only put declarative software definitions into build scripts
- Put all your custom build logic into Gradle plugins

Unfortunately, there's nothing ensuring we follow best practices. Many projects pepper gnarly build logic around their build scripts. It's also not always obvious if your "declarative" build code really is just that. Was your new Gradle code "software definition" or did you actually write "build logic"? Did you spot the build logic cleverly hidden in the example?

Enter: Declarative Gradle

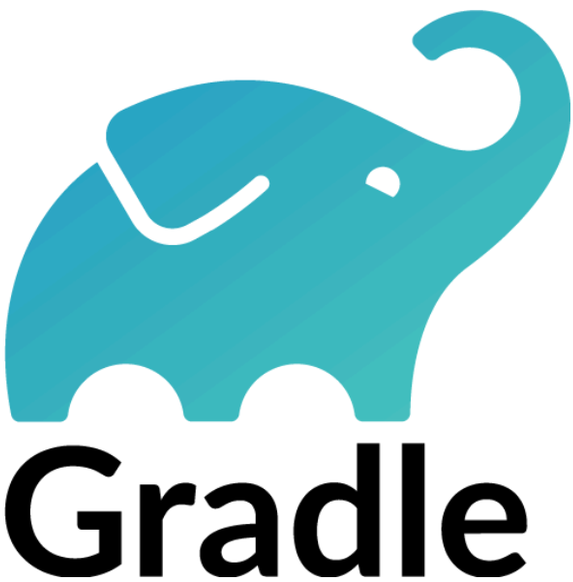
The good folks at Gradle are aware of this, and have started work on a new kind of Gradle script file that only supports build configuration functionality. This should ensure that build logic is safely tucked away in build plugins. Although this work is still in a prototyping phase, let's look at an example of a simple Kotlin Multiplatform application. Note the ".dcl" file extension.

Kotlin application `build.gradle.dcl`

```
kotlinApplication {
  dependencies {
    implementation(libs.someDependency)
  }

  targets {
    jvm {
      jdkVersion = 17
      mainClass = "com.example.AppKt"
      dependencies {
        implementation(libs.someJvmDependency)
      }
    }
    macOsArm64 {
      entryPoint = "com.example.main"
    }
  }
}
```

(continue reading on next page)



Gradle logo from gradle.com/brand

(continuation of "Declarative Gradle")

The Declarative Gradle example looks familiar, but arbitrary code is not supported. Gradle could therefore statically parse this file. This means that the IDE would be able to improve things like *auto-complete*, *refactoring*, and *documentation quick access*. Imagine the improvements we got when moving from Groovy build files to Kotlin, but even better!

It would even be possible to edit entire build files with a graphical interface, like with android's xml resource files.
Let's hope the Android Studio team isn't reading this...

Amped for Amper
Can we go even further? In the introduction, we made fun of Python's *requirements.txt* system. But what if our project setup was actually simple? Can we skip the confusing /src/jsMain/kotlin/com/example/my/code folder structure and have the project setup in a text file?

The Cargo build system for Rust is very approachable and simple to learn. Setting up projects is quick so you can go directly to writing code and arguing with the Rust borrow checker.

Can we do this in Kotlin? That's the question that the Amper project from JetBrains is trying to answer.

Currently, the Amper project is seen as a build configuration tool only, and the current implementation is built upon Gradle.

The syntax is new, but familiar. Current prototypes use *yaml*, but no decision has been made on filetype. Apologies if you are reading this in *KotlinConf 2024* and just watched the Amper team announce something about *requirements.txt*

Kotlin application `module.yaml`

```yaml
product: jvm/app

dependencies:
 - org.jetbrains.kotlinx:kotlinx-datetime:0.6.0

test-dependencies:
 - org.jetbrains.kotlin:kotlin-test:1.9.0

settings:
 kotlin:
   languageVersion: 1.8
 java:
   source: 17
 jvm:
   target: 17
```

Sure looks like *yaml*. Also looks nice and simple. Folder structure is different from a typical gradle project as well. Source files live in the */src folder*, or optionally in e.g. */src@android* for multiplatform modules. In theory, project and build setup could be drastically simplified by moving to Amper.

Get to the point
Declarative Gradle promises to improve tooling and speed up builds. It could provide a path to migrating existing projects to a simple but powerful version of Gradle that avoids the footguns of today. "Simple but powerful". Let's see what the Gradle developers can deliver!

Playing with *Amper* feels fun and refreshing, and should make it easier to start new projects, especially for new developers. At the same time, Amper is a big departure from the current system and it's not clear if us developers will ever migrate our projects to a new system.

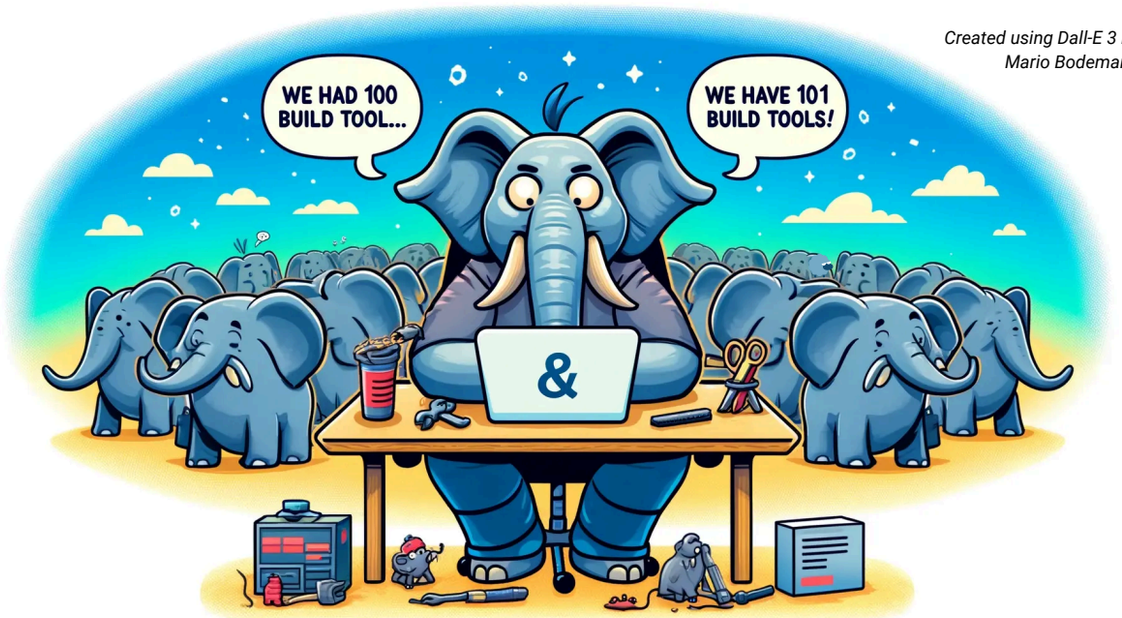
The future of Kotlin build systems is pretty exciting! Just remember that these projects are prototypes, so maybe don't rewrite your 200 module enterprise project to use either.

Further reading
blog.jetbrains.com/blog/2023/11/09/amper-improving-the-build-tooling-user-experience/

github.com/JetBrains/amper

github.com/gradle/declarative-gradle/

doc.rust-lang.org/rust-by-example/cargo.html



Created using Dall-E 3 by Mario Bodemann

What kind of Kotliner are you?

Martin Bonnin
Chocolate Droid Officer

Are you new to Kotlin? Or were you using Kotlin traits? Or maybe somewhere in the middle? Curious about what brought you to Kotlin and what made you stay?

Wait no more! Answer these questions and find out what kind of Kotliner you really are!
(turn the page upside down for answers)

1. What is your favorite Kotlin feature?

a. Functions types
b. IDE integration
c. Multiplatform support
d. Null safety

2. Your IDE of choice is:

a. Fleet
b. vi
c. IntelliJ/Android Studio
d. Visual Studio Code

3. Oh no! There's a bug in production. What's your immediate reaction?

a. Quick team meeting! We need everyone's input to solve this efficiently.
b. How is that even possible? We have 100% test coverage...
c. Quick revert! Good thing I can mitigate that server side.
d. Dig in the crash reports, better find the root cause as soon as possible.

4. THE feature of Kotlin you are expecting the most:

a. Faster builds
b. Not sure, surprise me!
c. Context receivers
d. Denotable union types

5. If Kotlin were an animal, what do you think it would it would be?

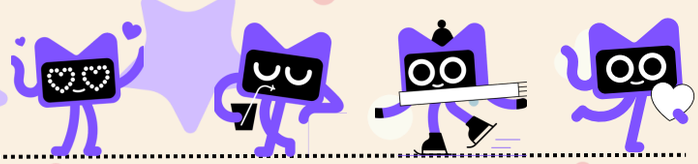
a. A dolphin
b. An elephant
c. A monkey
d. An eagle

6. A new version of Kotlin is just out. What do you do?

a. Start a pet project to play with the newest features.
b. Wait a bit for the next minor version. There's always a next minor version.
c. Add it to next sprint.
d. Merge the PR. You've been using the alphas and rcs for the last 2 months already.

7.To you, Kotlin is the name of:

a. an island
b. a programming language
c. an animal
d. a polish condiment



Count your numbers of '●', '▲', '■' and '◆': 1 a: ●, b: ▲, c: ■, d: ◆ | 2 a: ■, b: ●, c: ▲, d: ◆ | 3 a: ◆, b: ●, c: ■, d: ▲ | 4 a: ■, b: ◆, c: ▲, d: ● | 5 a: ●, b: ▲, c: ■, d: ◆ | 6 a: ●, b: ◆, c: ▲, d: ■ | 7 a: ●, b: ▲, c: ◆, d: ■

You have a majority of '■': You are a ***multiplatform enthusiast*.**

Write once, run everywhere! This has been the Holy Grail of the software industry for the last 50 years and with Kotlin you have finally found it! You deploy to multiple app stores from CI and are at peak productivity. You never lose an opportunity to showcase the latest Feet features to your iOS colleagues and discuss the merits of Kotlin on the backend. You're looking forward to Swift interop and wasm to integrate even better in those ecosystems. The future is bright!

You have a majority of '●': You are a ***functional purist*.**

A language is only as good as its mathematical foundations! You avoid mutable state like the plague and like your functions pure. You love that Kotlin comes with a non-turing complete type system and high order functions. You're dreaming of a world without bugs where programs can be proven at build time. And who knows, with Kotlin, this might happen sooner than expected!

You have a majority of '▲': You are a ***pragmatic crafter*.**

A good workman looks after his tools! Kotlin is simply the best tool for your job. You love the top-notch IDE integration, the rich ecosystem, the expressive language and the always resourceful community. You know all the IntelliJ/Andorid Studio shortcuts by heart and when it comes to debugging, you're not afraid of single stepping and using conditional breakpoints. The only thing slowing you down was compilation times but you already have a K2 branch ready to merge. Really there's not much to improve... Except maybe those AI bots that could automate things even more ?...

You have a majority of '◆': You are a ***curious polyglot*.**

Kotlin is not your main language. Maybe you've been doing Java for a while and heard about this null safety thing in Kotlin. Or maybe you like to control your memory allocations and stay closer to your hardware, save every CPU cycle! In all cases, you recognize that different jobs need different tools and are eager to learn about different technologies and what they can bring. You like exploring things and venturing in new spaces. Maybe one day, you'll settle in Kotlin land...

Embrace the Crisis

From an engineer's diary

Matthias Geisler

Developer @ Thermondo

Dear Diary,

Let us think about crises. Perhaps we define it first - the words' origin comes from ancient Greek - κρίσις (krisis)/ κρίνω (krino). κρίνω - in short it means something like to separate/divide/distinguish but includes to judge/contest/dispute as well. Interestingly, the crisis got a very negative coating over time. When we speak today about a crisis, we mean something like a difficult or dangerous point in a situation/time. What is also implied is that this turning point has one or more resolutions, which might end in a happy end or utter disaster.

So why should I care, as a software engineer, about crises? Is that not just some sort of shenanigans I get into here? Well, I think that crises and more importantly crisis handling are part of our work in software engineering, even if we will not acknowledge it. But let us scope the term first. We can chunk crises in 3 layers: *Organization* or suborganizations (like a company or sub-company) we are working in; *unit* or subunit (like a department or team); and the individual *contributors*. And to state the obvious - of course, all 3 layers are related to each other - be it directly or indirectly. Important here - if a crisis occurs in one of those layers, they shine into the others, even though their impact might be different. For example, if I experience a crisis as an engineer, the impact on my company will most likely be negligible. If my company is in a crisis, it might heavily impact my life and my psychological safety. Or let's say I am the CTO, and in a crisis, the impact on the organization is surely different by magnitudes.

Okay…enough definitions, but let me drop a thing before I come to the meat (I am vegetarian, I swear) of this text - Sociotechnical systems (STS). When I am thinking about tech and crises, it is a thing I should mention.

Well first and foremost, you need to realize that a crisis is ongoing, easy right? A surprise for me was that people sometimes are so used to acting in crisis mode that they forget how different it can be. Secondly, a crisis needs analysis. You need to know what you are dealing with. If you are not able to locate the root cause of it and take appropriate actions, the outcome is likely not in our favour. Moreover, a crisis will expose weaknesses, shortcomings, and inefficiencies in your Organization, Unit, or of yourself. This of course sounds scary, but it can work for you. I remember a point where an entire team was at stake. It had to achieve certain goals in a certain time and the team might have survived. However, while some invested in that endeavour, others just pursued their personal goals and agendas. This created friction since each tiny action needed to be discussed and prohibited fast-paced. From the outside, it might have looked like the team was not able to solve their issues or only individuals were skilled enough. That for sure raised the question - why is the rest of the team even needed? Ultimately, the team failed to succeed and it was laid off in the end. What I can learn here, is: if a team does not act as a unit in a time of crisis, it will lead to failure. This small example contains 2 things:
a) we have criteria that are measurable (to an extent), and
b) we can lay out some characteristics that are worth aiming for to increase team performance - key traits.

For measurable criteria *(a)* we care about most here is - how a unit or individual contributes to the overall success of its surroundings. Please, don't confuse it with a threshold. It is just a measurable criterion that can be monitored, nothing else. It helps to keep track of a situation and enables further analysis.

The key traits *(b)* are much more interesting. Given you don't deal with reluctant people, teams, or organizations (this is a topic for another day), a good crisis will let those characteristics shine. What I am talking about?

- a *shared understanding of work*. This includes what needs to be done, what has which priority, and which resources need to be allocated to achieve the work. I think this sounds easier than it is achievable. The problem space needs to be understood equally to each member of the team. It also requires an awareness of the skill sets and capability in the team. While there is no easy answer to how to get there, I think, a key component is listening, understanding, and compromising with your peers, especially in times of great pressure. But you also need to know where to push for it. A good tool might here pair/assemble programming or a shared activity like coding katas to create a shared understanding.

- a *shared understanding on the way to approach work*. No matter what you use Scrum, Kanban, or pair programming; everybody should understand why this method and which purpose it serves. You might want to add or change or remove rituals/ways of doing things. But also be open to try new things - like pair programming. It might need some time to get the hang of it, but once you master it (which might need longer than a sprint or two) to see the benefits. Also save things, especially retrospectives, which act as a corrective.

- everybody in the team is visible/*identifies with the team*. I cannot overstate how important this is. Each member of the team; be it an intern or team-lead; be it backend, or frontend, or designer; should have their five minutes, their card on the board, etc. The reason is simple: you need to drive work as a team not as single contributors. This prevents situations where you start to work against each other. The goal should be, here, that you want to say: "We do" instead of "I do". You are branded by your team, so to speak. You commit to being a part of it.

- *autonomous regulation*. The team makes its own decisions and takes responsibility for that. In other words, it owns their actions. This implies that the team can react and adapt quickly to new situations. Even more, this ensures that everybody participates in critical decisions and has the opportunity to raise their opinions/concerns. This prevents scenarios where people go rough and do their own agenda or the opposite where people execute what they are told to.

- a *safe space*. This is, in my opinion, the most crucial point. You can express that I think in one important word: Trust! While a safe space of course does not generate trust on its own, it facilitates it. And trust is what you need if you want to accelerate. Trust is like the sworn enemy of micromanaging, if you can trust your peers, you can trust their work. However, if trust is lost in/into a team, it is a crisis on its own, which will lead to failure sooner or later. Even though I love failure, this is something you can avoid.

(continue reading on next page)

(continuation of “Embrace the Crisis”)

Well, I am mumbling. A crisis will not automatically drive everybody to this, but due to its character of urgency, it creates an environment where each of those points gets much more prominent. Furthermore, it might give degrees of freedom that had been previously unimaginable, like rethinking the current tech stack or introducing a different methodology (I love you TDD) and so on since everything is in question anyway. The flip side is that it might also show limitations that are not achievable in the Organization or the Unit. In any case, a good crisis has the power to change things - don’t waste it, embrace it!

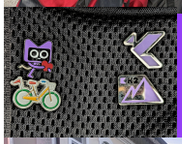
That’s it for today, write to you soon!

Martin & Louis cycle to KotlinConf

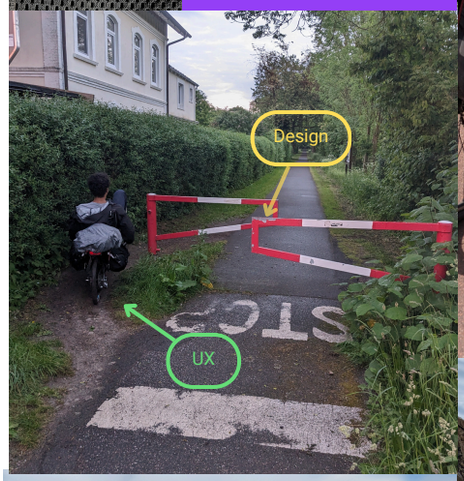
Last year (2023), Martin Bonnin and Louis CAD cycled over 300km from Lille, France, to KotlinConf Amsterdam…

This year they’re doing it again! There is more than 350km from Hamburg, Germany, to KotlinConf Copenhagen.

Martin and Louis after Paris - Köln/Cologne, before they knew their 2nd train would be late 😜



Kodee is coming too! (taken before Kodee got lost in the streets of Copenhagen, see page 13)

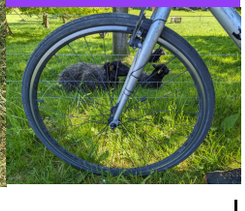


Nicole Terc’s sheep can be seen in this picture! Can you find all of them hidden in this magazine?

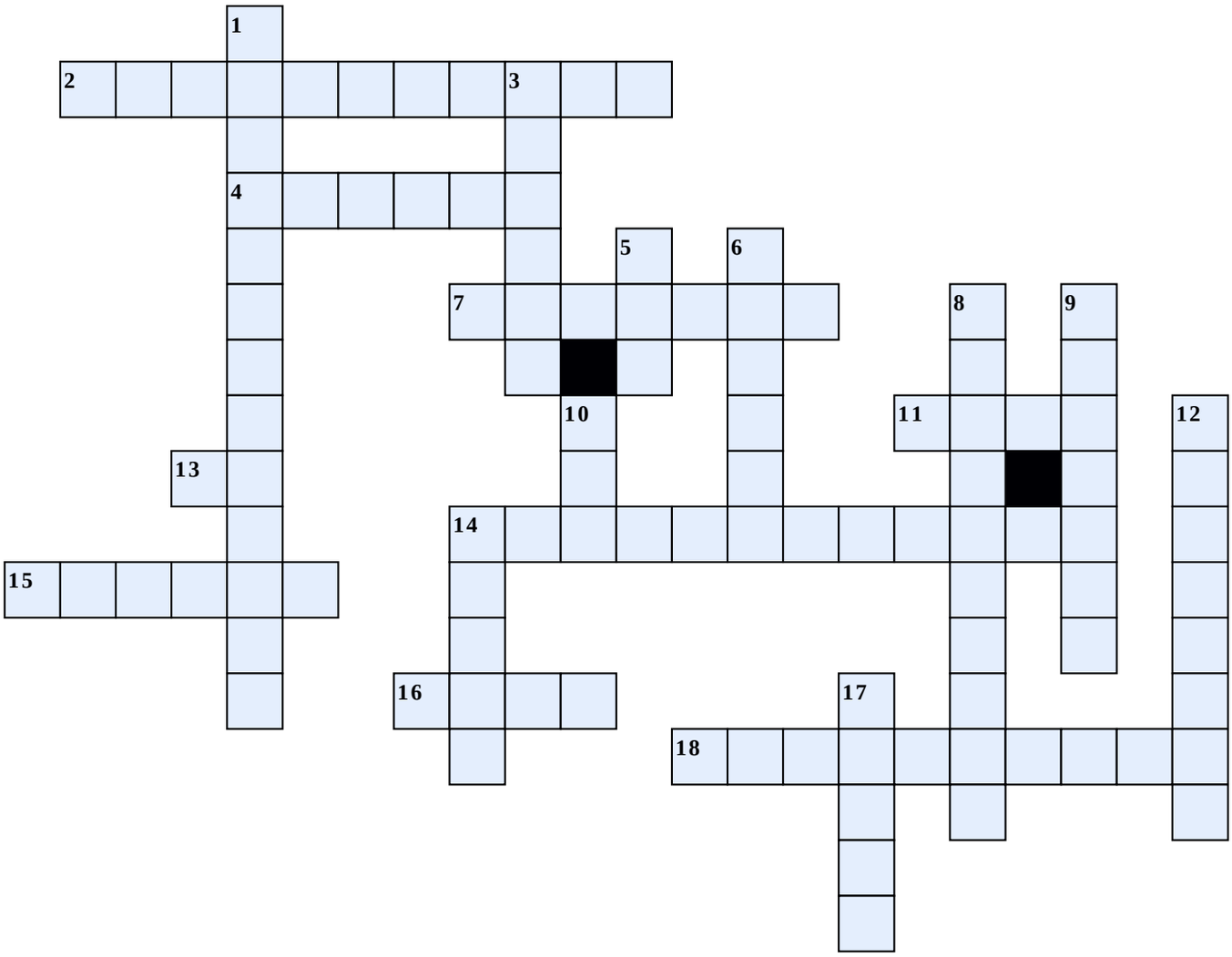
Rushing from Hamburg to Lübeck, against the sunset.

Natural signs of challenging wind speeds in Denmark…

Closer than 4km, just a shortcut away! Hit ⌘ + Z in case of wrong direction!



Kotlin Crossword Puzzle



Across

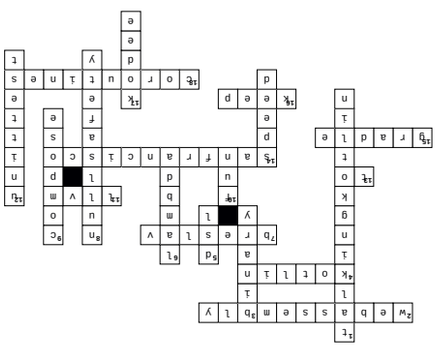
2. New technology to run code on web pages
4. Name of an island
7. Last name of first Kotlin lead language designer
11. Kotlin Native uses ... under the hood
13. Creates a Pair in Kotlin
14. City where the first KotlinConf took place
15. The build system with an elephant 🐘
16. Proposal for a new Kotlin language feature
18. Helps with asynchronous programming

Down

1. A podcast about Kotlin
3. The page number on page 9 is written in this form
5. Allows to build nicely readable APIs
6. Kotlin and the game Halflife have this in common
8. Feature Kotlin has that Java hasn't
9. UI Framework
10. Function
12. You can run this to check your code works
14. The new K2 compiler delivers better build ...
17. Kotlin mascot

Crossword puzzle built thanks to crosswordlabs.com

For solution: turn page upside down



fun code() **"Kotlin is beautiful".length**

magazineScope.cancel()

`have a nice Kotlin`()

x.com/funcode_mag

mastodon.social/@funcode_mag

kotlintoday.com