Building Kotlin-first Libraries

Kotlin London - 7th March 2024

Jamie Sanson

@jamiesanson

Hello! I'm Jamie Sanson. I've been working in Android for almost a decade now, and have recently made the move from Wellington, New Zealand over to London (as you can probably tell by the accent).

I'm working as a Staff Software engineer in the Mobile Platform space at Marks and Spencer, and I'm stoked to be here for the first Kotlin London in a while!

Agenda

1. Vintage Kotlin
2. "Modern" Kotlin
3. Designing a library API
4. Building Java-second
5. Tracking your API

@jamiesanson

Today I'm going to be talking about building Kotlin-first libraries, and there's a few things to cover!

First up, modern kotlin, and vintage kotlin, which feels like the inverse of modern to me!

Next up we're going to be diving in to library API design. There's a bunch of meaty topics in here - type theory, system design, you name it - but it's all backed by code samples, so should be alright!

We'll then look a bit at taking something Kotlin-y and making it Java-compatible, and finally look at keeping your lovely new API consistent.

Cast your mind back to the early days of Kotlin. Some of you may be relatively new to the Kotlin ecosystem, but we're talking way back - more than a decade.

Jetbrains introduced Kotlin to the world in 2012, as a new language for use in their IDEs. They wanted something like Scala in terms of language features, but wanted the language to compile *quickly* [1].

Kotlin went 1.0 in 2016, touted from the get go as a programming language for JVM and Android.

The language of Android
2017

@jamiesanson

In 2017, just one year after Kotlin reaches 1.0, Google announced first-class support for Kotlin in Android.

This announcement came alongside code labs and documentation from Google, as well as a java -> Kotlin converter.

In the early days of Kotlin support in Android, Google aimed to provide Kotlin APIs without rewriting their entire suite of libraries.

To do so, ktx (or Kotlin Extension) libraries were introduced. These libraries were thin layers on top of existing Java APIs - a sprinkling of syntactic sugar for the Kotlin early adopters.

Something to note here is that we've started out with Kotlin Library constrained to underlying Java mechanics. Sure, it minimised the amount of NullPointerExceptions apps saw, but the Kotlin facade was just Java with nullness.

Kotlin Multiplatform came along in late 2017, and started to lay the foundations for Kotlin in other places, like running on the JavaScript engine, or on native targets.

This changed things a little. If you write "common" Kotlin code, you're not writing directly on top of Java libraries & APIs anymore. You've just got the Kotlin Standard Library to deal with, which uses more of the language features Kotlin has on offer.

Then along came some other minor things..

Coroutines
2018

@jamiesanson

... like coroutines in 2018..

Android Jetpack Libraries 2019

@jamiesanson

... fully-Kotlin libraries for Android developers in 2019...

... and eventually you have Compose. The epitome of modern Kotlin - a declarative state management system build by both Jetbrains and Google.

Compose UI has changed the way we build Android apps, and if Jetbrains has anything to say about it it'll also change the way we build for desktop, web, and maybe even iOS?

Which takes us to where we are today. Kotlin as a language has grown like crazy compared to its original competitors like Java and Scala, and is ultimately a pretty different ecosystem nowadays.

What makes a library modern?

- Thoughtful on typing
- Frugal on language features
- Kotlin for Kotlin

@jamiesanson

This talk is about kotlin-first libraries, and to be kotlin-first, you need to be modern.

But what makes a library modern? It all comes down to using the right language features, thinking a little bit about designing your types, and leaning on the compiler to do the hard work for you.

To me, there's three things that show a library is modern and mature.

Aside - In the dependency tree

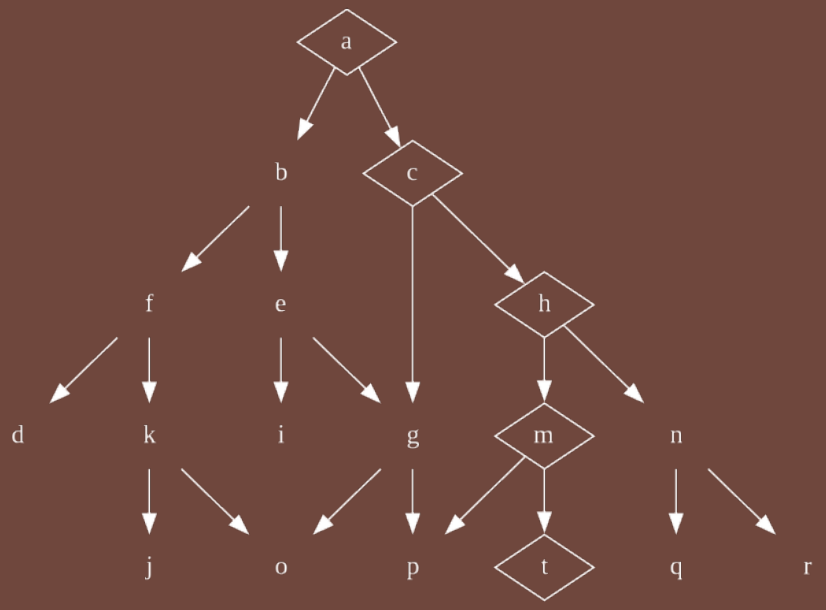
@jamiesanson

Before we get in to what I mean by those things, I figured we'd set the scene by briefly mentioning dependencies. This is less Kotlin-specific and more system-design-y, but it'll help to show the value of designing your APIs carefully.

All libraries are dependencies, but they differ by where they aim to sit in the hierarchy of design decisions in a consuming project.

For example, libraries can be leaf-level - think a simple UI component that plugs in to your project. Libraries can also be fundamental building blocks of your project. The former will have very little influence on how you change aspects of your project in future, whereas the latter most definitely will.

Aside - The Design Space tree [1]



1. https://two-wrongs.com/software-design-tree-and-program-families.html

@jamiesanson

I've recently come across this great article by a developer called Chris (I couldn't find his last name), who describes the overall design off a big complex project as a tree of decisions. I've added a link here, but don't worry about writing it down - I'll share the slides at the end.

The idea is that fundamental design decisions, like `a`, end up constraining how you design things further down the tree, like `m`.

I won't try to explain this in much more detail, as the article does a far better job of it than I could, and we've got Kotlin to talk about. The important thing to take away from this, is that our library could be an `m`, or it could be a `c`.

If we're designing something that sits in the center of a big dependency tree, we should make sure we think about how whatever we provide influences design decisions of consumers.

Thoughtful on typing

@jamiesanson

A good place to start with any system you're looking to build is to gather requirements to form an understanding of exactly the problem you're looking to solve.

When building a library, you've generally got a pretty good understanding of inputs and outputs, and what exactly it is you need to be able to do. The thoughtfulness around typing comes when you consider who is actually going to use your code.

Typing & the Law of Demeter

@jamiesanson

First up on Typing, let's chat about a term you may be familiar with - the Law of Demeter, also known as the Principle of Least Knowledge.

The Law of Demeter pretty much says don't talk to strangers. Good advice in general.

When applied to designing a type, it largely means don't give away too much information about your inner workings, and keep a nice and succinct API surface that doesn't leak abstractions all over the place.

Typing & the Law of Demeter

```
// Too much knowledge
fun <T> getFlagValue(
    name: String,
    deserialize: (JSON) -> T
): T
```

@jamiesanson

This example is pretty relevant, as you'll see later. Imagine you've got a feature flagging library that deals with JSON values under the hood, and you're looking to get a value for a given flag. This could be modeled in a bunch of different ways, but here's one quite flawed way of doing it.

In this example, consumers need to know quite a few things: the name of the flag (a given), the type they expect for a flag (also a given), and more importantly the fact that a Flag is JSON-backed and they need to figure out how to deserialize it.

The problem with this approach is the JSON bit. Every interaction with this API needs to be wary of an underlying fact of the system - it's JSON backed.

Typing & the Law of Demeter

```
class Flag<T>(
    val name: String,
    internal val deserialize:
(JSON) -> T,
)
```

@jamiesanson

Let's now look to improve this a wee bit by introducing a type to model this "Flag". All we're doing here is taking the three inputs from our first function - the name of the Flag, the type of the Flag, and some way of turning JSON into that type.

Typing & the Law of Demeter

```
// Juuuuuust right
fun <T> getFlagValue(
    flag: Flag<T>
): T
```

@jamiesanson

This better example then uses that Flag type, and doesn't actually change the inputs to the function at all. What it does do is package concerns more tightly - getting in to the "not talking to strangers" bit of the Law of Demeter.

By introducing a "Flag" type, we're minimizing the amount of knowledge *most* of the interactions with our library need to have. We still expose the notion of being JSON-backed when creating a Flag, but even this can be minimized by setting a default value for our `deserialize` input.

Something else to consider when designing your API is to consider what states are *valid*, and what you can do with types to minimize *invalid* states.

There's many a blog post on this topic, and you'll find my favorites[1] referenced in the slides. The gist of these is: the more you constrain your inputs and outputs, the less state checking you need to do, and ultimately your chance of bugs arising goes down.

Typing & invalid states

```
class Flag<T>(
    val name: String,
    internal val deserialize:
(JSON) -> T,
)
```

@jamiesanson

Going back to our flag example from before, we noted that we had three inputs - type, name, and how to get a type from JSON.

Clearly name is the "name" of the flag, right? This is probably something snake_cased, and refers to something you'd get from some kind of backend.

Typing & invalid states

```
val myFlag = Flag(
    name = "Jamie Sanson",
    deserialize = ...
)
```

@jamiesanson

But then along comes Jamie. He thinks "name" means his name, and doesn't think twice about it.

This is obviously a pretty contrived example, but it shows the point I'm trying to make pretty well. If you're using raw types in your API, chances are *someone* is going to pass the wrong thing in at some point, and end up in a weird state. You could argue that's on the consumer, but in general it's better to constrain your inputs as much as you can from the get-go.

Typing & invalid states

```
class Flag<T>(
    val name: Name,
    internal val deserialize:
(JSON) -> T,
) {
    value class Name(val value:
String)
}
```

@jamiesanson

A fix for this problem is to introduce a `value class` to wrap our input into something as little more meaningful.

This time when creating a Flag we *need* to provide a `Flag.Name`, making it super explicit that the string we're providing isn't just any old string.

The extra neat thing here is that in using `value class`es, we get the benefit of strong typing on our inputs, at no cost to memory usage at runtime!

Okay, we've talked typing, but we're not done yet. The next principle that I think makes a modern Kotlin library is a good understanding of language features - both what to use, and what not to use. When used correctly, library maintainers can describe exactly what inputs and outputs should be, produce APIs which can be easily added to in future, and give consumers the right level of access to change the behavior of the system without shooting themselves in the foot.

We've already seen that I think `value class`es are definitely worth using, but there's quite a bit more to think about here, both with your inputs AND outputs.

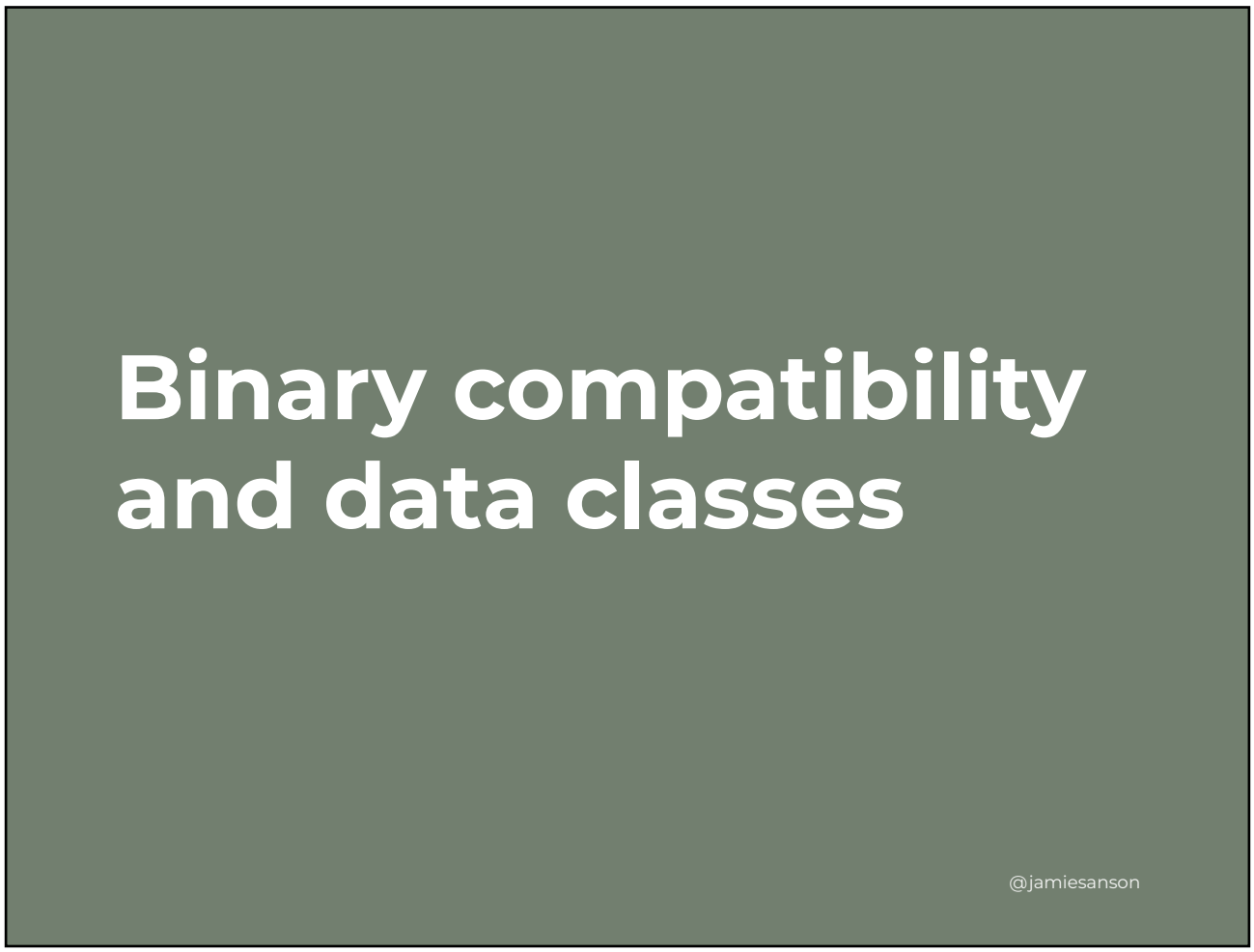
Language features - data classes

```
class Flag<T>(
    val name: Name,
    internal val deserialize:
(JSON) -> T,
) {
    value class Name(val value:
String)
}
```

@jamiesanson

Let's chat API maintenance, and jump back to our Flag example from before. The astute listener in the audience has probably noticed something curious - we're not using a `data class` here.

We're actually not using data classes for good reason - to allow us to add information to this type without producing a breaking change in binary compatibility for consumers. Let's dig in to this a little.



Data classes are a nifty shortcut in Kotlin that allows us to create struct-like types. Take a class with properties in it's constructor, whack a `data` on the front of it, and you get a bunch of functionality for free: Generated equals, hashcode and toString methods, as well as copy-constructor, and componentN functions.

The problem with data classes in public API comes from the latter two bits of free functionality - copy-constructors and componentN functions.

I'll using a Flag-related example to keep in theme, but know that I'm referring to yet another great blog post - this one by a very well-known Android developer named Jake Wharton[1].

Binary compatibility and data classes

```
 data class Flag<T>(
   val name: Name,
+  val description: String?,
   internal val deserialize: (JSON) -> T,
) {
+  constructor(
+      name: Name,
+      deserialize: (JSON) -> T
+   ) : this(name, null, deserialize)
}
```

@jamiesanson

Let's pretend our Flag is a data class for a moment, and we want to add a new property - "description". We're trying to not break the API for consumers using the constructor without a description, so also add in a new constructor matching the old signature.

Not bad, right? Well unfortunately, yes bad. Although this looks pretty benign, we're actually introducing TWO binary-incompatible changes, and that's all thanks to those things I mentioned earlier - copy-constructors and componentN functions for destructuring.

Binary compatibility and data classes

```
// Before change ✅
val (name, deserialize: (JSON) -> Int) =
flag

// After change 💥
val (name, deserialize: (JSON) -> Int) =
flag
            ^ 'component2()' function
returns
              'String?', but '(JSON) ->
Int' is expected
```

@jamiesanson

The easiest way to illustrate this is to consider what happens to destructuring when we add in parameters. If we add parameters wherever we want - in this case in the middle of our parameter list - we change the return type of the following `componentN` functions.

Here component2 used to return a lambda, but now returns a nullable String. Although consumers probably *aren't* relying on these methods, we're allowing them to through the use of data classes in public APIs.

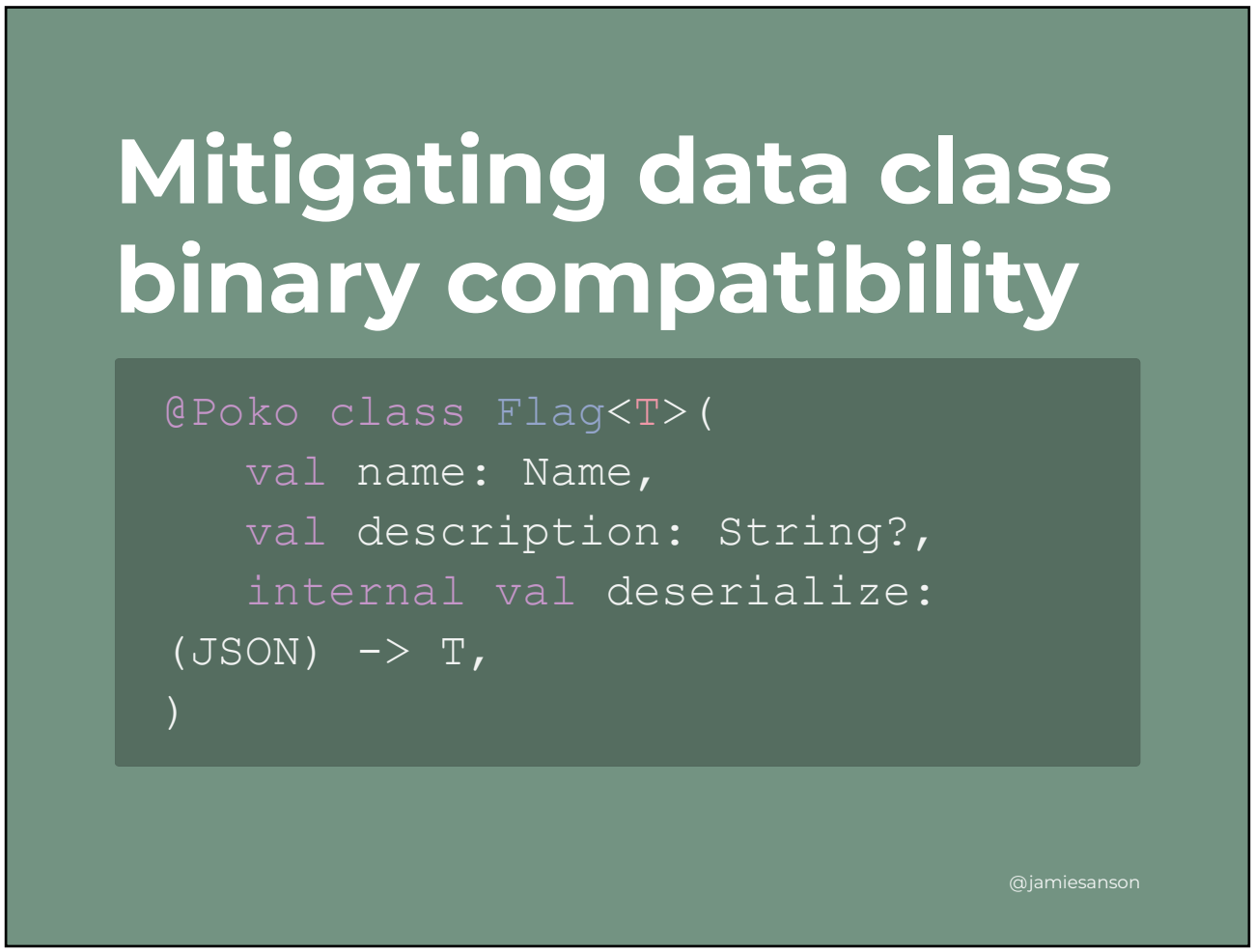
Copy-constructors break API in similar ways. In the interest of time I'll refer you back to that blog post I mentioned, which will be included in the slides!

Mitigating data class binary compatibility

@jamiesanson

The fix for these issues is largely to just not use data classes. This means writing your own equals and hashcode and toString functions, which feels archaic and no good.

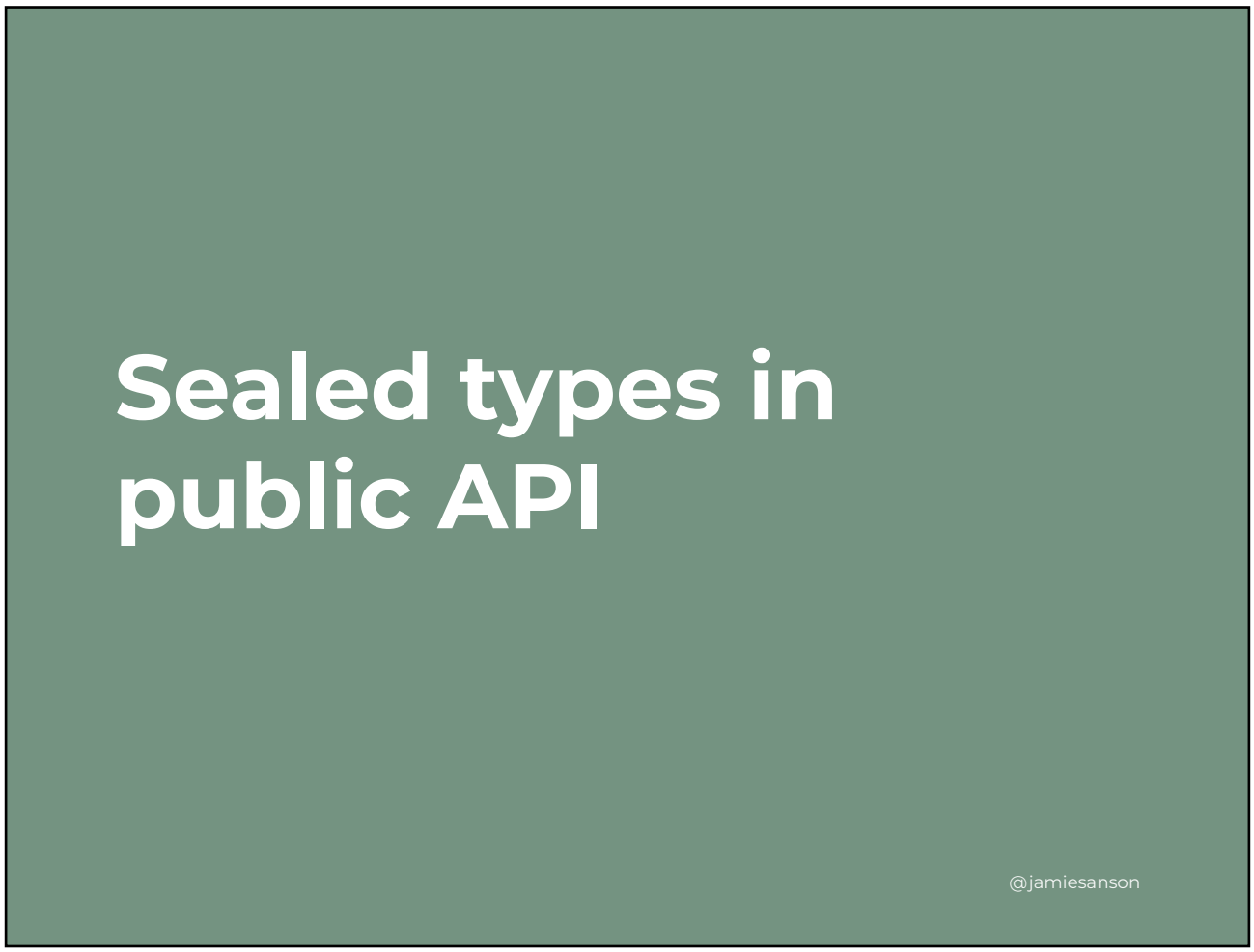
Luckily there is a happy middle-ground, and that's through the use of compiler plugins.



A very handy compiler plugin exists call Poko, which stands for Plain old Kotlin Objects.

This plugin gives us the nice, binary-compatible generated methods (equals, hashcode and toString), without any of the footguns. Swap out the `data` modifier with an `@Poko` annotation, and you're there! You'll still need a new constructor to not break Java consumers, but it's a lot less for us to think about as library maintainers.

There's a couple more nuances to consider here around the use of the builder pattern to avoid further Kotlin complexities - if you're interested in reading more in to this I'd definitely recommend having a read through Jake's post.



Another language feature to watch out for in your public APIs is sealed types. Kotlin has a few of these - `sealed class`es, `sealed interface`s, and `enums`. While these are great for modeling sum types - a type which could be one of a defined set of options - Kotlin makes sure you handle all these options.

Sealed types & exhaustive `when`

```
sealed interface FlagType {
    data object Feature: FlagType

    @Poko class Experiment(
        val id: String
    ): FlagType
}
```

@jamiesanson

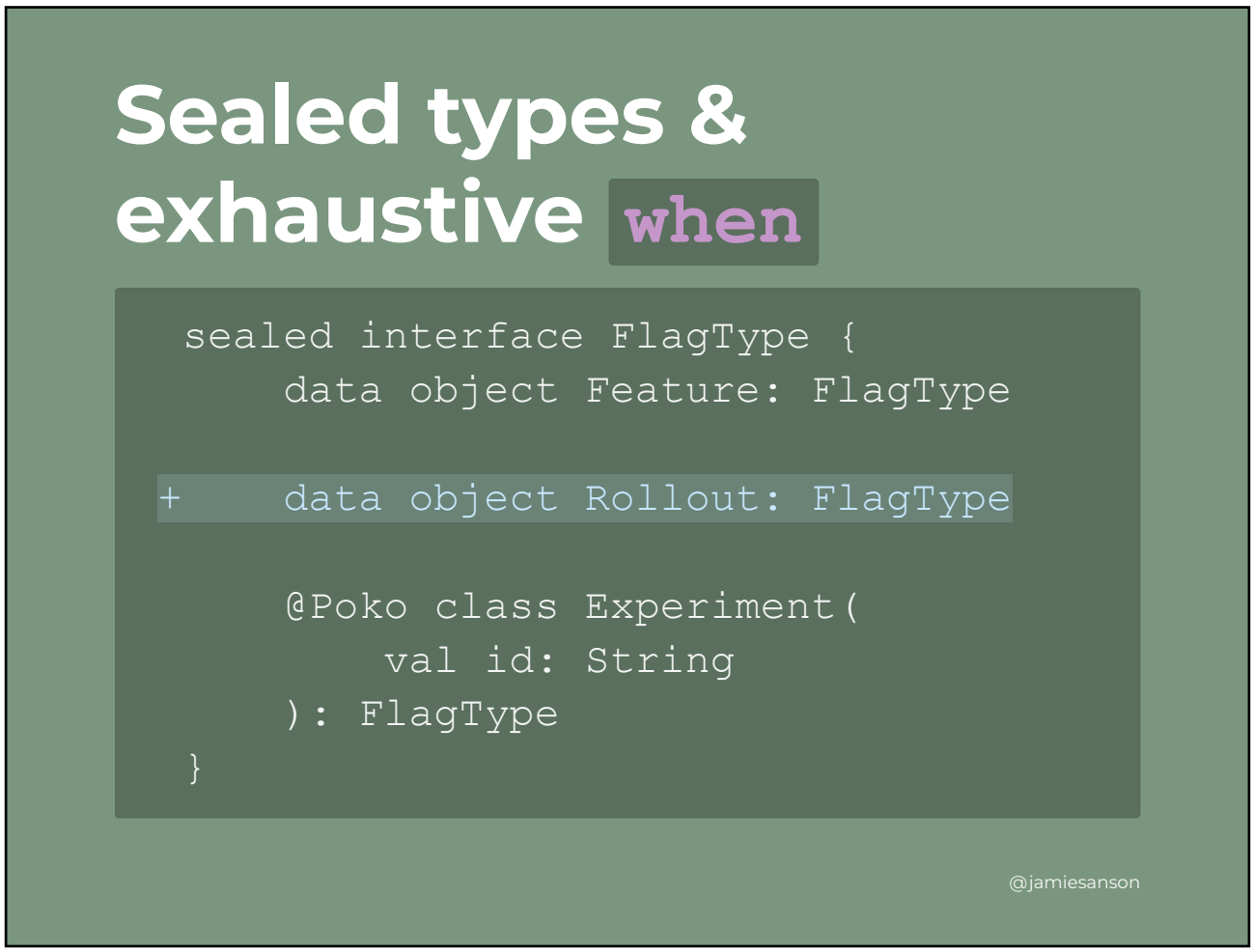
Let's add a "type" to our flag. For example, we know flags could either be feature flags, i.e turning something on and off, or they could be experiments, in which case they might need an ID or something.

Sealed types & exhaustive `when`

```
when (val type = flag.type) {
    is Feature ->
        doSomething()
    is Experiment ->
        experiment(type.id)
}
```

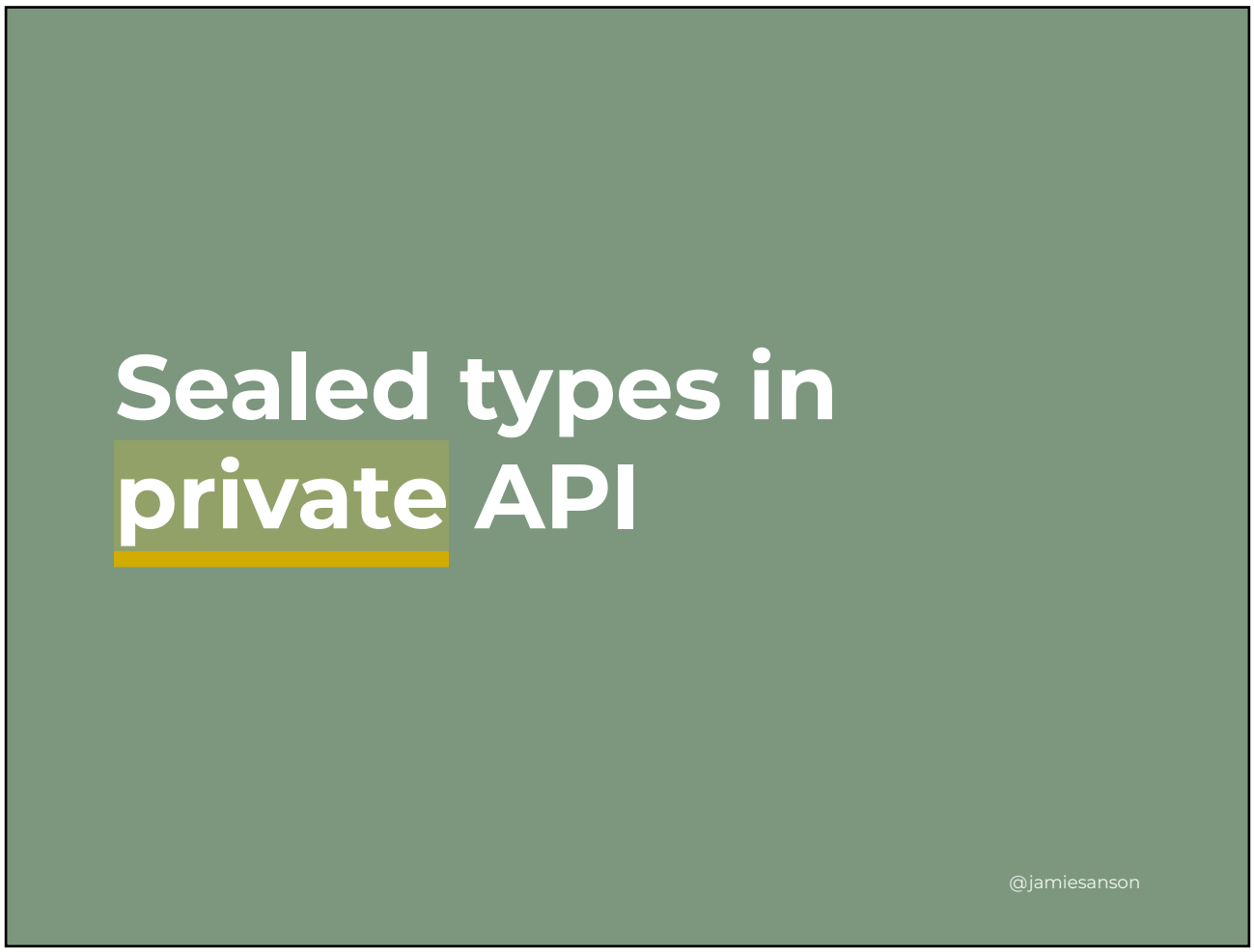
@jamiesanson

There's nothing stopping a consumer writing code that looks like this! For whatever reason someone might be switching on the type of the flag, and doing something with the result.



Further down the track, we might decide that there's actually a third flag type we want to model, that were previously bundled in with features. Rollout flags might represent flags that exist to ramp a feature up from 0-100% adoption, and can then be removed later on.

Unfortunately, we're using a sealed interface, which means this addition *can't be made* without breaking those exhaustive when's that might exist in consuming codebases.



The problem is that sealed types are actually really handy. We want to be using them in our own code, as they let the compiler do more work on our behalf.

The good news is we can definitely keep using sealed types internally by just being slightly more mindful of how we define what's public and what isn't.

Sealed types in private API

```
class User internal constructor(
    internal val type: Type,
) {
    constructor(id: ID) : this(
        type = Type.Identified(id = id),
    )

    companion object {
        val Guest: User get() = User(Type.Guest)

        value class ID(val id: String)

        internal sealed interface Type {
            data object Guest : Type
            value class Identified(val id: ID) : Type
        }
    }
```

@jamiesanson

There's a lot of code on screen here, but bear with me.

This example shows how we might model a `User`, which is either a Guest, or someone identified through an ID.

We give consumers two public entry-points to get an instance of this thing. One constructor taking an ID, and a Guest getter on the companion object.

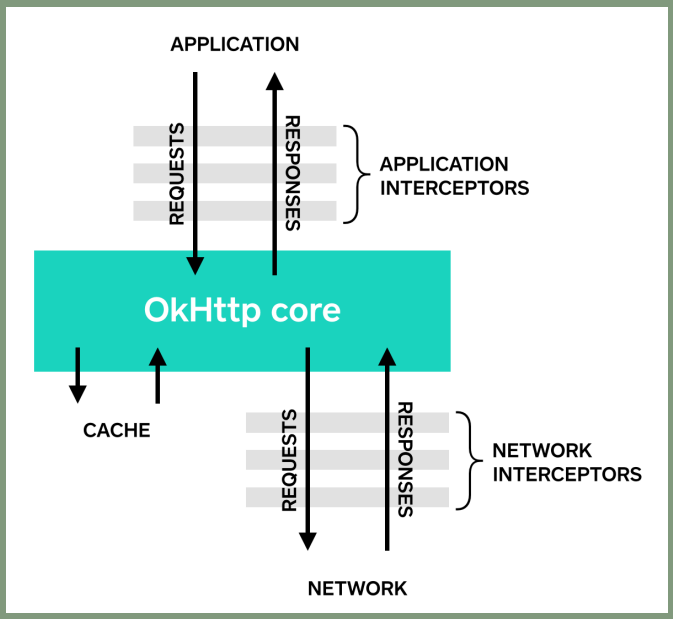
Consumers now can't do much at all with this `User` type, but internally we're able to get all the benefits of sealing, and can even use language features like data classes without worrying about API incompatibility.

Adding a new case to `Type` would still break internal API, but that's kind of what we expect!

There are a few other Kotlin features you should be wary of when building your public API. This one's again more general, though, and applies to most programming languages with functional paradigms going on.

Having inputs be functions which end up being composed internally is a neat way to abstract operations away, but they can quite easily get out of hand.

Interceptor pattern - OkHttp [1]



1. https://square.github.io/okhttp/features/interceptors/

@jamiesanson

Functions which end up being composed to create a chain of behavior is super powerful.

An example of these being used for good is in a library called OkHttp. This library models a call chain to an Http endpoint as a composition of interceptors, which can modify and even veto requests & responses as they pass through the stack.

You can plug in on either side of OkHttp core, giving you full access to the lifecycle of a request, with the ability to rewrite requests and responses as they go!

This style of design is known as the Interceptor pattern. It's super powerful, and allows libraries - in this case OkHttp core - to define extensible actions while having no knowledge of whatever random functionality you've decided to plug in.

This ends up being very useful, and a big reason why OkHttp has been so well loved over the years, but it also exposes a massively broad integration point.

Interceptor pattern

> Returned JSON content is always prepended with a while(1){} clause to mitigate abuse.

@jamiesanson

Having a broad integration point for something as generic and at times as fiddly as HTTP works out quite well.

For example, imagine you're reading through an API spec, and see this line of documentation.

Aside from the fact that prepending a while true loop to a JSON document is completely ridiculous, we can abstract this away from almost all of our networking stack through an interceptor!

Interceptor pattern

```
class RemoveBodyPrefixInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain):
Response {
        val response = chain.proceed(chain.request())

        if (response.body == null) return response

        val newBody =
            response.body
                ?.string()
                ?.removePrefix("while (1) {}\n")
                ?.toResponseBody()

        return response.newBuilder()
            .body(newBody)
            .build()
    }
}
```

@jamiesanson

Here we have an actual implementation of one of these interceptors - the one I've written to deal with this abuse mitigation thing that Adobe likes to include in their APIs.

Although inputs and outputs are nicely typed, this `Chain` thing refers to a composition of functions.

What this interceptor allows us to do is make the call, and rewrite the response body if it includes the while true thing.

Serialization plugs in at a different layer, and knows nothing at all about this weird quirk!

As you can see, this pattern is super useful!

That's not to say that *every* library should use this pattern. OkHttp is modeling something massively generic, and often times massively intricate. It's also almost exclusively used in a very low layer of that decision tree thing we talked about earlier.

That doesn't change the fact that consumers can come along, pass in an incorrect interceptor, and break all networking across their entire project. Obviously it's on the consumers to **not** do this, with great power comes great responsibility and all that, but it raises an interesting thought - do we need something this general?

I think the answer, in most cases, is "probably not".

- If your library involves screwing in screws, it's probably better to give your users a screwdriver rather than a hammer.
- If your library involves getting the value of a feature flag from one of a bunch of different sources, you probably don't need an interceptor chain because it's probably a little easier to work with than HTTP.

In a nutshell, the point I'm trying to make in this section is that typing matters for your consumers, and there's a bunch of language features you can avoid that'll make your life easier in future.

That's not to say you shouldn't use all the features that makes sense to you internally. If you need sealed types, data classes, and function composition, then use it! But if you want to use them publicly, consider what it means for maintenance and your consumers.

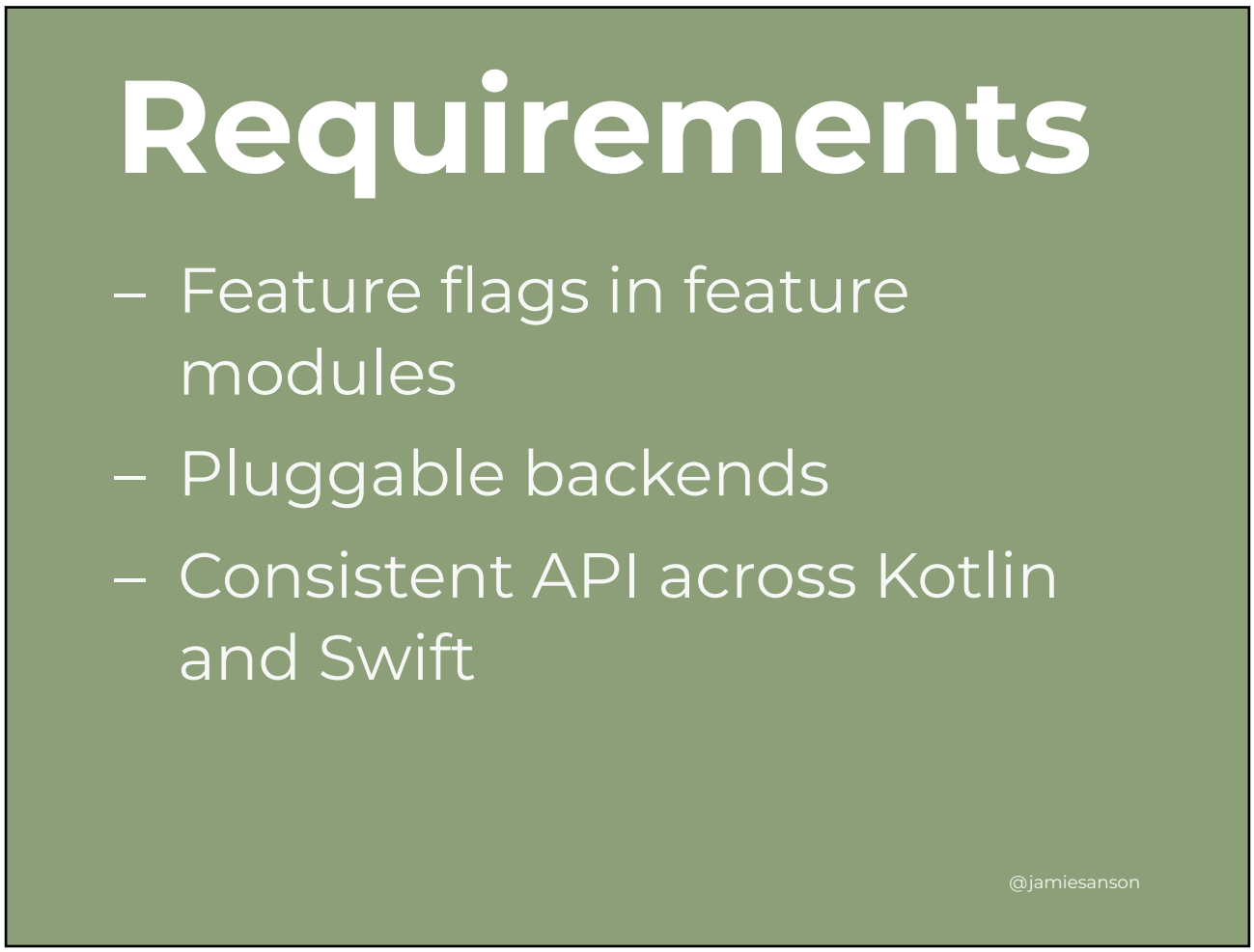
Building a modern Kotlin library

Example: Feature flagging

@jamiesanson

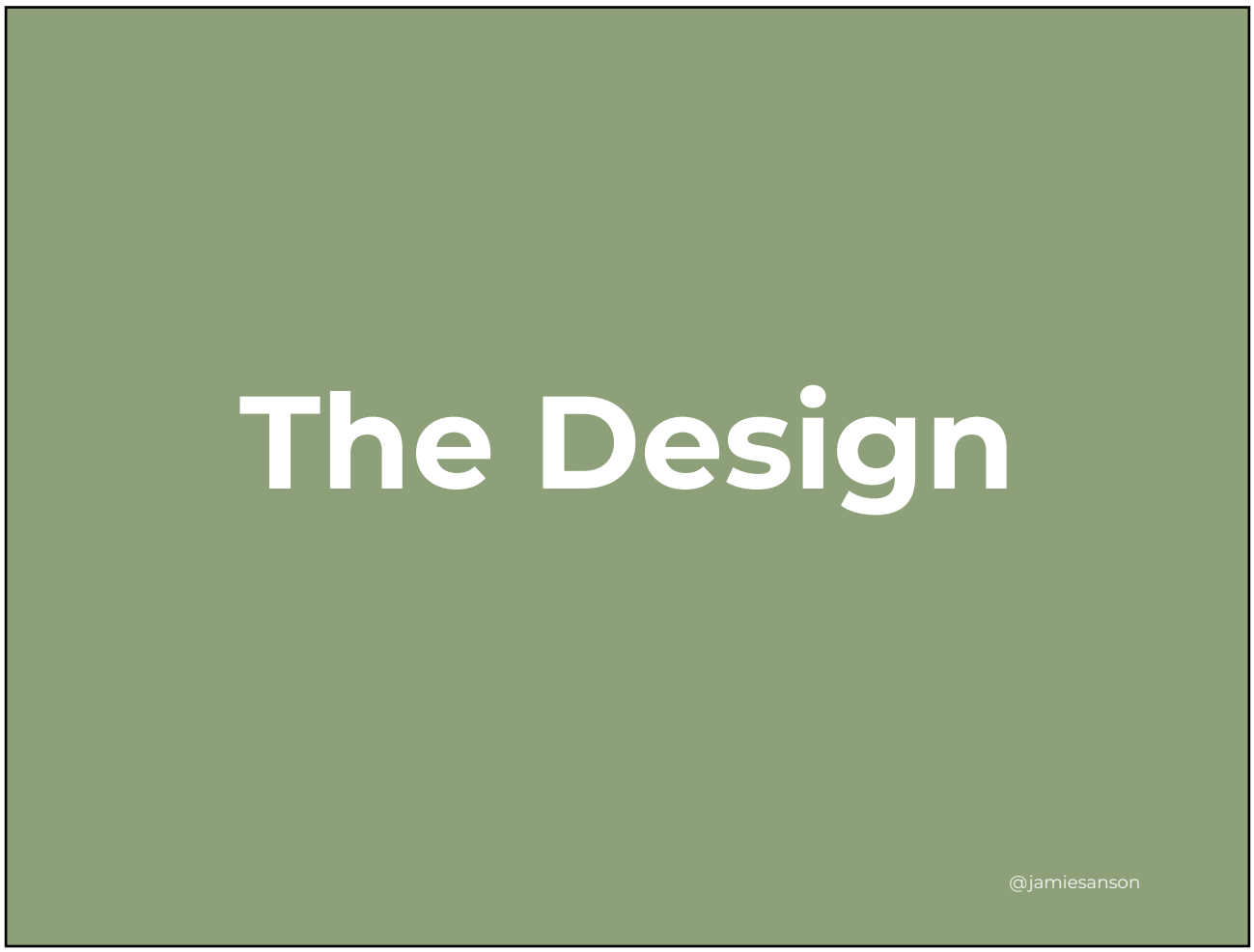
We've talked a lot about things to consider when building a Kotlin library, but let's look more at an example.

At M&S, we've recently re-thought how we do feature flagging. This lets us avoid vendor lock-in, and work more easily in a highly modularized codebase.



The feature flagging work had a few core requirements.

1. Allow feature flags to be defined per-module, meaning adding of flags doesn't invalidate build caches in other modules
2. Pluggable backends. Vendor lock-in is something we want to avoid, and allowing consumers to plug in whatever they want as a source of flag values was something we needed to consider from the get-go
3. API Consistency. We're a Mobile platform team, and at M&S we largely do mobile via Native apps.



With requirements sorted and a crack team of engineers assembled (myself and the iOS guy - Daniel Tull), we got to work thinking about design.

Now in this presentation we've talked a lot about design principles, and how you figure out what your API surface should look like. In practice we didn't actually *start* with principles.

Abstractions

- Flag
- Source of Flag values

@jamiesanson

We started out by thinking about abstractions. To fit the requirements, we only really needed two things - something to model a Flag, and something to model a thing that provides values for flags.

Flag

```
@Poko class Flag<T>(
    val name: Name,
    val description: Description,
    internal val defaultValue: ()
-> T,
    internal val deserialize:
(JSON) -> T,
)
```

@jamiesanson

We started out thinking about a Flag, specifically what kind of information it should be holding on to. What we landed on is something very similar to what you've seen already.

We know a flag needs a name. We WANT a flag to have a description and default value when no providers have a value for it.

We also figured it'd be cool to have a Flag know how to deserialize itself, meaning the library itself doesn't need to know about any custom serialization setups your app might have.

Why JSON? We know we need multiple sources, and to compose them together we're going to need a common language. We picked JSON given some of the third-party tooling we were integrating with. We could have used anything here, really, but went with JSON as it worked for our use-cases!

FlagProvider

```
typealias FlagProvider =
(Flag.Name) -> JSON?
```

@jamiesanson

By using JSON as this common language between flags and the thing providing their values, we can then model a flag provider as a simple function - something that takes a Flag name, and maybe gives you a JSON value back.

We started here, but as the design <> build iteration went on we found a few drawbacks of modeling this so simply.

FlagProvider

```
fun vetoProvider(
    delegate: FlagProvider
) = { name ->
    // Get the value
    val result = delegate(name)
    // Throw it out
    null
}
```

@jamiesanson

The main drawback from using functional types as input, as we chatted about earlier, is that they end up maybe being a little too powerful. This example shows how you can compose one of these functions to do pretty much whatever you want - any kind of side effect, any kind of interception, whatever!

While powerful, this allows consumers to change the behavior of the library entirely. The problematic thing here is that we can't predict how consumers will use these inputs, meaning we can't write tests for these inputs.

Constrained functional inputs

@jamiesanson

Alright, so we kind of want functional inputs, but we don't want consumers to plug in absolutely any behavior they'd like. How do we model this??

We largely landed on the solution through trial-and-error, and considering side-effects like analytics when flags are evaluated.

The result is something that keeps the public-facing API pretty functional, but constrains how much they can be composed - let's see a code example.

Constrained functional inputs

```
fun FlagProvider(
    name: String,
    resolve: (Flag.Name) -> JSON?
): FlagProvider
```

@jamiesanson

Instead of simply using a type-alias, let's introduce a new type. Our public API doesn't change all that much!

We're now using a top-level function to create a FlagProvider, which allows us to pass in other information, like the "name" of a provider - useful for debugging and side-effects in general.

Constrained functional inputs

```
class FlagProvider internal
constructor(
    internal val resolve:
(Flag.Name) -> Output,
)
```

@jamiesanson

Our FlagProvider implementation now uses a slightly different looking function under the hood. The only real difference here is the return type - this thing called `Output`. But what's an output?

Constrained functional inputs

```
internal data class Output(
    internal val outcome:
Outcome,
    internal val effect: Effect,
)
```

@jamiesanson

Things get a bit weird from here. Our "Output" is an "Outcome" and an "Effect"?

These types exist to do two separate things. The outcome describes the result of a flag resolution. It's internal, so it can be sealed, and be one of either a "Found", "NotFound" or "Failure" (for the case where a provider throws).

An "Effect" is another internal thing - this one representing a function to call as a side-effect to flag evaluation.

I won't go into the details of either of these types in this talk, but the important thing to note is the **visibility modifiers**. We've taken a function which deals with public types, i.e the Flag name -> JSON thing, and wrapped it into something internal.

This gives us total control over the underlying functionality, meaning we could completely rework this Output thing without impacting consumers at all!

Keeping functional behaviours internal

```
fun FlagProvider(vararg providers: FlagProvider):
FlagProvider =
   FlagProvider composite@{ flagName ->
        for (provider in providers) {
            val output = provider.resolve(flagName)

            if (output.outcome is Outcome.Found) {
                return@composite output
            }
        }

        FlagProvider.Output(
            outcome = Outcome.NotFound(flagName),
            effect = Effect.Empty,
        )
    }
```

@jamiesanson

The neat thing in keeping the implementation of this flag provider thing a function is that we still get all the same benefits of writing functional code.

Consumers probably want more than one flag provider - maybe one for Firebase Remote Config, and another for local overrides. We can write a composite flag provider using the underlying functional type pretty easily!

Bringing it all together

@jamiesanson

The one thing we haven't mentioned is the API for taking a flag, and getting a value. Luckily for you, you've already seen it!

Bringing it all together

```
class FlagProvider(...) {
    fun <Value> valueOf(flag:
Flag<Value>): Value
}
```

@jamiesanson

Our API for getting the value of a flag looks like this - the same example we saw earlier when talking about the Law of Demeter.

Consumers using the flagging library can create an instance of this FlagProvider somewhere centrally, pass it around, and use it to get the value for a Flag they've defined in their feature modules.

A handy side-effects of this is that those feature modules are now not tied to *any* implementation of something at a lower level in their own project. This means they can be compiled independently so long as *something* has a `FlagProvider` to contribute.

Extra - Reactive Flags

@jamiesanson

A requirement that we deemed out of scope for the first cut is to support reactive flags. By reactive flags, we mean flag values that are observable.

Our whole library is currently synchronous, being made out of one-shot functions. How would we go about supporting reactive flags?

Our first thought was "oh dear, I guess we need something reactive at the heart of this thing. Back to the drawing board"

Our second thought was "oh dear oh dear, which framework do we use when we don't know what consumers are using"

Our third thought: "maybe we shouldn't"

As it turns out, the third thought was correct.

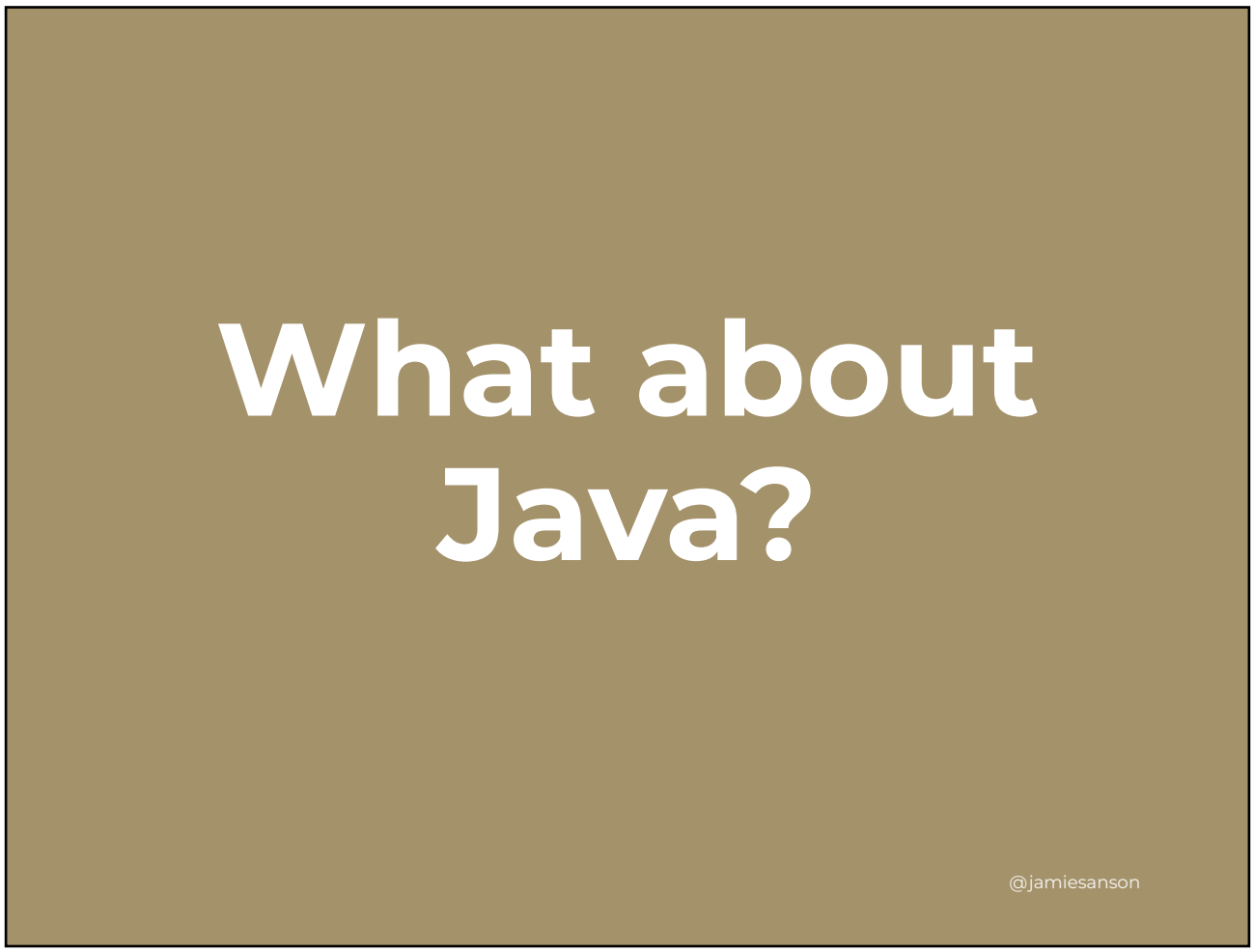
Extra - Reactive Flags

```
val flagProvider:
StateFlow<FlagProvider>
```

@jamiesanson

Rather than making the return value of the `valueOf` function reactive, what if we simply let consumers create their own reactive thing from their data source?

Instead of reactive flag values, we settled on promoting reactive flag providers. This allows consumers to opt-in to reactive-ness if they need it, or use the synchronous APIs if they don't!



At this point we have a pretty robust, very easily testable library! We've implemented this alongside a Swift version, which has been designed in the exact same way. The only thing left was to write some docs, publish it, and release it to the world!

In reality that wasn't quite the case. We started out with the assumption that our consumers would all be Kotlin-based, so didn't hold back from using things like functional types, inline reified functions, and value classes. Unfortunately, our first consumer was a Java 8-based Android app.

Supporting Java retroactively

@jamiesanson

We had a few problems to solve in supporting Java consumers nicely. What do we do about value classes? How do we model functional types? What about reified generics? And how do we keep Kotlin and Java APIs separate?

Supporting Java retroactively

```
// For use in Java 8 😢
fun FlagProvider(
    name: String,
    resolve: (String) -> String?
): FlagProvider
```

@jamiesanson

The solution for all of these problems was upsetting - we needed to relax all the beautiful type-safety we'd just put together.

This function did the job, and allows Java consumers to pass in a lambda for the "resolve" bit.

Supporting Java retroactively

```
// Usage from Java 8 😕
FlagProvider flagProvider =
    FlagProviderKt.FlagProvider(
        "name",
        flagName -> null
    );
```

@jamiesanson

Calling this function ends up being a touch awkward. It's exposed as a static function on a type generated by Kotlin, which by default is based on the name of the file.

Thankfully, Kotlin has a bunch of useful annotations we can apply to turn this output into something a little nicer.

Supporting Java retroactively

```
@file:JvmName("FlagProviders")
package
com.marksandspencer.flagging

⋮

@JvmName("create")
fun FlagProvider(...): FlagProvider
```

@jamiesanson

The JvmName annotation tells the Kotlin Compiler to emit a different name for the element it's attached to.

We can use this annotation to change the names of both the generated class for holding top-level functions, and the function itself!

By using @file before the package, we can tell the compiler that we want to name the class "FlagProviders", and that our factory function should be called "create".

Supporting Java retroactively

```
// Usage from Java 8 🙂
FlagProvider flagProvider =
    FlagProviders.create(
        "name",
        flagName -> null
    );
```

@jamiesanson

We now don't have weird naming! Very nice.

The next step - hide the other, Kotlin-specific APIs from Java. Turns out Kotlin has another handy annotation for this!

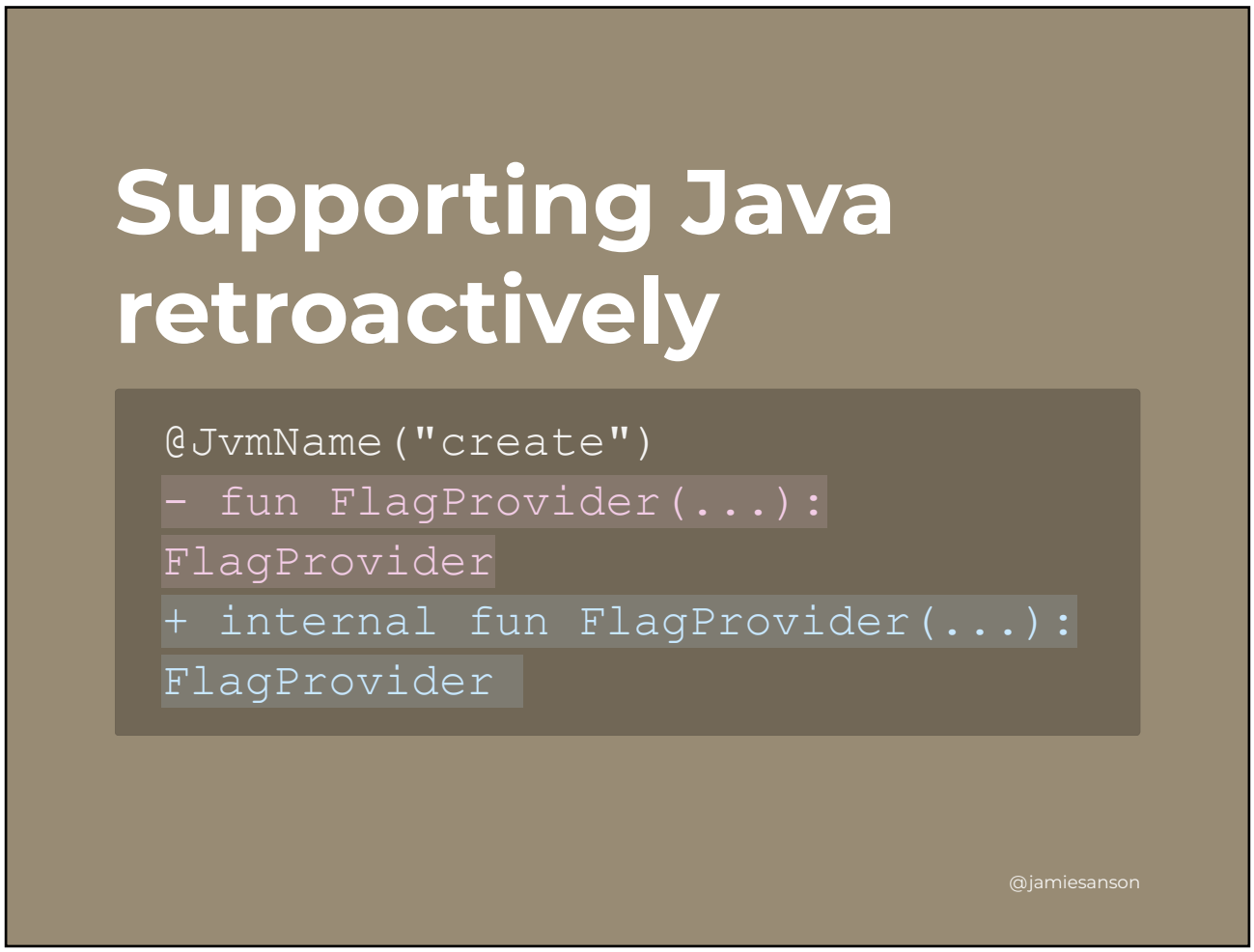
Supporting Java retroactively

```
@JvmSynthetic
fun FlagProvider(
    name: String,
    resolve: (Flag.Name) -> JSON?
): FlagProvider
```

@jamiesanson

Introducing `@JvmSynthetic`. This handy annotation is something you'll rarely need, but when you DO need it you'll be thankful it exists.

By annotating an element as synthetic, we're telling the compiler to mark this as synthetic in the generated Java bytecode. This makes the element effectively inaccessible from Java, while keeping it available to Kotlin sources!



But what about going the other way around? Can we hide those nasty, loosely typed declarations from Kotlin consumers?

Turns out, yes! This one's a little less nice. The `internal` visibility modifier doesn't have a Java equivalent, so the Kotlin compiler simply marks `internal` members as public in the generated bytecode!

Kotlin knows about `internal` through metadata added to these classfiles, but from a Java consumers perspective, they're free to use!

We can exploit this behavior by simply marking the things we don't want Kotlin seeing as `internal`. Java consumers get to see em, Kotlin does not. This *does* result in warnings in Java projects, but they can be easily suppressed - a bit of a trade-off, but it means we get to keep our nicely typed API!

Keeping your API consistent

@jamiesanson

One last quick tip. You've worked hard on building a Kotlin-first library, with a well considered API. How do you ensure you don't break things for your consumers?

Enter: Metalava

Metalava API tracking

@jamiesanson

Metalava is a tool created by Google, used primarily for API tracking in public-facing libraries used by Android developers.

Metalava builds a "signature" of your public API, which can then be used to compare the compatibility of that API from version to version of your code!

The signature for our "flag" type looks a little like this:

Metalava API tracking

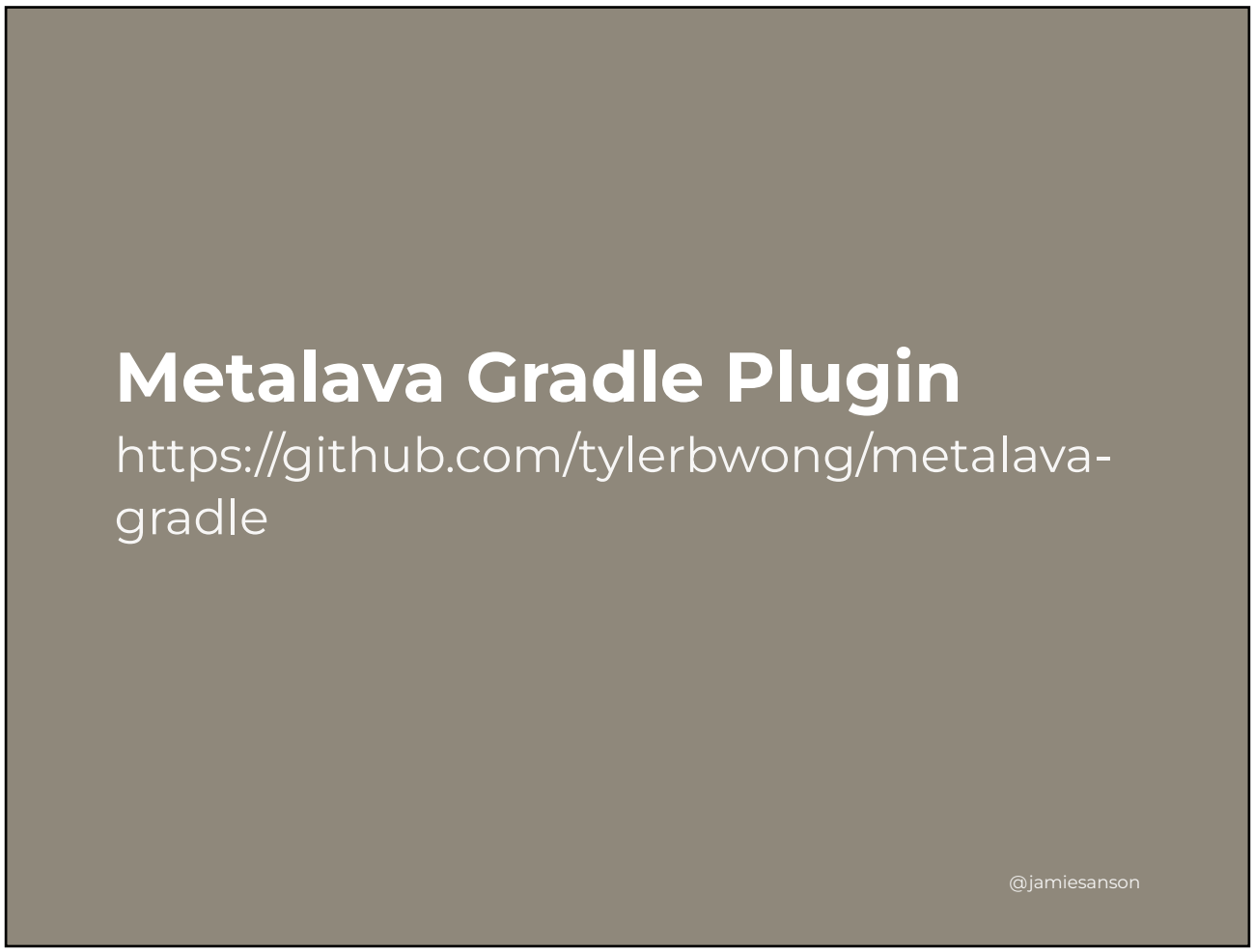
```
@dev.drewhamilton.poko.Poko public final class
Flag<Value> {
    ctor public Flag(String name, String
description, kotlin.jvm.functions.Function1<? super
com.marksandspencer.flagging.JSON,? extends Value>
decode, kotlin.jvm.functions.Function0<? extends
Value> defaultValue);
    method public String getDescription();
    method public String getName();
    property public final String description;
    property public final String name;
    field public static final
com.marksandspencer.flagging.Flag.Companion
Companion;
  }
```

@jamiesanson

This is clearly pretty terse, but it's not really intended to be human readable. If we generate this signature and check it into source, we can then use it to compare with the current reference.

For example, if I remove the "description" property accidentally my CI can now explicitly tell me I've made a breaking change, and I can then take steps to make sure I *don't* break my consumers!

The real utility of Metalava comes in when you inadvertently make changes that'll break your API. You'd probably be aware when you make the change that removing a property will break things, but as we talked about earlier, it's not that hard to make breaking changes through language features like data classes.



Metalava itself is open-source, but it's not all that convenient on its own as it's simply packaged as a jar. Luckily, the world of open source exists!

If you're building a Kotlin project, changes are you're using Gradle for build tooling. A gradle plugin exists that does all the wiring up for you, and exposes a couple of tasks for generating and validating these API signatures.

If you're interested in Metalava, I'd highly recommend checking this out.

Conclusions

@jamiesanson

Whew, that's quite a lot of content for a Thursday evening.

In conclusion, if you're building a kotlin library

Conclusions

- Think about typing
- Be wary of API foot guns, like data classes
- Think Kotlin, first. Don't worry about Java!

Questions

@jamiesanson

If you're after the slides, you can find them at this link. You'll have access to all the speaker notes, as well as links to all the posts I've referenced throughout. Cheers for listening!